Compiling with Consumers

Dissertation

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

vorgelegt von
Marius Müller, M.Sc.
aus Balingen

Tübingen
2025

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

Tag der mündlichen Qualifikation:	24.07.2026
Dekan:	Prof. Dr. Thilo Stehle
1. Berichterstatter:	Prof. Dr. Klaus Ostermann
2. Berichterstatter:	Prof. Dr.-Ing Mira Mezini
3. Berichterstatter:	Assoc. Prof. Dr. Dariusz Biernacki

Abstract

In our highly digitalized world, computer programs are ubiquitous. There is a multitude of high-level programming languages with complex language features that facilitate writing code in a structured way. Eventually, however, this code must be executed on real hardware, which is much less structured and cannot express complex high-level features directly. Compiling a program from a surface language, used by programmers to write their programs, into instructions a real machine understands hence is a complicated task. A compiler should not only generate machine code that correctly implements the intended behavior of the program written in the surface language, but the generated code should also execute efficiently. This complicated task is made possible by employing a compiler intermediate representation that bridges the gap between high-level conveniences and low-level concerns.

Modern functional programming languages tend to exhibit a particularly high level of abstraction. While this often enables programmers to express programs in a concise and elegant way, it also makes the compilation of such languages particularly challenging. This is exacerbated by the fact that modern programs often continually interact with external components, which must be done in an asynchronous way to avoid blocking execution while waiting for the result of the external component. Asynchronous communication requires the ability to express non-local control flow. Compilers typically deal with this in one of two ways. Either they add support for some kind of control operator in the runtime system, or they apply a translation to express non-local control effects in terms of other constructs present in the compiler intermediate language.

One possibility for expressing non-local control flow is to use an intermediate language that exhibits *explicit consumers*. A well-studied example are intermediate representations in *continuation-passing style* (CPS). In continuation-passing style, the continuation, i.e., the rest of the computation, is explicitly available as a term in the language. Languages in continuation-passing style have long been established as an option for compiler intermediate languages. A different option that has received much less attention so far are languages based on *sequent calculus*. Sequent calculus is the sibling calculus of natural deduction, known from logic. While natural deduction has long been used as a basis for functional programming languages, through its correspondence with the lambda calculus by the Curry-Howard isomorphism, it took until relatively recently to find a satisfying term-assignment system for the classical sequent calculus. In contrast to natural deduction and lambda calculus, which are biased towards proofs corresponding to *producers*, classical sequent calculus and languages based on it treat proofs and refutations, which correspond to consumers, in a symmetric way. This makes such languages an interesting alternative to continuation-passing style for intermediate representations.

In this thesis, we examine two different ways how explicit consumers can be used in the compilation of functional programming languages. First, we consider intermediate languages in continuation-passing style and translations into them. A question that naturally arises in this context is whether it is possible to translate from continuation-passing style *back to direct style* in which programs are usually written. This is not only of theoretical interest, but there is also a practical aspect. While continuation-passing style has many benefits for intermediate representations, it also comes with downsides compared to direct style. In particular, many platforms provide a call stack, which programs in continuation-passing style do not make use of. A suitable translation back to direct style could enable a compiler to reap the benefits of both styles. In the first part of this thesis, we thus present the design of a translation back to direct style having many properties that are important in the context of a compiler, laying the groundwork for making such a translation practically applicable.

In the second part of this thesis, we go in the other direction, compiling down to machine code. However, instead of using continuation-passing style, we examine intermediate languages based on classical sequent calculus. We present a complete compilation pipeline, starting from a functional surface language with interesting features, and ending with the generation of machine code that can be executed on a real computer. We identify a normal form of our sequent-

calculus-based intermediate representation that is suitable for a surprisingly direct way of code generation. The symmetric structure of languages based on classical sequent calculus makes them a promising target, also with regard to optimizations, and our pipeline forms the basis for future investigations.

Zusammenfassung

In unserer hochgradig digitalisierten Welt sind Computerprogramme allgegenwärtig. Es gibt eine Vielzahl von Hochsprachen mit komplexen Sprachmerkmalen, die das strukturierte Schreiben von Code erleichtern. Letztlich muss dieser Code jedoch auf realer Hardware ausgeführt werden, die weitaus weniger strukturiert ist und komplexe hochsprachliche Merkmale nicht direkt ausdrücken kann. Ein Programm aus einer Ausgangssprache, in der Programmierer ihre Programme schreiben, in Befehle zu kompilieren, die eine reale Maschine versteht, ist daher eine komplexe Aufgabe. Ein Compiler sollte nicht nur Maschinencode erzeugen, der das beabsichtigte Verhalten des Programms korrekt umsetzt, sondern der erzeugte Code sollte auch effizient ausgeführt werden. Diese anspruchsvolle Aufgabe wird durch den Einsatz einer Zwischenrepräsentation im Compiler ermöglicht, die die Lücke zwischen hochsprachlichen Annehmlichkeiten und maschinennahen Anforderungen überbrückt.

Moderne funktionale Programmiersprachen zeichnen sich oft durch ein besonders hohes Abstraktionsniveau aus. Dies ermöglicht es Programmierern zwar häufig Programme auf eine prägnante und elegante Weise auszudrücken, macht jedoch zugleich die Kompilierung solcher Sprachen besonders herausfordernd. Erschwerend kommt hinzu, dass moderne Programme oft kontinuierlich mit externen Komponenten interagieren, was asynchron erfolgen muss, um die Ausführung nicht zu blockieren, während auf das Ergebnis der externen Komponente gewartet wird. Asynchrone Kommunikation erfordert die Fähigkeit, nicht-lokale Kontrollflüsse auszudrücken. Compiler gehen damit typischerweise auf zwei Arten um: Entweder sie fügen dem Laufzeitsystem Unterstützung für eine Art von Kontrolloperator hinzu, oder sie wenden eine Übersetzung an, die nicht-lokale Kontrolleffekte durch andere Konstrukte ausdrückt, die in der Zwischenrepräsentation des Compilers bereits vorhanden sind.

Eine Möglichkeit, nicht-lokalen Kontrollfluss auszudrücken, ist die Verwendung einer Zwischenrepräsentation mit *expliziten Konsumenten* (*consumers*). Ein gut untersuchtes Beispiel hierfür sind Zwischenrepräsentationen im *Continuation-Passing Style* (CPS). Im Continuation-Passing Style ist die Continuation, d.h. der Rest der Berechnung, explizit als Term in der Sprache vorhanden. Sprachen im Continuation-Passing Style sind seit langem als Option für Zwischenrepräsentationen in Compilern etabliert. Eine andere, bisher deutlich weniger beachtete Möglichkeit sind Sprachen, die auf dem *Sequenzenkalkül* basieren. Der Sequenzenkalkül ist das Gegenstück zum natürlichen Schließen, bekannt aus der Logik. Während das natürliche Schließen durch seine Entsprechung mit dem Lambda-Kalkül via des Curry-Howard-Isomorphismus seit langem als Grundlage funktionaler Programmiersprachen dient, wurde erst vergleichsweise kürzlich ein zufriedenstellendes Termzuweisungssystem für den klassischen Sequenzenkalkül gefunden. Im Gegensatz zum natürlichen Schließen und dem Lambda-Kalkül, die auf Beweise, welche zu *Produzenten* (*producers*) korrespondieren, ausgerichtet sind, behandeln der klassische Sequenzenkalkül und darauf basierende Sprachen Beweise und Widerlegungen, welche zu Konsumenten korrespondieren, auf symmetrische Weise. Dies macht solche Sprachen zu einer interessanten Alternative zum Continuation-Passing Style für Zwischenrepräsentationen.

In dieser Dissertation untersuchen wir zwei verschiedene Möglichkeiten, wie explizite Konsumenten bei der Kompilierung funktionaler Programmiersprachen eingesetzt werden können. Zunächst betrachten wir Zwischenrepräsentationen im Continuation-Passing Style sowie Übersetzungen in diese. Eine naheliegende Frage in diesem Zusammenhang ist, ob es möglich ist, Programme aus dem Continuation-Passing Style wieder *zurück in den Direktstil* (*direct style*) zu übersetzen, in dem Programme üblicherweise geschrieben werden. Dies ist nicht nur theoretisch von Interesse, sondern hat auch praktische Relevanz. Zwar bringt der Continuation-Passing Style viele Vorteile für Zwischenrepräsentationen mit sich, jedoch auch einige Nachteile im Vergleich zum Direktstil. Insbesondere stellen viele Plattformen einen Aufrufstapel (*call stack*) bereit, der von Programmen im Continuation-Passing Style nicht genutzt wird. Eine geeignete Rückübersetzung in den Direktstil könnte es einem Compiler ermöglichen, die Vorteile beider Stile zu nutzen. Im ersten Teil dieser Arbeit stellen wir daher das Design einer Rückübersetzung

in den Direktstil vor, die viele Eigenschaften besitzt, die im Compiler-Kontext wichtig sind, und legen damit die Grundlage für eine praktische Anwendbarkeit einer solchen Übersetzung.

Im zweiten Teil dieser Arbeit gehen wir in die andere Richtung: Wir kompilieren bis hinunter zu Maschinencode. Dabei verwenden wir jedoch nicht den Continuation-Passing Style, sondern untersuchen Zwischenrepräsentationen auf Basis des klassischen Sequenzenkalküls. Wir präsentieren eine vollständige Kompilationskette, beginnend mit einer funktionalen Ausgangssprache mit interessanten Eigenschaften, bis hin zur Erzeugung von Maschinencode, der auf einem realen Computer ausgeführt werden kann. Wir identifizieren eine Normalform unserer sequenzenkalkülbasierten Zwischenrepräsentation, die sich für eine überraschend direkte Möglichkeit der Codegenerierung eignet. Die symmetrische Struktur von Sprachen, die auf dem klassischen Sequenzenkalkül basieren, macht sie, auch im Hinblick auf Optimierungen, zu einem vielversprechenden Ziel, und unsere Kompilationskette bildet die Grundlage für zukünftige Untersuchungen.

Contents

List of Figures

1 Introduction

Compiling modern functional languages like Haskell, Scala, or OCaml to efficient machine code requires compiler authors to bridge the gap between high-level conveniences and low-level concerns. Key to making this task feasible is the design of suitable compiler intermediate representations (IRs). Every such IR lives in the force field between a high-level source language and a low-level target language: On the one hand, it should be close to the high-level language to facilitate **lowering into the IR**, and on the other hand, it should be close to low-level machine code to facilitate **code generation from the IR**. It should also be easy to **work within the IR**, in order to implement and reason about optimizations. Designing a single IR that simultaneously fulfills all the above needs is difficult: Lowering a source language into the IR and reasoning about semantics-preserving rewrites usually requires complex language constructs, such as higher-order functions, objects, or algebraic data types, that make invariants hold by construction. In contrast, code generation from the IR and accurate reasoning about execution cost are easier if the IR provides simple but unsafe low-level constructs for memory allocations, operations on registers, or jumps. An example of an intermediate language of the first kind is GHC's Core, which is based on the polymorphic lambda calculus, while an example of the second kind of IR is LLVM, which is a platform-independent assembly language.

Another important concern in the design of IRs is that modern programs are interactive and continually communicate with external hardware and software components. This communication must be asynchronous in order to utilize hardware effectively: The communicating part of the computation must be suspended, allowing other parts to run, and execution of the suspended part must be continued once the result of the communication is ready. Programming languages provide first-class support for this non-local control flow in the form of features like resumable exceptions, generators, async IO, and effect handlers. One possibility to compile such language features is to provide runtime support. Indeed, Haskell and OCaml recently gained runtime support for stack switching in the form of control operators and effect handlers, respectively[1]. Another option is to use an IR that can represent such non-local control flow directly. Attractive candidates in this regard, which at the same time strike a balance between the above design considerations, are IRs with *explicit consumers*.

But what is a consumer, anyway? To answer this question, first consider the kind of terms typical programming languages are made of, which are *producers*. A producer creates a result, generally from smaller parts. Examples found in many programming languages are arithmetic operations, which produce a new integer from given ones, constructors of algebraic data types, which can pack up several given parts, and function calls, which compute a result from given inputs. Producers are ubiquitous in typical programming languages.

A consumer uses a previously produced result to do something with it, generally without producing a further result. Consumers are not usually present in programming languages in the same way producers are. Yet, there is always at least one consumer at every point in the program, albeit an implicit one: the rest of the computation, i.e., the *continuation* or *program context*.

Continuations

For example, consider the following function, which multiplies a given list of integers.

[1] For Haskell, see the GHC proposal at https://github.com/ghc-proposals/ghc-proposals/pull/313, and for OCaml see Sivaramakrishnan et al. (2021).

```
def mult(l : List): Int {
  l.case {
    Nil() ⇒ 1,
    Cons(x, xs) ⇒ x * mult(xs)
  }
}
```

When we make the recursive call in the `Cons` branch, the rest of the computation consists of multiplying the recursive result with the head `x` of the list. But how can we make this consumer explicit, that is, how can we make it accessible to another part in the program? One way to do so is to translate the program from standard *direct style* (DS) into *continuation-passing style* (CPS). In CPS, every function receives an additional argument for passing the current continuation into it at every call site. Instead of returning the computed result, functions invoke the continuation with this result. The above example hence becomes (we use different styling for better discriminability)

```
def mult(l : List | k : ¬Int) {
  l.case {
    Nil() ⇒ k(1),
    Cons(x, xs) ⇒ mult(xs | r ⇒ k(x * r))
  }
}
```

mult has an additional parameter k for the continuation. In the recursive call, mult receives the continuation $r \Rightarrow k(x * r)$, which expects the recursive result r, multiplies it by the head x of the list, and then continues with the outer continuation k.

It has been demonstrated that continuation-passing style is well-suited for use in a compiler IR (Steele, 1978; Appel, 1992; Kennedy, 2007), in particular in the presence of complex non-local control flow. As continuations are explicit and named in CPS, they can be used in non-pure ways, for example, invoking the same continuation twice, or not invoking a continuation at all, thus facilitating non-local control flow without additional constructs. Explicit continuations also enable several program optimizations by inlining and reduction (Schuster et al., 2020) that would be much more difficult to achieve with implicit continuations in direct style. Continuation-passing style and translations into it are well-studied and exist in many different variants.

A naturally arising question is whether it is possible to translate programs from CPS back to direct style. Direct-style translations have been studied as well (Danvy, 1992; Danvy and Lawall, 1992; Sabry and Felleisen, 1993; Biernacki et al., 2020, 2021), albeit not as much as CPS translations. Usually, they are presented as the inverse of some CPS translation. Having well-behaved translations back and forth between DS and CPS is theoretically interesting, as it helps to understand the relationship between the two styles. But DS translations are also interesting from a more practical point of view in compilers using a CPS IR. CPS comes with a few downsides (Maurer et al., 2017; Cong et al., 2019) compared to direct style. In particular, many platforms have support for DS code since they come with a call stack where frames are allocated, whereas in CPS, stack frames are allocated as closures on the heap (Farvardin and Reppy, 2020). It can thus be beneficial for compilers to translate programs in CPS back to direct style after processing them.

Structured Consumers

With first-class continuations, CPS features explicit consumers. However, there still is an asymmetry between producers and consumers. While there is only one form of consumer, the continuations, producers typically come in many different forms with different structures, such as arithmetic operations, constructors of data types, or function abstractions. This begs the question of whether we can also exhibit more and differently structured consumers explicitly.

For example, a consumer for a constructor of an algebraic data type is a pattern match, and a consumer for an object is a method called on it. In general, elimination forms for producers are consumers for them. These are usually implicit in typical languages based on natural deduction, because to apply an elimination rule, we first have to produce a corresponding value. Thus, the consumer is built into a compound construct and is not itself a first-class entity.

However, when Gentzen (1935a) introduced natural deduction, which is biased towards proofs, he also defined a sibling calculus, the (classical) sequent calculus, in which proofs and refutations are completely symmetric. It is well-known (Howard, 1980) that, via the Curry-Howard isomorphism, proofs in natural deduction correspond to terms of languages based on the lambda calculus, i.e., to producers. A satisfying term assignment system for sequent calculus has only been given rather recently (Curien and Herbelin, 2000) with the $\lambda\mu\tilde{\mu}$-calculus. It is completely symmetric for producers and consumers, with producers corresponding to proofs and consumers corresponding to refutations. Producers and consumers interact when they meet in a *cut*, which represents computation, written $\langle p \mid c \rangle$, where p is a producer and c is a consumer. In a language based on sequent calculus, the above example of multiplication of a list of integers might look as follows.

$$
\begin{array}{l}
\textsf{def}\ \mathsf{mult}(l :^{\mathsf{prd}} \mathtt{List},\ \alpha :^{\mathsf{cns}} \mathsf{Int})\ \{ \\
\quad \langle l \mid \textsf{case}\ \{ \\
\qquad \mathtt{Nil}() \Rightarrow \langle 1 \mid \alpha \rangle, \\
\qquad \mathtt{Cons}(x,\ xs) \Rightarrow \langle x * \mu\beta.\, \mathsf{mult}(xs,\ \beta) \mid \alpha \rangle \\
\quad \}\rangle \\
\}
\end{array}
$$

In some respects, this is quite similar to the CPS version. In particular, the function mult abstracts a consumer parameter with a *covariable* α, which is supposed to consume the result the function computes, similar to the continuation in CPS. α is thus used in both branches of the pattern match in the outer cuts. The above program also differs from the CPS version in some regards, however. First, the recursive call of mult is not lifted out of the multiplication. Rather, it locally abstracts over a consumer β with a μ-abstraction, where β stands for the program context that contains the multiplication by the head x of the list and then continues with the outer consumer α. This is possible because, unlike a lambda abstraction, the μ-abstraction is generally not a value and hence does not hide the computation in its body from evaluation. The sequent calculus version thus is structurally closer to the direct style version. Second, the pattern match is separated from the scrutinee l upon which it acts. They now meet in a cut, with l being the producer and the pattern match being the consumer. That is, the pattern match is a first-class construct on its own, which can also be bound to a covariable and passed as an argument. More generally, languages based on classical sequent calculus usually exhibit structured consumers in the same way as structured producers.

These properties make intermediate representations based on sequent calculus an interesting alternative to CPS for the compilation of functional programming languages.

1.1 Overview and Contributions

In this thesis, we investigate two different, yet related, ways how explicit consumers can be used in the compilation of functional programming languages. Accordingly, the thesis consists of two parts. An overview is given in Figure 1.1. It shows the numerous calculi that we will present in the course of this thesis, and how they are related by translations and transformations between them. We use dashed arrows for the connections that we discuss in a less precise and formal way, or even leave implicit. We also use different kinds of arrows to indicate properties of the translation or transformation they represent. We use $\xrightarrow{\sim}$ for isomorphisms (bijective), $\hookrightarrow$ for monomorphisms (injective), and $\twoheadrightarrow$ for epimorphisms (surjective).

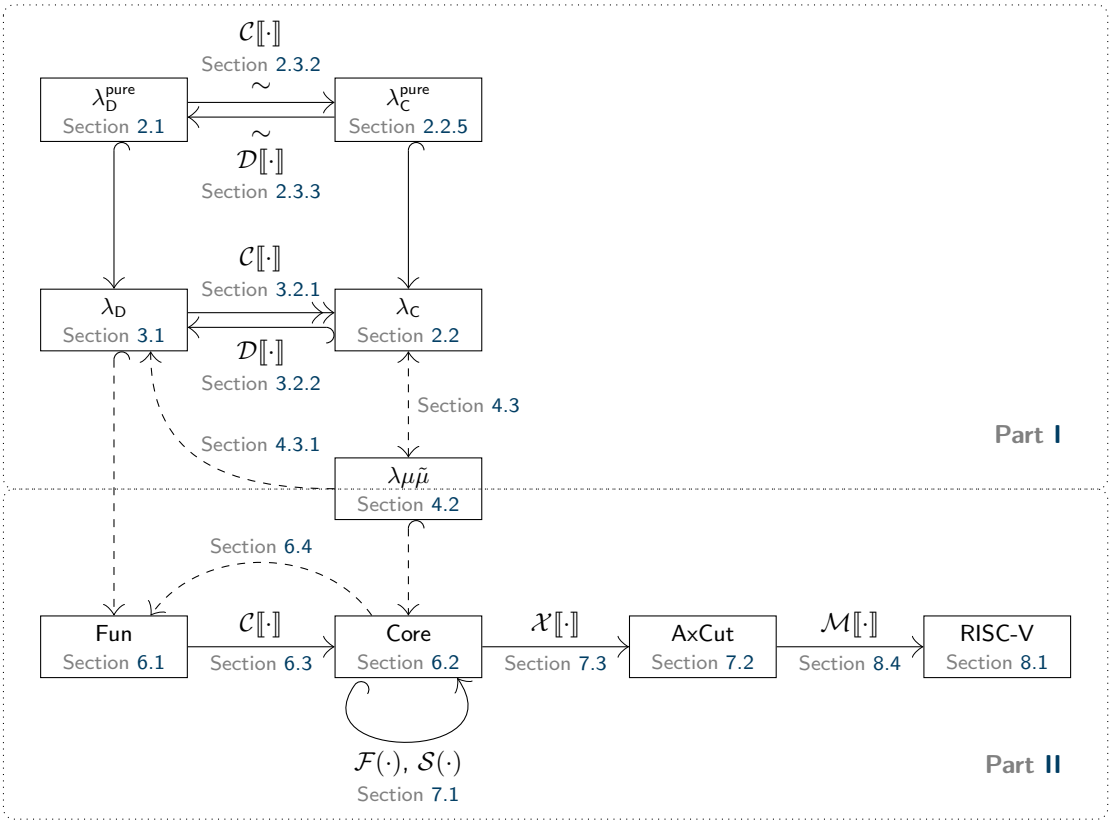


Figure 1.1: Overview for the calculi and relations between them.

The upper half of the diagram concerns the first part of the thesis, where we consider continuation-passing style and translations back to direct style. The usefulness of continuation-passing style as a compiler intermediate language has been debated controversially in the literature for quite some time (Appel, 1992; Kennedy, 2007; Flanagan et al., 1993; Maurer et al., 2017; Cong et al., 2019). CPS is particularly well-suited to compile and optimize programs that make use of non-local control effects. However, programs in direct style may benefit from using the call stack provided by many platforms, and are typically easier to understand. With the ability to translate programs back to direct style, a compiler may reap the benefits of both CPS and DS. To be practically usable, such a translation should satisfy a number of properties that are of less concern for more theoretical considerations (Sabry and Felleisen, 1993; Sabry and Wadler, 1997; Hatcliff and Danvy, 1994). For example, it should support complex non-local control flow, but it should not generate unnecessary usages of control operators in direct style. In Part I of this thesis, we examine the design of a translation back to direct style satisfying many of these properties. It is supposed to lay the groundwork for a practically usable translation in a compiler.

While CPS has long been established as an intermediate representation for compilation down to machine code, intermediate representations based on sequent calculus have received only little attention so far (Downen et al., 2016). Such languages share many similarities with CPS, but the symmetric treatment of producers and consumers makes languages based on classical sequent calculus a promising alternative to languages based on CPS for compiler IRs that is worth investigating further. In Part II of this thesis, illustrated in the lower half of the above diagram, we will examine how intermediate representations based on classical sequent calculus can be employed in the compilation of functional programming languages. We will present a complete compilation pipeline, starting from an interesting practical surface language and ending with executable machine code. The basic design forms a foundation for future investigations, in particular with regard to optimizations enabled by the structure of sequent calulus.

Before diving into the details, we give an overview of the two parts, walking through the diagram in Figure 1.1, and summarizing the contributions.

1.1.1 Back to Direct Style

In the first part of this thesis, we examine translations from continuation-passing style back to direct style. In Chapters 2 and 3 we present two calculi, one in direct style and one in continuation-passing style. Both calculi allow for complex non-local control flow, but we also identify their pure fragments, which only allow for trivial control flow. We further present a translation from the DS language to the CPS language and a translation back to direct style. We first consider the translations for the pure languages. This allows us to explain the main ideas in a simple setting, before extending the translations to the more complex case of the full languages. The calculi and translations between them are designed in such a way that they satisfy several meta-theoretical properties. As both calculi are typed, they also allow for a logical interpretation. In Chapter 4 we discuss how our calculi correspond to classical logic in natural deduction. We further examine how close our CPS calculus is to classical sequent calculus by comparing it to the $\lambda\mu\tilde{\mu}$-calculus.

Related Publications The first part of this thesis is mostly based on the following publication:

> Marius Müller, Philipp Schuster, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. 2023. Back to Direct Style: Typed and Tight. *Proc. ACM Program. Lang.* 7, OOPSLA1. DOI: https://doi.org/10.1145/3586056

Parts of Chapter 4 are further based on the following publication:

> David Binder, Marco Tzschentke, Marius Müller, and Klaus Ostermann. 2024. Grokking the Sequent Calculus (Functional Pearl). *Proc. ACM Program. Lang.* 8, ICFP. DOI: https://doi.org/10.1145/3674639

Contributions In Chapter 2, we present the following results:

- We introduce the pure fragment $\lambda_{\mathsf{D}}^{\mathsf{pure}}$ of our DS calculus (Section 2.1). We further present our CPS calculus λ_{C} (Section 2.2) and identify its pure fragment (Section 2.2.5).
- Both calculi are typed and are given an operational semantics in terms of an abstract machine. Both systems are type safe, manifested by the typical theorems of progress and preservation.
- We present a CPS translation $\mathcal{C}[\![\cdot]\!]$ from $\lambda_{\mathsf{D}}^{\mathsf{pure}}$ to $\lambda_{\mathsf{C}}^{\mathsf{pure}}$ (Section 2.3.2) and a DS translation $\mathcal{D}[\![\cdot]\!]$ from $\lambda_{\mathsf{C}}^{\mathsf{pure}}$ to $\lambda_{\mathsf{D}}^{\mathsf{pure}}$ (Section 2.3.3). Both translations are also defined on machine states.
- We show a number of meta-theoretical properties for the translations (Section 2.3.4), such as well-typedness preservation and step-by-step semantics preservation. Moreover, we show that the translations are syntactic inverses of each other.

While similar results have been described in the literature before (Danvy, 1992; Flanagan et al., 1993), the simple setting of the pure languages allows us to explain our basic ideas without getting lost in the complexity added by non-trivial control flow.

In Chapter 3 we present the following results, which form the main contributions of the first part:

- We extend the pure DS calculus by augmenting it with a set of control operators, resulting in a calculus λ_{D} that allows for non-trivial control flow (Section 3.1). λ_{D} is still typed and its abstract machine semantics is type safe.
- We extend the CPS translation (Section 3.2.1) and the DS translation (Section 3.2.2) to the full languages and abstract machines.
- We show that the translations still satisfy several meta-theoretical properties (Section 3.2.3), although some are weaker than in the pure case. In particular, semantics is not preserved step-by-step anymore, but we can still give a tight global upper bound for the number of steps taken.
- We discuss how the languages can be extended with conditional statements to make them more practically usable (Section 3.3).

In Chapter 4 we present the following results:

- We discuss how our DS calculus corresponds to logical calculi in natural deduction (Section 4.1). In particular, we show how our control operators enable classical reasoning, as opposed to the pure calculus, which corresponds to intuitionistic logic.
- We discuss how our CPS calculus corresponds to a double negation embedding of classical logic into intuitionistic logic and how this makes it, in some respects, appear to be closer to sequent calculus than to natural deduction (Section 4.3). To this end, we further examine how the $\lambda\mu\tilde{\mu}$-calculus (Section 4.2) compares to our CPS calculus.
- We sketch how a translation back to direct style might proceed for the $\lambda\mu\tilde{\mu}$-calculus (Section 4.3.1).

The discussion in this chapter is less formal and precise, and the connections to logic are well-known. Still, these logical considerations allow us to introduce the $\lambda\mu\tilde{\mu}$-calculus, which forms the basis for the intermediate languages in Part II, and to make a connection and draw a comparison between the calculi in Part I and those in Part II.

1.1.2 Compiling with the Sequent Calculus

In the second part of this thesis, we examine how intermediate representations based on classical sequent calculus can be employed in the compilation of functional programming languages. We present a complete compilation pipeline, starting from a functional surface language with interesting features, and show how to compile it down to machine code that can be executed on a real computer. To do so, we use sequent-calculus-inspired languages throughout all the intermediate stages. We give an overview of the compilation pipeline in Chapter 5. In Chapter 6 we extend our basic DS calculus λ_{D} from Part I to a practically usable surface language Fun. In particular, we add general algebraic data types and pattern matching, as well as algebraic codata types (Hagino, 1989) and copattern matching (Abel et al., 2013). While data types are a common feature in most modern (functional) programming languages, codata types are less well-known. Codata types are similar to interfaces in object-oriented programming in that they are defined by destructors similar to methods. Copattern matching is then used to define an object of a codata type by implementing what happens on a destructor invocation on the object. The duality between data and codata types is most clearly visible in a symmetric language based on sequent calculus, which makes such a language an ideal candidate for an intermediate representation in this regard. Our high-level intermediate representation Core is based on the $\lambda\mu\tilde{\mu}$-calculus, extended in a similar way how Fun is extended from λ_{D}.

As we eventually want to generate machine code, we derive a normal form for the sequent calculus terms of Core that is suitable for code generation in Chapter 7. We first apply two transformations on Core to name all terms and to shrink the language. Then we translate the transformed version into a language AxCut with a slightly different structure. In AxCut the structural rules of exchange, weakening, and contraction are made explicit by introducing a language construct for them. In Chapter 8 we show how to generate code from AxCut programs in a surprisingly direct way. We have implemented the complete compilation pipeline and conducted benchmarks on a considerable number of programs, which we discuss in Chapter 9.

Related Publications The second part of this thesis is closely based on the following journal article, currently under consideration for publication:

> Marius Müller, David Binder, Marco Tzschentke, Philipp Schuster, Klaus Ostermann, and Jonathan Immanuel Brachthäuser. 2026. Compiling with the Sequent Calculus. Submitted to *ACM Trans. Program. Lang. Syst.*

The journal article itself is based on the following publications:

> Philipp Schuster, Marius Müller, Klaus Ostermann, and Jonathan Immanuel Brachthäuser. 2025. Compiling Classical Sequent Calculus to Stock Hardware: The Duality of Compilation. *Proc. ACM Program. Lang.* 9, OOPSLA1. DOI: https://doi.org/10.1145/3720507
>
> David Binder, Marco Tzschentke, Marius Müller, and Klaus Ostermann. 2024. Grokking the Sequent Calculus (Functional Pearl). *Proc. ACM Program. Lang.* 8, ICFP. DOI: https://doi.org/10.1145/3674639

Contributions In Chapter 6 we present the following results:

– We present a typed functional surface language Fun with general data and codata types (Section 6.1). Fun mixes different evaluation strategies, with data types being evaluated with call-by-value order and codata types being evaluated with call-by-name order. Moreover, Fun supports complex non-local control flow through the use of control operators for undelimited continuations.

- We present a typed high-level intermediate language Core based on classical sequent calculus (Section 6.2). Core also features general data and codata types. Due to the symmetry between producers and consumers, it naturally allows for complex non-local control flow.
- We present an efficient translation $\mathcal{C}[\![\cdot]\!]$ from Fun to Core (Section 6.3). It borrows ideas from one-pass, first-order, and compositional continuation-passing style translations to avoid the generation of administrative redexes. The translation preserves well-typedness.
- We discuss what problems the additional features, not present in the calculi in Part I, pose for translating Core back to direct style and sketch potential solutions (Section 6.4).

In Chapter 7 we present the following results:

- We present two transformations on Core, each targeting a smaller fragment of the whole language (Section 7.1). The first transformation $\mathcal{F}(\cdot)$ is an efficient focusing transformation, which names all terms in argument position (Section 7.1.1). The second transformation $\mathcal{S}(\cdot)$ shrinks the language by removing several redundant constructs (Section 7.1.2). Both transformations are type-preserving.
- We present a typed lower-level intermediate language AxCut (Section 7.2), which structurally differs slightly from the shrunk fragment of Core. In particular, it features a construct for explicit substitutions that encompasses the structural rules of exchange, weakening, and contraction, which are explicit in AxCut, in contrast to the higher-level calculi. AxCut comes equipped with a type-safe abstract machine semantics.
- We present a type-preserving translation $\mathcal{X}[\![\cdot]\!]$ from the shrunk fragment of Core into AxCut (Section 7.3). It infers explicit substitutions, with the goal of dropping objects not needed anymore as early as possible and duplicating objects as late as possible.

In Chapter 8 we present the following results:

- We present a translation $\mathcal{M}[\![\cdot]\!]$ (Section 8.4) that directly generates RISC-V assembly code (Section 8.1) from AxCut.
- We show how the design of AxCut facilitates a real-time garbage collection technique with a fast path for linear usage, based on lazy reference counting.

In Chapter 9 we present the following results:

- We have implemented the complete compilation pipeline (Section 9.1). Besides targeting RISC-V assembly code, it allows us to generate executable binaries for x86-64 and AArch64.
- We evaluate the performance of our approach by benchmarking our implementation against several other compilers for various languages on a considerable benchmark suite we have implemented (Section 9.2). The results demonstrate that the approach of our compilation pipeline is viable.

Part I

Back to Direct Style

Abstract. Translating programs into continuation-passing style is a well-studied tool to explicitly deal with the control structure of programs. This is useful, for example, for compilation. A naturally arising question is whether there is an inverse translation back to direct style. The answer to this question depends on how the continuation-passing translation is defined and on the domain of the inverse translation.

In Chapter 2 we discuss the design of a typed language in continuation-passing style and study a translation from a simply typed lambda calculus in direct style into this CPS language and a translation back to direct style. To properly define the translation back to direct style, we have to restrict the CPS language to its pure fragment, i.e., to trivial control flow. Both translations are type-preserving. Moreover, both languages are equipped with an abstract machine semantics, which is preserved by the translations step by step, giving an operational correspondence between the two languages. In fact, the two translations are full syntactic inverses with the CPS language restricted to trivial control flow.

In general, translating programs from continuation-passing style back to direct style requires the use of control operators to account for the use of continuations in non-trivial ways. In Chapter 3 we therefore extend the direct-style language with a set of control operators that allow for non-trivial control flow. We then extend the translations into continuation-passing style and back to direct style to the full languages, including the abstract machines. The extended translations are still type-preserving and still preserve the abstract machine semantics in a very precise way. Moreover, we show that the compositions of the translations are still well-behaved. In particular, they are syntactic one-sided inverses for the full languages.

In our typed setting, there is also a logical interpretation of our calculi, and the translation into continuation-passing style can be viewed as an embedding of classical logic into intuitionistic logic. In Chapter 4 we discuss these logical aspects. Both the DS calculus and the CPS calculus are based on natural deduction. However, the CPS calculus bears some similarities to term-assignment systems for the sibling calculus of natural deduction, the sequent calculus. The sequent calculus is a proof system that was designed as a more symmetric alternative to natural deduction. A term assignment system for the classical sequent calculus is the $\lambda\mu\tilde{\mu}$-calculus. We compare our CPS calculus to the $\lambda\mu\tilde{\mu}$-calculus and briefly discuss what a translation from the $\lambda\mu\tilde{\mu}$-calculus back to our DS calculus might look like.

This part is mainly based on the following publication (Müller et al., 2023):

Marius Müller, Philipp Schuster, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. 2023. Back to Direct Style: Typed and Tight. *Proc. ACM Program. Lang.* 7, OOPSLA1. DOI: https://doi.org/10.1145/3586056

Parts of Chapter 4 are additionally based on the following publication (Binder et al., 2024):

David Binder, Marco Tzschentke, Marius Müller, and Klaus Ostermann. 2024. Grokking the Sequent Calculus (Functional Pearl). *Proc. ACM Program. Lang.* 8, ICFP. DOI: https://doi.org/10.1145/3674639

2 Continuation-Passing Style and Direct Style

Continuation-passing style (CPS) is a representation of programs where control flow is explicit and named, which is why it is useful as a compiler intermediate representation (Appel, 1992; Kennedy, 2007). It enables several program optimizations by inlining and reduction. Moreover, it is useful as an implementation technique for control operators, because the current continuation is always immediately available. Continuation-passing style translations translate a program to CPS. They are well-studied and exist in many different variants. In typed languages, they are defined as translations on types and terms. When interpreted logically, they correspond to double-negation translations. However, CPS also has disadvantages (Maurer et al., 2017; Cong et al., 2019), as programs are less readable, stack frames are allocated as closures on the heap (Farvardin and Reppy, 2020), and control flow is potentially arbitrary, destroying some of the guarantees of structured programming.

Direct style (DS) does not have these disadvantages. It is the preferred way in which to write programs. Many platforms have support for DS code since they come with a call stack where frames are allocated. In a typed language without control operators, programs directly correspond to proofs in intuitionistic logic (Howard, 1980). In a typed language with control operators, programs correspond to proofs in classical logic (Griffin, 1989). Direct-style translations translate a program from CPS back to DS. DS translations have been studied as well (Danvy, 1992; Danvy and Lawall, 1992; Sabry and Felleisen, 1993; Biernacki et al., 2020, 2021), albeit not as much as CPS translations. Usually, they are presented as the inverse of some CPS translation. In a typed setting, DS translations can be seen as the inverse of a double-negation translation.

Having well-behaved translations back and forth between DS and CPS is theoretically interesting, as it helps to understand the relationship between the two styles. But there is also a more practical aspect. Programmers want to write programs in DS, as they are easier to read. However, some optimizations of programs are easier to accomplish in CPS, in particular, advanced control-flow optimizations often amount to simple beta-reduction (Schuster et al., 2020). Thus, it can be beneficial to translate programs to CPS for compilation, especially for languages with control operators. On the other hand, programs in DS can often be executed more efficiently, as frames are allocated on the stack and not on the heap. So, ideally, we would also like to have the possibility of translating optimized programs back to DS to then run them. To obtain theoretically satisfying results and also lay the groundwork for practical usability, the DS language, the CPS language, the DS translation, the CPS translation, and their combination should satisfy a number of meta-theoretical properties.

In this chapter, we restrict our focus to pure programs, i.e., programs with trivial control flow. This allows us to explain the basic design of our calculi and of the translations between them in a simple setting, providing the basis for the treatment of non-trivial control flow in Chapter 3. In Chapter 4 we will discuss some of the logical aspects.

We present two languages in this chapter, $\lambda_{\mathsf{D}}^{\mathsf{pure}}$ in direct style in Section 2.1, and λ_{C} in continuation-passing style together with its pure fragment $\lambda_{\mathsf{C}}^{\mathsf{pure}}$ in Section 2.2.

– Both have a type system, which is useful for compiler intermediate representations, and which means they admit a logical interpretation.

- Both have an abstract machine with a small-step operational semantics. This allows us to prove a stepwise operational correspondence.
- Both have the properties of progress and preservation (Theorems 1 and 2 for $\lambda_{\mathsf{D}}^{\mathsf{pure}}$, Theorems 4 and 5, and 7 and 8 for λ_{C} and $\lambda_{\mathsf{C}}^{\mathsf{pure}}$, respectively). Well-typed programs do not get stuck.

We present two translations between $\lambda_{\mathsf{D}}^{\mathsf{pure}}$ and $\lambda_{\mathsf{C}}^{\mathsf{pure}}$, a CPS translation and a DS translation, in Section 2.3. We will extend them to the full non-pure languages in Chapter 3.

- Both translations are compositional, first-order, and one-pass. These are desirable properties of CPS translations and we argue of DS translations as well.
- Both preserve well-typedness (Theorem 10). This is important if we want to interpret them as a double-negation translation and its inverse, respectively.
- Both preserve semantics (Theorems 11 and 14). Indeed, we prove a step-by-step correspondence.
- Neither duplicates code. This is important if we want to actually use them in a compiler, where code size matters.

Moreover, when taken together, these two translations have the following desirable properties.

- The translations are full syntactic inverses. When we translate a term forth and back, we arrive at the same term (Theorems 17 and 18).
- We extend both translations to machine states, which allows us to freely translate back and forth between arbitrary intermediate states.

2.1 The Pure DS Language $\lambda_{\mathsf{D}}^{\mathsf{pure}}$

We first present our language $\lambda_{\mathsf{D}}^{\mathsf{pure}}$ in direct style. After briefly motivating the design, we present the syntax and typing rules and give an operational semantics in terms of an abstract machine.

2.1.1 Naming Intermediate Results

In CPS, the control flow of a program is made explicit by reifying the remaining computation as a continuation at each computation step. This means that the evaluation order of programs is syntactically explicit. For example, assuming call-by-value evaluation order, the CPS translation of a nested function call might look as follows.

$$\mathcal{C}[\![f(g(3))]\!]_k = g(3, \lambda x \Rightarrow f(x, k))$$

Here k is the outer continuation. The evaluation order is made explicit by first calling the inner function g and moving the call of the outer function f into the continuation $\lambda x \Rightarrow f(x, k)$ passed to g. It would not be sound to translate the nested function call as follows.

$$\mathcal{C}[\![f(g(3))]\!]_k \neq f(\lambda k_0 \Rightarrow g(3, k_0), k)$$

Here the result of the translation would not evaluate the call of the inner function g first, as it is hidden inside a lambda abstraction that abstracts over its continuation. The translation to CPS hence forgets about the original nesting structure of the translated term. We can also write an equivalent term in the DS language that is not nested and for which we arrive at the

Syntax of $\lambda_{\mathsf{D}}^{\mathsf{pure}}$

Statements	s	::=	$\mathbf{val}\, x = s;\, s$	sequence
		|	$\mathbf{ret}\, e$	return
		|	$\mathbf{def}\, f(x : \tau)\, \{\, s \,\};\, s$	define
		|	$f(e)$	call
Variables	v	::=	x, f	variables
Expressions	e	::=	v	variables
		|	$0 \mid 1 \mid \ldots$	integers

Syntax of Types

Types	τ	::=	$\tau \to \tau$	function type
		|	Int	base type
Environment Type	Γ	::=	$\Gamma,\, x : \tau$	extended environment
		|	$\emptyset$	empty environment

Figure 2.1: The pure direct-style language $\lambda_{\mathsf{D}}^{\mathsf{pure}}$.

same result when translating to CPS. Assuming that we have let-bindings, denoted by **val**, we get the following translation.

$$\mathcal{C}[\![\, \mathbf{val}\, x = g(3);\, f(x) \,]\!]_k = g(3,\, \lambda x \Rightarrow f(x,\, k))$$

Now, when translating the CPS term back to direct style, we do not know whether the term originally was nested or not, so we have to pick one DS term. Choosing the sequenced version makes the similarity between the DS version and the CPS version most apparent.

$$\mathcal{D}[\![\, g(3,\, \lambda x \Rightarrow f(x,\, k)) \,]\!] = \mathbf{val}\, x = g(3);\, f(x)$$

With this choice, a round trip starting in DS, i.e., translating a DS term to CPS and back, keeps the names of all intermediate results. To root out ambiguity when translating back and forth, we design our DS language $\lambda_{\mathsf{D}}^{\mathsf{pure}}$ in such a way that naming intermediate results is always required. This is just as in A-normal form and in fact, translating to CPS and back is essentially how ANF came about (Sabry and Felleisen, 1992; Flanagan et al., 1993). There is a second requirement in ANF, which is that sequencing must not be nested, but we do not demand this for $\lambda_{\mathsf{D}}^{\mathsf{pure}}$. Thus, our language is, in a sense, halfway in ANF and we also say that our calculus is *focused*. The notion of focusing stems from the realms of proof search in logic (Andreoli, 1992). It has been shown to correspond to the one half of the requirements of ANF demanding that intermediate results are named (Binder and Piecha, 2022).

2.1.2 Syntax

Figure 2.1 defines the syntax of our pure DS language $\lambda_{\mathsf{D}}^{\mathsf{pure}}$. We syntactically distinguish between statements and expressions, i.e., we use a form of fine-grain call-by-value (Levy et al., 2003). We will extend statements with control effects in Chapter 3, but expressions will stay pure.

Syntax of Expressions

The only expressions are variables and literals. In particular, this means that there are no anonymous functions, and each function must be given a name.

Statement Typing $\boxed{\Gamma \vdash s : \tau}$

$$\frac{\Gamma \vdash s_0 : \tau_0 \quad \Gamma, x_0 : \tau_0 \vdash s : \tau}{\Gamma \vdash \mathsf{val}\, x_0 = s_0;\, s : \tau}\ [\textsc{Sequence}] \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{ret}\, e : \tau}\ [\textsc{Return}]$$

$$\frac{\Gamma, x : \tau \vdash s_0 : \tau_0 \quad \Gamma, f : \tau \to \tau_0 \vdash s : \tau_r}{\Gamma \vdash \mathsf{def}\, f(x : \tau)\,\{\, s_0 \,\};\, s : \tau_r}\ [\textsc{Define}] \qquad \frac{f : \tau \to \tau_0 \in \Gamma \quad \Gamma \vdash e : \tau}{\Gamma \vdash f(e) : \tau_0}\ [\textsc{Call}]$$

Expression Typing $\boxed{\Gamma \vdash e : \tau}$

$$\frac{v : \tau \in \Gamma}{\Gamma \vdash v : \tau}\ [\textsc{Var}] \qquad \frac{}{\Gamma \vdash 19 : \mathsf{Int}}\ [\textsc{Int}]$$

Figure 2.2: Typing rules for $\lambda_{\mathsf{D}}^{\mathsf{pure}}$.

Syntax of Statements

For a cleaner presentation, sequencing of statements and returning expressions is made explicit with a keyword. Moreover, functions are always unary. Function application is always on a single expression argument, which means that all intermediate results need to be named. This greatly helps to make the correspondence between the DS language and the CPS language clear. As explained above, while $\lambda_{\mathsf{D}}^{\mathsf{pure}}$ shares this property with ANF, it is different from the latter in that we allow nesting of sequencing statements.

Syntax of Types

There is one base type Int and a standard function type. Typing environments are entirely standard.

Example

As an illustration consider the following simple example.

```
def f(y : Int) : Int {
  ret y + 1
};
val z = f(x);
ret z + 2
```

Here, we define a function f that increments and then returns its parameter. We then call the function f on a free variable x. Finally, we increment the result z by two and return it.

2.1.3 Typing Rules

The typing rules for $\lambda_{\mathsf{D}}^{\mathsf{pure}}$ are given in Figure 2.2. There are two typing judgments, one for expressions and one for statements. There is one rule for each kind of expression and one rule for each kind of statement. All of the rules are completely standard.

2.1.4 Operational Semantics

We define the operational semantics of $\lambda_{\mathsf{D}}^{\mathsf{pure}}$ in terms of an abstract machine. We could, in principle, also give a substitution-based semantics using evaluation contexts instead of using an abstract machine. It is well-known that these different kinds of operational semantics are equivalent (Biernacka and Danvy, 2007). However, our abstract machine formulation is particularly

Syntax of Machine States for $\lambda_{\mathsf{D}}^{\mathsf{pure}}$

Values	V	$::=$	$\{\,\mathsf{E},\,(x\,:\,\tau)\Rightarrow s\,\}$	closure
		$\mid$	$0 \mid 1 \mid \ldots$	integer
Environments	E	$::=$	$\mathsf{E},\,x\mapsto\mathsf{V}$	binding
		$\mid$	$\bullet$	empty
Stacks	K	$::=$	done	underflow
		$\mid$	$\{\,\mathsf{E},\,(x\,:\,\tau)\Rightarrow s\,\}\,::\,\mathsf{K}$	frame
Configurations	M	$::=$	$\langle\mathsf{E}\parallel s\parallel\mathsf{K}\rangle$	delimited execution

Figure 2.3: Operational semantics of $\lambda_{\mathsf{D}}^{\mathsf{pure}}$.

Configuration Typing $\boxed{\vdash \mathsf{M}}$

$$\frac{\mathsf{E}\vdash_{\mathrm{env}}\Gamma \qquad \Gamma\vdash s:\tau \qquad \tau\vdash_{\mathrm{stk}}\mathsf{K}}{\vdash\langle\mathsf{E}\parallel s\parallel\mathsf{K}\rangle}\ [\textsc{Delim}]$$

Figure 2.4: Typing rules for machine states of $\lambda_{\mathsf{D}}^{\mathsf{pure}}$.

helpful to investigate the relationship between direct style and continuation-passing style, as it explicitly models the stack. Moreover, using environments for function closures makes it easier to uphold the restriction that only variables and numbers can occur in argument position.

Machine States

Figure 2.3 defines the syntax of the machine. A machine state, or configuration, consists of a statement currently in focus, an environment, and a stack the statement returns to. Environments are mappings from variables to values, where values are either closures or integers. A stack is a list of frames whose last frame is an underflow frame that indicates the end of evaluation. The final states of the machine are thus of the form $\langle\mathsf{E}\parallel\mathbf{ret}\,e\parallel\mathsf{done}\rangle$ with e being the final result. We use done as a bottom frame to start evaluation of a closed program s in state $\langle\,\bullet\parallel s\parallel\mathsf{done}\rangle$. Typing of the machine is unsurprising. Figure 2.4 defines the rule for configurations, the full set of rules can be found in Müller et al. (2023). For a machine state to be well-typed, the free variables of the statement in focus must be covered by the environment. Moreover, the type of the statement must agree with the type the stack expects.

(push)	$\langle\mathsf{E}\parallel\mathbf{val}\,x_0\,=\,s_0;\,s\parallel\mathsf{K}\rangle$	$\to$	$\langle\mathsf{E}\parallel s_0\parallel\{\,\mathsf{E},\,(x_0\,:\,\tau_0)\Rightarrow s\,\}\,::\,\mathsf{K}\rangle$
(ret)	$\langle\mathsf{E},\,v\mapsto\mathsf{V}\parallel\mathbf{ret}\,v\parallel\{\,\mathsf{E}_0,\,(x\,:\,\tau)\Rightarrow s\,\}\,::\,\mathsf{K}\rangle$	$\to$	$\langle\mathsf{E}_0,\,x\mapsto\mathsf{V}\parallel s\parallel\mathsf{K}\rangle$
(reti)	$\langle\mathsf{E}\parallel\mathbf{ret}\,42\parallel\{\,\mathsf{E}_0,\,(x\,:\,\tau)\Rightarrow s\,\}\,::\,\mathsf{K}\rangle$	$\to$	$\langle\mathsf{E}_0,\,x\mapsto 42\parallel s\parallel\mathsf{K}\rangle$
(def)	$\langle\mathsf{E}\parallel\mathbf{def}\,f(x\,:\,\tau)\,\{\,s_0\,\};\,s\parallel\mathsf{K}\rangle$	$\to$	$\langle\mathsf{E},\,f\mapsto\{\,\mathsf{E},\,(x\,:\,\tau)\Rightarrow s_0\,\}\parallel s\parallel\mathsf{K}\rangle$
(call)	$\langle\mathsf{E},\,f\mapsto\{\,\mathsf{E}_0,\,(x\,:\,\tau)\Rightarrow s\,\},\,v\mapsto\mathsf{V}\parallel f(v)\parallel\mathsf{K}\rangle$	$\to$	$\langle\mathsf{E}_0,\,x\mapsto\mathsf{V}\parallel s\parallel\mathsf{K}\rangle$
(calli)	$\langle\mathsf{E},\,f\mapsto\{\,\mathsf{E}_0,\,(x\,:\,\tau)\Rightarrow s\,\}\parallel f(42)\parallel\mathsf{K}\rangle$	$\to$	$\langle\mathsf{E}_0,\,x\mapsto 42\parallel s\parallel\mathsf{K}\rangle$

Figure 2.5: Machine steps of $\lambda_{\mathsf{D}}^{\mathsf{pure}}$.

Machine Steps

The evaluation steps of the abstract machine are defined in Figure 2.5. The rules are again unsurprising. Rule *push* pushes a frame onto the stack, capturing the current environment, and then continues execution with the first statement in focus. In the rules *ret* and *reti* for returning to the stack, the environment of the frame is reinstated and a binding for the returned value is added. Execution then goes on with the statement of the frame and with the remaining stack. There must always be at least an underflow frame left for these rules to fire, otherwise evaluation is finished. Rule *def* for defining a function adds a closure to the environment and focuses on the remaining statement s. When calling a function f in rules *call* or *calli*, it is looked up in the environment, and the environment that was captured in f is reinstalled. The body of f comes into focus and an additional binding for the parameter is added to the environment.

Type Safety

$\lambda_{\mathsf{D}}^{\mathsf{pure}}$ satisfies the usual theorems of progress (Theorem 1) and preservation (Theorem 2). These theorems have been shown in Idris 2 by an intrinsically typed implementation and the totality of the step-function on machine states.

Theorem 1 (Progress (pure DS))**.**
For a machine state M in $\lambda_{\mathsf{D}}^{\mathsf{pure}}$, if $\vdash M$ then either M is of the form $\langle E \,\|\, \mathbf{ret}\, e \,\|\, \mathsf{done}\rangle$ or there exists a unique machine state M' such that $M \to M'$.

Theorem 2 (Preservation (pure DS))**.**
For a machine state M in $\lambda_{\mathsf{D}}^{\mathsf{pure}}$, if $\vdash M$ and $M \to M'$ then $\vdash M'$.

Moreover, for a closed program with type τ, the expression resulting from evaluating it has the same type.

Theorem 3 (Type preservation (pure DS))**.**
If $\vdash s : \tau$ and $\langle \bullet \,\|\, s \,\|\, \mathsf{done}\rangle \to^* \langle E \,\|\, \mathbf{ret}\, e \,\|\, \mathsf{done}\rangle$, then $\Gamma \vdash e : \tau$ where $E \vdash_{env} \Gamma$.

Proof. None of the machine steps can add a done, so the final done is the initial done. Thus, e has the type that done expects, i.e., τ. □

2.2 The CPS Language λ_{C}

We now present our language λ_{C} in continuation-passing style. Again, after briefly motivating the design, we present the syntax and typing rules and give an operational semantics in terms of an abstract machine.

2.2.1 Continuations are not Functions

One way to view CPS is as a restriction of the lambda calculus, where all calls are tail calls (Belanger et al., 2019; Paraskevopoulou and Grover, 2021). Under this point of view, continuations are functions that represent the rest of the computation, but never return anything. To give continuations a type, languages in CPS typically add a bottom type $\bot$ and model continuations with the following type signature:

$$\tau \to \bot$$

This implicitly marks continuations as *consumers* for their input, not producing a result. As functions do not return either in CPS, their return type also is $\bot$. The fact that they are intended to return a result is encoded by passing an additional continuation argument to functions, which can then be invoked to consume the result. In CPS, the type $\tau_1 \to \tau_2$ of a function hence becomes

$$(\tau_1, \tau_2 \to \bot) \to \bot$$

Syntax of λ_{C}

Terms	t	::=	$\mathbf{let}\, f(x : \tau \mid k : \neg\tau)\,\{\, t \,\};\, t$	function
		|	$f(e \mid e)$	application
		|	$\mathbf{cnt}\, k(x : \tau)\,\{\, t \,\};\, t$	continuation
		|	$k(e)$	jump
		|	$\mathbf{exit}\, e$	exit
Variables	v	::=	x, f, k	variables
Expressions	e	::=	v	variables
		|	$0 \mid 1 \mid \ldots$	integers

Syntax of Types

Types	τ	::=	$\tau \to \tau$	function type
		|	$\neg\tau$	continuation type
		|	Int	base type
Environment Type	Γ	::=	$\Gamma, x : \tau$	extended environment
		|	$\emptyset$	empty environment

Figure 2.6: The continuation-passing-style language λ_{C}.

Continuations differ from functions, however, in that they are not intended to return anything, but rather represent the rest of the computation. Moreover, functions and continuations are treated quite differently in practice during compilation, for example, when performing optimizations. Furthermore, for our translation back to direct style, it is crucial that we can easily distinguish functions from continuations. Therefore, we use a different language construct for continuations and give them a built-in type different from functions. We will use an explicit negation type $\neg\tau$ (Kennedy, 2007). This also allows us to keep function types as they are in direct style.

2.2.2 Syntax

Figure 2.6 defines the syntax of our CPS language λ_{C}. The syntax for expressions is identical to that of $\lambda_{\mathsf{D}}^{\mathsf{pure}}$. The only change in the syntax for types is that we add an explicit continuation type $\neg\tau$ for continuations expecting an argument of type τ. Note, however, that, while having the same type as in $\lambda_{\mathsf{D}}^{\mathsf{pure}}$, functions in λ_{C} never return. In fact, no term ever returns, instead, they will jump to the next continuation.

Syntax of Terms

Functions in this calculus have exactly one ordinary parameter, but they additionally have a separate continuation parameter, which is syntactically separated with a bar. This is also reflected in function application. The $\mathbf{cnt}\, k(x : \tau)\,\{\, t \,\};\, t$ term can be used to construct a continuation. Such continuations take exactly one parameter and are called with exactly one expression. Moreover, there is a construct $\mathbf{exit}\, e$ that terminates the program with an expression e.

Term Typing $\boxed{\Gamma \vdash t}$

$$\frac{\Gamma, x : \tau, k : \neg\tau_0 \vdash t_0 \quad \Gamma, f : \tau \to \tau_0 \vdash t}{\Gamma \vdash \mathsf{let}\, f(x : \tau \mid k : \neg\tau_0)\,\{\, t_0 \,\};\, t}\ [\textsc{Let}]$$

$$\frac{f : \tau \to \tau_0 \in \Gamma \quad \Gamma \vdash e : \tau \quad \Gamma \vdash c : \neg\tau_0}{\Gamma \vdash f(e \mid c)}\ [\textsc{App}] \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{exit}\, e}\ [\textsc{Ext}]$$

$$\frac{\Gamma, x : \tau \vdash t_0 \quad \Gamma, k : \neg\tau \vdash t}{\Gamma \vdash \mathsf{cnt}\, k(x : \tau)\,\{\, t_0 \,\};\, t}\ [\textsc{Cnt}] \qquad \frac{k : \neg\tau \in \Gamma \quad \Gamma \vdash e : \tau}{\Gamma \vdash k(e)}\ [\textsc{Jmp}]$$

Expression Typing $\boxed{\Gamma \vdash e : \tau}$

$$\frac{v : \tau \in \Gamma}{\Gamma \vdash v : \tau}\ [\textsc{Var}] \qquad \frac{}{\Gamma \vdash 19 : \mathsf{Int}}\ [\textsc{Int}]$$

Figure 2.7: Typing rules for λ_{C}.

Example

As an illustration, consider again the simple example from the end of Section 2.1, this time in λ_{C}.

```
let f(y : Int | k : ¬Int) {
  k(y + 1)
};
cnt k0(z) {
  done(z + 2)
};
f(x | k0)
```

The function f now receives a continuation parameter k. Instead of returning, it jumps to this continuation with the result. After defining f, we construct a continuation k_0, which represents the rest of the program after the call to f. It increments z by two and calls the free top-level continuation done. Finally, the call to f now receives this newly constructed continuation.

2.2.3 Typing

Figure 2.7 defines the typing rules for λ_{C}. The judgment and rules for expressions are again the same as in $\lambda_{\mathsf{D}}^{\mathsf{pure}}$. In the judgment for terms, we see that since terms do not return anything, they are only typed in an environment, but not against a type. The typing rules are unsurprising. In rule Let the return type of the function being defined is the type expected by its continuation parameter, as usual in CPS. Rule App shows that the continuation expression c of a function application must have continuation type. In the following, we will thus usually assume c to be a variable.

2.2.4 Semantics

The operational semantics of λ_{C} is again given by an abstract machine.

Machine States

The syntax of the machine is shown in Figure 2.8. In contrast to the DS machine, a machine state only consists of an environment and a term in focus, but no stack. Environments look the same as in direct style, but the values are different (except for integers). Closures now

Syntax of Machine States for λ_{C}

Values	V	$::=$	$\{\, \mathsf{E},\, (x\, :\, \tau \mid k\, :\, \neg\, \tau) \Rightarrow t\,\}$	function closure
		$\mid$	$\{\, \mathsf{E},\, (x\, :\, \tau) \Rightarrow t\,\}$	continuation closure
		$\mid$	$0 \mid 1 \mid \ldots$	integer
Environments	E	$::=$	$\mathsf{E},\, x \mapsto \mathsf{V}$	binding
		$\mid$	$\bullet$	empty
Configurations	M	$::=$	$\langle\, \mathsf{E} \parallel t\,\rangle$	execution

Figure 2.8: Operational semantics of λ_{C}.

(let)	$\langle\, \mathsf{E} \parallel \mathbf{let}\, f(x\, :\, \tau \mid k\, :\, \neg\, \tau_0)\, \{\, t_0\,\};\, t\,\rangle$ $\langle\, \mathsf{E},\, f \mapsto \{\, \mathsf{E},\, (x\, :\, \tau \mid k\, :\, \neg\, \tau_0) \Rightarrow t_0\,\} \parallel t\,\rangle$	$\rightarrow$
(app)	$\langle\, \mathsf{E},\, f \mapsto \{\, \mathsf{E}_0,\, (x\, :\, \tau \mid k\, :\, \neg\, \tau_0) \Rightarrow t\,\},\, v \mapsto \mathsf{V},\, c \mapsto W \parallel f(v \mid c)\,\rangle$ $\langle\, \mathsf{E}_0,\, x \mapsto \mathsf{V},\, k \mapsto W \parallel t\,\rangle$	$\rightarrow$
(appi)	$\langle\, \mathsf{E},\, f \mapsto \{\, \mathsf{E}_0,\, (x\, :\, \tau \mid k\, :\, \neg\, \tau_0) \Rightarrow t\,\},\, c \mapsto W \parallel f(42 \mid c)\,\rangle$ $\langle\, \mathsf{E}_0,\, x \mapsto 42,\, k \mapsto W \parallel t\,\rangle$	$\rightarrow$
(cnt)	$\langle\, \mathsf{E} \parallel \mathbf{cnt}\, k(x\, :\, \tau)\, \{\, t_0\,\};\, t\,\rangle$ $\langle\, \mathsf{E},\, k \mapsto \{\, \mathsf{E},\, (x\, :\, \tau) \Rightarrow t_0\,\} \parallel t\,\rangle$	$\rightarrow$
(jmp)	$\langle\, \mathsf{E},\, k \mapsto \{\, \mathsf{E}_0,\, (x\, :\, \tau) \Rightarrow t\,\},\, v \mapsto \mathsf{V} \parallel k(v)\,\rangle$ $\langle\, \mathsf{E}_0,\, x \mapsto \mathsf{V} \parallel t\,\rangle$	$\rightarrow$
(jmpi)	$\langle\, \mathsf{E},\, k \mapsto \{\, \mathsf{E}_0,\, (x\, :\, \tau) \Rightarrow t\,\} \parallel k(42)\,\rangle$ $\langle\, \mathsf{E}_0,\, x \mapsto 42 \parallel t\,\rangle$	$\rightarrow$

Figure 2.9: Machine steps of λ_{C}

come in two flavors: function closures with a continuation parameter and continuation closures without one. This is also visible in the typing of values (the rules can be found in Müller et al. (2023)). While closures with a continuation parameter are of function type, those without are of continuation type. Having an explicit **exit** term in the language, the final states of the machine are now $\langle\, \mathsf{E} \parallel \mathbf{exit}\, e\,\rangle$ with e being the final result. We define done to be $\{\, \bullet,\, (x\, :\, \tau) \Rightarrow \mathbf{exit}\, x\,\}$ and define a closed program t to have one free variable k and start evaluation in state $\langle\, k \mapsto \mathsf{done} \parallel t\,\rangle$.

Machine Steps

The evaluation steps of the abstract machine are shown in Figure 2.9. Rule *let* for defining a function adds a closure with a continuation parameter to the environment and focuses on the remaining term. Calling a function f in rules *app* and *appi* essentially proceeds in the same way as in $\lambda_{\mathsf{D}}^{\mathsf{pure}}$, but instead of having a stack, there must be a binding for the continuation argument in the environment. The rules *cnt* for defining and *jmp* and *jmpi* for calling a continuation are similar, but a continuation closure is added to the environment and the call needs no continuation argument.

Type Safety

λ_{C} satisfies the usual theorems of progress (Theorem 4) and preservation (Theorem 5). These theorems have been shown in Idris 2 by an intrinsically typed implementation and the totality of the step-function on machine states.

Theorem 4 (Progress (CPS))**.**
For a machine state M in λ_{C}, if $\vdash M$ then either M is of the form $\langle E \parallel \textbf{\textit{exit}}\, e \rangle$ or there exists a unique machine state M' such that $M \to M'$.

Theorem 5 (Preservation (CPS))**.**
For a machine state M in λ_{C}, if $\vdash M$ and $M \to M'$ then $\vdash M'$.

Moreover, if a closed program with type τ does not contain **exit**, then the resulting expression has the same type. Note that if a program does contain **exit**, it may end with an expression having a different type.

Theorem 6 (Type preservation (CPS))**.**
*If $k : \neg\tau \vdash t$ and t does not contain **exit** and $\langle k \mapsto \mathsf{done} \parallel t \rangle \to^* \langle E \parallel \textbf{\textit{exit}}\, e \rangle$,*
then $\Gamma \vdash e : \tau$ where $E \vdash_{env} \Gamma$.

Proof. None of the machine steps can add an **exit**, so the final **exit** comes from calling the initial done. Thus, e has the type that done expects, i.e., τ. □

2.2.5 Pure Fragment

The CPS calculus λ_{C} allows for first-class usage of continuations, i.e., it supports complex non-local control flow. For example, a continuation can be duplicated and called multiple times, or dropped and not called at all, as in the following function.

```
let tryReadingFile(path : Path | cont : ¬String) {
  cnt onSuccess(_) { cont("success") };
  cnt onFailure(_) { cont("failure") };
  #readFile(path, onSuccess, onFailure)
}
```

This function uses a primitive asynchronous operation *#readFile* to read a file. The primitive operation expects two continuations, one for success and one for failure, and invokes one of them when ready. They are both defined in terms of the continuation passed to the encompassing function *tryReadingFile*. The DS calculus $\lambda_{\mathsf{D}}^{\mathsf{pure}}$ does not support such non-trivial control flow, it is pure. Therefore, to be able to translate from λ_{C} back to $\lambda_{\mathsf{D}}^{\mathsf{pure}}$, we have to restrict the CPS calculus to its pure fragment. In Chapter 3 we will extend the DS calculus to support non-trivial control flow and hence will be able to translate back from the full CPS language, including the above example.

To rule out non-trivial control flow in λ_{C} and hence restrict it to its pure fragment, we have to make sure that there is *exactly one* free continuation variable representing the *current continuation* in every term. This guarantees linear usage of the current continuation and rules out the existence of any other continuation. To do so, we first remove the **exit** term from the language. This means that terms must end in a call of a continuation or of a function that gets passed a continuation, so there is always at least one free continuation variable. Moreover, we remove the continuation type $\neg\tau$ from the types, so that it can only appear in the typing of continuation bindings. This ensures that continuations cannot be passed as arguments to other continuations, nor as the non-continuation argument of functions. In other words, there can never be more than one continuation argument in such calls. Finally, we adapt the typing contexts and typing rules a bit. We require the typing context to always contain exactly one binding with continuation type, say the right-most one, and we remove the old binding when a new one comes into scope. We could alternatively use a separate singleton continuation context for this. This ensures that only the current continuation can be called or passed to a function. We adapt those typing rules in which continuations are bound, that is, Let and Cnt, as follows.

$$\frac{\Gamma,\, x\,:\,\tau,\, k\,:\,\neg\,\tau_0 \vdash t_0 \qquad \Gamma,\, f\,:\,\tau \to \tau_0,\, k_c : \neg\,\tau_c \vdash t}{\Gamma,\, k_c : \neg\,\tau_c \vdash \mathbf{let}\, f(x\,:\,\tau \mid k\,:\,\neg\,\tau_0)\,\{\,t_0\,\};\, t}\ [\textsc{Let}]$$

$$\frac{\Gamma,\, x\,:\,\tau,\, k_c : \neg\,\tau_c \vdash t_0 \qquad \Gamma,\, k\,:\,\neg\,\tau \vdash t}{\Gamma,\, k_c : \neg\,\tau_c \vdash \mathbf{cnt}\, k(x\,:\,\tau)\,\{\,t_0\,\};\, t}\ [\textsc{Cnt}]$$

Whenever a new continuation variable is bound in a term, this new variable now represents the current continuation and the old variable is removed. Note that with our restriction of the context, all variables in Γ in these rules must be of non-continuation type.

We also have to adapt the abstract machine to reflect these changes. As there is no **exit** term anymore, we introduce a separate done value, which is bound to the one free variable of a closed program in a start configuration, instead of defining done to be a continuation closure whose body is **exit**. The final states now are of the form $\langle \mathsf{E},\, k \mapsto \mathsf{done} \parallel k(e) \rangle$. These changes make the treatment more similar to what we have seen for the pure DS language $\lambda_{\mathsf{D}}^{\mathsf{pure}}$. We could alternatively have used **exit** only as runtime term and still defined done in terms of it. Similarly we could have introduced an **exit** statement for the runtime in $\lambda_{\mathsf{D}}^{\mathsf{pure}}$ and also defined done in terms of it there. For simplicity, we stick with the primitive done here. We also adapt the machine steps where continuations are bound.

(let)	$\langle \mathsf{E},\, c \mapsto \mathsf{V} \parallel \mathbf{let}\, f(x\,:\,\tau \mid k\,:\,\neg\,\tau_0)\,\{\,t_0\,\};\, t \rangle$	$\to$
	$\langle \mathsf{E},\, f \mapsto \{\,\mathsf{E},\, (x\,:\,\tau \mid k\,:\,\neg\,\tau_0) \Rightarrow t_0\,\},\, c \mapsto \mathsf{V} \parallel t \rangle$	
(cnt)	$\langle \mathsf{E},\, c \mapsto \mathsf{V} \parallel \mathbf{cnt}\, k(x\,:\,\tau)\,\{\,t_0\,\};\, t \rangle$	$\to$
	$\langle \mathsf{E},\, k \mapsto \{\,\mathsf{E},\, c \mapsto \mathsf{V},\, (x\,:\,\tau) \Rightarrow t_0\,\} \parallel t \rangle$	

In accordance with the typing rules Let and Cnt, in rule *let* the function closure does not capture the continuation in the environment and leaves it in place, while in rule *cnt* the continuation closure does capture the continuation and removes it from the environment. Typing of the environment of a configuration and our restriction of typing contexts also ensure that there is always exactly one continuation closure in the environment of a configuration. Restricting the rules this way is not strictly necessary because typing ensures the pure usage of continuations, but having the restriction built into the machine simplifies the translations and theorems.

We call the pure fragment of our CPS calculus that we have just described $\lambda_{\mathsf{C}}^{\mathsf{pure}}$, and use it in the translations from and into $\lambda_{\mathsf{D}}^{\mathsf{pure}}$.

Type Safety

$\lambda_{\mathsf{C}}^{\mathsf{pure}}$ also satisfies the usual theorems of progress (Theorem 7) and preservation (Theorem 8). These theorems have been shown in Idris 2 by an intrinsically typed implementation and the totality of the step-function on machine states.

Theorem 7 (Progress (pure CPS))**.**
For a machine state M in $\lambda_{\mathsf{C}}^{\mathsf{pure}}$, if $\vdash M$ then either M is of the form $\langle E,\, k \mapsto \mathsf{done} \parallel k(e) \rangle$ or there exists a unique machine state M' such that $M \to M'$.

Theorem 8 (Preservation (pure CPS))**.**
For a machine state M in $\lambda_{\mathsf{C}}^{\mathsf{pure}}$, if $\vdash M$ and $M \to M'$ then $\vdash M'$.

```
def f(y: Int): Int {
  ret y + 1
};
val z = f(x);
ret z + 2
```

(a)

```
let f(y : Int ∣ k : ¬Int) {
  k(y + 1)
};
cnt k0(z) {
  done(z + 2)
};
f(x ∣ k0)
```

(b)

```
cnt f(y : Int) {
  cnt k0(z : Int) {
    done(z + 2)
  };
  k0(y + 1)
};
f(x)
```

(c)

```
val y = ret x;
val z = ret y + 1;
ret z + 2
```

(d)

Figure 2.10: Left to right: (a) Original program in DS, (b) CPS program obtained by translation, (c) Transformed (contified) CPS program, (d) Program translated back to DS.

Moreover, for the pure fragment, we have again that the result of evaluating a closed program with type τ has the same type.

Theorem 9 (Type preservation (pure CPS))**.**
If $k : \neg\tau \vdash t$ *and* $\langle k \mapsto \mathsf{done} \parallel t \rangle \to^* \langle E,\, k \mapsto \mathsf{done} \parallel k(e) \rangle$,
then $\Gamma \vdash e : \tau$ *where* $E \vdash_{env} \Gamma$.

Proof. None of the machine steps can add a done, so the final done is the initial done. Thus, e has the type that done expects, i.e., τ. □

2.3 To CPS and Back to DS

We now present the translations between the pure DS language and the pure fragment of the CPS language. We start with a basic example. Then we give the formal definitions of the translations on terms and on the abstract machines. In either case, we state how the translations act on typing derivations. Note, however, that no typing information is used in an essential way and the same translations work in an untyped setting. In fact, typing information is only needed for type annotations of parameters, and we often leave these annotations out in the definition of the translations. At the end, we show several properties the translations have.

2.3.1 Basic Example

To illustrate our languages and translations, we start with the basic example in Figure 2.10. We have already seen the programs in Subfigure 2.10a and Subfigure 2.10b in the previous sections. For better discriminability, we again use colors for DS programs and boldface for keywords in CPS programs.

Translating to CPS From the $\lambda_{\mathsf{D}}^{\mathsf{pure}}$ program in Subfigure 2.10a, our CPS translation produces the $\lambda_{\mathsf{C}}^{\mathsf{pure}}$ program in Subfigure 2.10b. Running our DS translation on this program in CPS produces exactly the original program in Subfigure 2.10a: It converts programs back to direct style.

Transforming We now transform the CPS program into an equivalent one. An example for such a transformation is *contification*, which specializes a function to one of its call sites. It is often applied in optimizing compilers (Kennedy, 2007) that use CPS as intermediate language. The program in Subfigure 2.10c is the result of applying contification to function f and its call site. The function f has become a continuation, hence the name contification. Instead of taking a continuation as a parameter, it now always jumps to k_0.

$$(\Gamma \vdash s : \tau \times k) \longrightarrow \Gamma, k : \neg\tau \vdash t$$

Translation of Statements

$$\begin{array}{lcll}
\mathcal{C}[\![\,\mathsf{ret}\; e\,]\!]_k & = & k(e) & \\
\mathcal{C}[\![\,\mathsf{val}\; x = s_0;\; s\,]\!]_k & = & \mathsf{cnt}\; k_0(x)\,\{\,\mathcal{C}[\![\,s\,]\!]_k\,\};\; \mathcal{C}[\![\,s_0\,]\!]_{k_0} & \text{where } k_0 \text{ fresh} \\
\mathcal{C}[\![\,f(e)\,]\!]_k & = & f(e \mid k) & \\
\mathcal{C}[\![\,\mathsf{def}\; f(x)\,\{\,s_0\,\};\; s\,]\!]_k & = & \mathsf{let}\; f(x \mid k_0)\,\{\,\mathcal{C}[\![\,s_0\,]\!]_{k_0}\,\};\; \mathcal{C}[\![\,s\,]\!]_k & \text{where } k_0 \text{ fresh}
\end{array}$$

Figure 2.11: Translation to pure continuation-passing style.

Translating back to DS Finally, we translate the manipulated CPS program back to direct style. Subfigure 2.10d shows the result of our DS translation. In this example, the resulting program is a sequence of bindings. All continuations have been eliminated. Running our CPS translation on this program produces exactly the program in CPS shown in Subfigure 2.10c.

Both translations are defined for the pure languages without any side conditions. This allows us to freely transform programs before translating back. On these pure languages, both of our translations are inverses of each other, syntactically (Theorems 17 and 18). Finally, both translations not only preserve semantics, but on the pure languages they also execute exactly the same number of steps before reaching the final state (Theorem 11 and Theorem 14).

2.3.2 From Direct Style to Continuation-Passing Style

We first describe the CPS translation on statements and then show how to extend it to the abstract machine. Note that expressions are translated trivially.

CPS Translation of Pure Statements

Figure 2.11 defines the CPS translation on statements. To avoid administrative redexes, the translation has an additional input, which stands for the current continuation represented by a continuation variable. The continuation variable is added to the typing context for the translated statement.

As usual, return statements are translated to calls to the current continuation. Sequencing is translated by defining a new continuation with the translation of the second statement as the body, and then translating the first statement with this new continuation. A translated function application, as usual, gets the current continuation as its continuation argument. For function definitions, the translated function definition abstracts over a fresh continuation parameter, which is then used to translate the body. The remaining statement is translated with the original input continuation.

CPS Translation of Pure Machine

The extension of the CPS translation to the abstract machine is shown in Figure 2.12. As can be seen in the translation on the typing judgments, stacks that expect values of type τ themselves become values of type $\neg\tau$.

In the translation of machine configurations, the stack is translated to a continuation closure value that is added to the translated environment with a fresh variable, and this variable is then used as the input continuation for the translation of the statement.

Environments are translated by pointwise translation of the bound values. For values there are two cases: Integers are translated trivially and closures are translated in essentially the same way as function definitions, but put together with the translated environment. For stacks, there are two cases. done frames are translated to done values. The translation of a frame on top of a remaining stack is similar to the case of machine configurations in that it adds the translation of

Translation of Values $\boxed{\vdash_{\text{val}} \mathsf{V} : \tau \longrightarrow \vdash_{\text{val}} \mathsf{V} : \tau}$

$$\begin{array}{lcll} \mathcal{C}_V[\![\{\, \mathsf{E},\, (x : \tau) \Rightarrow s \,\}]\!] & = & \{\, \mathcal{C}_E[\![\mathsf{E}]\!],\, (x : \tau \mid k : \neg \tau_0) \Rightarrow \mathcal{C}[\![s]\!]_k \,\} & \text{where } k \text{ fresh} \\ \mathcal{C}_V[\![19]\!] & = & 19 & \end{array}$$

Translation of Environments $\boxed{\mathsf{E} \vdash_{\text{env}} \Gamma \longrightarrow \mathsf{E} \vdash_{\text{env}} \Gamma}$

$$\begin{array}{lcl} \mathcal{C}_E[\![\bullet]\!] & = & \bullet \\ \mathcal{C}_E[\![\mathsf{E},\, x \mapsto \mathsf{V}]\!] & = & \mathcal{C}_E[\![\mathsf{E}]\!],\, x \mapsto \mathcal{C}_V[\![\mathsf{V}]\!] \end{array}$$

Translation of Stacks $\boxed{\tau \vdash_{\text{stk}} \mathsf{K} \longrightarrow \vdash_{\text{val}} \mathsf{V} : \neg \tau}$

$$\begin{array}{lcll} \mathcal{C}_K[\![\mathsf{done}]\!] & = & \mathsf{done} & \\ \mathcal{C}_K[\![\{\, \mathsf{E},\, (x : \tau) \Rightarrow s \,\} :: \mathsf{K}]\!] & = & \{\, \mathcal{C}_E[\![\mathsf{E}]\!],\, k \mapsto \mathcal{C}_K[\![\mathsf{K}]\!],\, (x : \tau) \Rightarrow \mathcal{C}[\![s]\!]_k \,\} & \text{where } k \text{ fresh} \end{array}$$

Translation of Configurations $\boxed{\vdash \mathsf{M} \longrightarrow \vdash \mathsf{M}}$

$$\mathcal{C}_M[\![\langle \mathsf{E} \parallel s \parallel \mathsf{K} \rangle]\!] = \langle \mathcal{C}_E[\![\mathsf{E}]\!],\, k \mapsto \mathcal{C}_K[\![\mathsf{K}]\!] \parallel \mathcal{C}[\![s]\!]_k \rangle \qquad \text{where } k \text{ fresh}$$

Figure 2.12: Translation of machine to pure continuation-passing style.

the remaining stack to the translated environment with a fresh variable, and uses this variable as input continuation for the translation of the body.

The translation illustrates the main difference between the DS machine and the CPS machine. While the DS machine uses a runtime stack, the CPS machine reifies this stack as a chain of nested continuation closures. There is one continuation closure per stack frame, closing over the continuation closure that reifies the next stack frame.

2.3.3 From Continuation-Passing Style Back to Direct Style

Let us now consider the opposite direction. We first describe how the DS translation behaves on terms and then extend it to the abstract machine. Again, expressions are translated trivially. We give a bottom-up translation in the style of inference rules. While we could also easily give a top-down translation for our pure languages, we will see in Chapter 3 that the style we use here is more well-suited for the full non-pure languages.

DS Translation of Pure Terms

Figure 2.13 defines the translation on terms. Where the CPS translation has an additional input, the DS translation has as an additional output a continuation variable standing for the stack the translated term returns to. This variable is then removed from the environment. Let us refer to this additional output as continuation output. In the pure case, this continuation output is unnecessary, because we have ensured that there is always exactly one continuation variable in the context, which stands for the current continuation, and the translated term must return to this variable. But we will see in Chapter 3 that the continuation output is crucial for programs with non-trivial control flow, since it tells us where we have to insert control operators for non-pure usage of continuations.

The translation is bottom-up in that it first recursively translates the subterms. As explained above, in the pure case the continuation output must always be the one continuation variable in the context. There are two base cases without subterms. The translation of a function application $f(e \mid k)$ is again a function application with the continuation output being the continuation passed to the function, which becomes the stack the translated term returns to. In the case of calling a continuation $k(e)$ the translation simply returns e with the continuation output being the called continuation k. Again, this is the stack that the translated term returns to.

$\boxed{\Gamma,\, k : \neg\tau \vdash t \;\longrightarrow (\Gamma \vdash s : \tau \rightsquigarrow k)}$

Translation of Terms

$$\frac{}{\mathcal{D}[\![\, f(e \mid k) \,]\!] = f(e) \rightsquigarrow k}\ [\textsc{DPApp}] \qquad \frac{}{\mathcal{D}[\![\, k(e) \,]\!] = \mathsf{ret}\, e \rightsquigarrow k}\ [\textsc{DPJmp}]$$

$$\frac{\mathcal{D}[\![\, t_0 \,]\!] = s_0 \rightsquigarrow k_0 \qquad \mathcal{D}[\![\, t \,]\!] = s \rightsquigarrow k}{\mathcal{D}[\![\, \mathsf{let}\, f(x \mid k_0)\, \{\, t_0 \,\};\, t \,]\!] = \mathsf{def}\, f(x)\, \{\, s_0 \,\};\, s \rightsquigarrow k}\ [\textsc{DPLet}]$$

$$\frac{\mathcal{D}[\![\, t_0 \,]\!] = s_0 \rightsquigarrow k \qquad \mathcal{D}[\![\, t \,]\!] = s \rightsquigarrow k_0}{\mathcal{D}[\![\, \mathsf{cnt}\, k_0(x)\, \{\, t_0 \,\};\, t \,]\!] = \mathsf{val}\, x = s;\, s_0 \rightsquigarrow k}\ [\textsc{DPCnt}]$$

Figure 2.13: Translation back to pure direct style.

For function definitions, the result of the translation is again a function definition in $\lambda_{\mathsf{D}}^{\mathsf{pure}}$. The translated body of the function definition must return to the continuation parameter k_0, and the continuation output of the remaining term is the continuation output of the whole statement. The definition of a continuation becomes a sequencing statement. Here, the translated body of the continuation definition determines the continuation output of the whole statement, while the translation of the remaining term must return to the continuation k_0 defined for it.

DS Translation of Pure Machine

Figure 2.14 shows how the DS translation is extended to the abstract machine. Note how, inverse to the CPS translation, values of type $\neg\tau$ become stacks that expect a value of type τ. Machine configurations are translated based on the translation s of the term in focus. This s must return to k, which must be bound to the one continuation value in the environment. This binding is removed from the translated environment and its translation is used as stack for the translated machine configuration. The translation function on the environment is hence only called on non-continuation bindings, which is always the case for the pure languages, as we will see below. Environments are again translated by pointwise translation of the bound (non-continuation) values.

For values, we distinguish continuation values from non-continuation ones. Non-continuation values are integers, which are translated trivially, and function closures, which are again translated in essentially the same way as function definitions put together with the translated environment. Note that the environment captured in a function closure never contains a continuation binding, as the function already binds a continuation variable itself. Continuation values are translated to stacks. The done value is translated to the done frame. The translation of continuation closures is similar to the translation of a machine configuration. The one continuation binding k the closure must have captured is removed from the translated environment, and the translation of the value bound to k is used as the remaining stack on top of which the frame for the closure is put. This turns the chain of nested continuation closures into a list of stack frames again, with one stack frame per continuation closure.

2.3.4 Properties

With the languages and translations in place, we want to show that the translations are well-behaved. To this end, we show several meta-theoretical properties.

Translation of Values $\boxed{\vdash_{\mathrm{val}} \mathsf{V} : \tau \longrightarrow \vdash_{\mathrm{val}} \mathsf{V} : \tau}$

$$\frac{\mathcal{D}[\![\, t_0 \,]\!] = s_0 \rightsquigarrow k_0}{\mathcal{D}_V[\![\, \{\, \mathsf{E},\, (x : t \mid k_0 : \neg\, t_0) \Rightarrow t_0 \,\}\,]\!] = \{\, \mathcal{D}_E[\![\, \mathsf{E} \,]\!],\, (x : t) \Rightarrow s_0 \,\}} \;[\textsc{DPFCLo}]$$

$$\frac{}{\mathcal{D}_V[\![\, 19 \,]\!] = 19} \;[\textsc{DPInt}]$$

Translation of Environments $\boxed{\mathsf{E} \vdash_{\mathrm{env}} \Gamma \longrightarrow \mathsf{E} \vdash_{\mathrm{env}} \Gamma}$

$$\frac{}{\mathcal{D}_E[\![\, \bullet \,]\!] = \bullet} \;[\textsc{DEnv1}] \qquad \frac{}{\mathcal{D}_E[\![\, \mathsf{E},\, x \mapsto \mathsf{V} \,]\!] = \mathcal{D}_E[\![\, \mathsf{E} \,]\!],\, x \mapsto \mathcal{D}_V[\![\, \mathsf{V} \,]\!]} \;[\textsc{DEnv2}]$$

Translation of Continuation Closures $\boxed{\vdash_{\mathrm{val}} \mathsf{V} : \neg\, \tau \longrightarrow \tau \vdash_{\mathrm{stk}} \mathsf{K}}$

$$\frac{}{\mathcal{D}_K[\![\, \mathsf{done} \,]\!] = \mathsf{done}} \;[\textsc{DPDone}]$$

$$\frac{\mathcal{D}[\![\, t_0 \,]\!] = s_0 \rightsquigarrow k}{\mathcal{D}_K[\![\, \{\, \mathsf{E},\, k \mapsto \mathsf{V},\, (x : t) \Rightarrow t_0 \,\}\,]\!] = \{\, \mathcal{D}_E[\![\, \mathsf{E} \,]\!],\, (x : t) \Rightarrow s_0 \,\} :: \mathcal{D}_K[\![\, \mathsf{V} \,]\!]} \;[\textsc{DPCCLo}]$$

Translation of Configurations $\boxed{\vdash \mathsf{M} \longrightarrow \vdash \mathsf{M}}$

$$\frac{\mathcal{D}[\![\, t \,]\!] = s \rightsquigarrow k}{\mathcal{D}_M[\![\, \langle \mathsf{E},\, k \mapsto \mathsf{V} \parallel t \rangle \,]\!] = \langle \mathcal{D}_E[\![\, \mathsf{E} \,]\!] \parallel s \parallel \mathcal{D}_K[\![\, \mathsf{V} \,]\!] \rangle} \;[\textsc{DPMch}]$$

Figure 2.14: Translation of machine back to pure direct style.

Typeability Preservation

We start with typeability preservation. This has been proven in Idris 2 by the intrinsic typing of the languages and the various translation functions on them.

Theorem 10 (Typeability preservation)**.**
All translations take typing derivations to typing derivations as stated above their definitions.

Semantics Preservation

Next, we prove semantics preservation. Since we use a small-step abstract machine semantics, we can be very precise about how semantics is preserved. In fact, for our pure languages, we have a step-by-step correspondence. In the following, we identify machine configurations and frames that can be identified by weakening, contraction, or reordering of environments, except for bindings of continuation type, which must be used linearly. Moreover, we consider equality always modulo α-equivalence. The proofs are straightforward and are contained in those given in Müller et al. (2023).

We start with the CPS translation. For every step the DS machine takes, the CPS machine also takes exactly one step.

Theorem 11 (Simulation of pure $\mathcal{C}_M$)**.**
If $\vdash M$ *and* $M \to M'$*, then* $\mathcal{C}_M[\![M]\!] \to \mathcal{C}_M[\![M']\!]$*.*

As a corollary, we obtain that the CPS machine takes exactly as many steps as the DS machine.

Corollary 12 (Operational reduction (pure CPS))**.**
If $\vdash M$ *and* $M \to^n M'$*, then* $\mathcal{C}_M[\![M]\!] \to^n \mathcal{C}_M[\![M']\!]$*.*

Proof. By induction on n using Theorem 11. □

In particular, a closed term that evaluates to a result in DS, evaluates to the same result in CPS in the same number of steps.

Corollary 13 (Evaluation (pure CPS))**.**
If $\vdash s : \tau$ *and* $\langle \bullet \parallel s \parallel \mathsf{done} \rangle \to^n \langle E \parallel \mathbf{ret}\, e \parallel \mathsf{done} \rangle$
then $\langle k \mapsto \mathsf{done} \parallel \mathcal{C}[\![s]\!]_k \rangle \to^n \langle \mathcal{C}_E[\![E]\!],\, k \mapsto \mathsf{done} \parallel k(e) \rangle$ *where* k *is fresh.*

Let us now look at the DS translation. Again, for every step the CPS machine takes, the DS machine also takes exactly one step.

Theorem 14 (Simulation of pure $\mathcal{D}_M$)**.**
If $\vdash M$ *and* $M \to M'$*, then* $\mathcal{D}_M[\![M]\!] \to \mathcal{D}_M[\![M']\!]$*.*

We obtain similar corollaries to those the CPS translation above.

Corollary 15 (Operational reduction (pure DS))**.**
If $\vdash M$ *and* $M \to^n M'$*, then* $\mathcal{D}_M[\![M]\!] \to^n \mathcal{D}_M[\![M']\!]$*.*

Proof. By induction on n using Theorem 14. □

Corollary 16 (Evaluation (pure DS))**.**
If $\vdash t$ *with* $\mathcal{D}[\![t]\!] = s \rightsquigarrow k$ *and* $\langle k \mapsto \mathsf{done} \parallel t \rangle \to^n \langle E,\, k \mapsto \mathsf{done} \parallel k(e) \rangle$
then $\langle \bullet \parallel s \parallel \mathsf{done} \rangle \to^n \langle \mathcal{D}_E[\![E]\!] \parallel \mathbf{ret}\, e \parallel \mathsf{done} \rangle$*.*

Round Trips

Another interesting question is how the compositions of the two translations behave. For our pure languages, the translations have the particularly pleasing property of being syntactic inverses of each other.

Theorem 17 (Pure $\mathcal{D}$ is right inverse of pure $\mathcal{C}$)**.**
If $\Gamma \vdash t$ *and* $\mathcal{D}[\![t]\!] = s \rightsquigarrow k$*, then* $\mathcal{C}[\![s]\!]_k = t$*.*

Theorem 18 (Pure $\mathcal{D}$ is left inverse of pure $\mathcal{C}$)**.**
If $\Gamma \vdash s : \tau$*, then* $\mathcal{D}[\![\mathcal{C}[\![s]\!]_k]\!] = s \rightsquigarrow k$ *where* k *is fresh.*

We can extend this property to the abstract machines.

Theorem 19 (Right inverse for pure machines)**.**
If $\vdash M$*, then* $\mathcal{C}_M[\![\mathcal{D}_M[\![M]\!]]\!] = M$*.*
If $\vdash_{val} V : \tau$*, then* $\mathcal{C}_V[\![\mathcal{D}_V[\![V]\!]]\!] = V$*.*
If $E \vdash_{env} \Gamma$*, then* $\mathcal{C}_E[\![\mathcal{D}_E[\![E]\!]]\!] = E$*.*
If $\vdash_{val} V : \neg\tau$*, then* $\mathcal{C}_K[\![\mathcal{D}_K[\![V]\!]]\!] = V$*.*

Theorem 20 (Left inverse for pure machines)**.**
If $\vdash M$*, then* $\mathcal{D}_M[\![\mathcal{C}_M[\![M]\!]]\!] = M$*.*
If $\vdash_{val} V : \tau$*, then* $\mathcal{D}_V[\![\mathcal{C}_V[\![V]\!]]\!] = V$*.*
If $E \vdash_{env} \Gamma$*, then* $\mathcal{D}_E[\![\mathcal{C}_E[\![E]\!]]\!] = E$*.*
If $\tau \vdash_{stk} K$*, then* $\mathcal{D}_K[\![\mathcal{C}_K[\![K]\!]]\!] = K$*.*

Semantics Reflection

The round-trip results show that we can go forth and back freely between the pure DS calculus and the pure fragment of the CPS calculus during evaluation. In particular, we can reflect the machine steps taken in either machine. That is, if there is a step between machine configurations in the translated machine, then there must be exactly one step between the original states.

Corollary 21 ($Pure\ \mathcal{D}_M$ is step-reflecting)**.**
Given $\vdash M$ *and* $\vdash M'$*, then* $\mathcal{D}_M[\![M]\!] \to \mathcal{D}_M[\![M']\!]$ *implies* $M \to M'$*.*

Corollary 22 (Pure $\mathcal{C}_M$ is step-reflecting)**.**
Given $\vdash M$ *and* $\vdash M'$*, then* $\mathcal{C}_M[\![M]\!] \to \mathcal{C}_M[\![M']\!]$ *implies* $M \to M'$*.*

2.4 Conclusion

We have presented a translation from continuation-passing style back to direct style in a typed setting on the pure fragment of our CPS calculus. It is the full syntactic inverse of our CPS translation. Both translations are total and preserve the small-step evaluation of the abstract machine semantics of the languages step-by-step, giving a 1-to-1 correspondence between the DS calculus and the pure fragment of the CPS calculus. Moreover, both translations are first-order, one-pass, and compositional, facilitating proofs by structural induction. Next, in Chapter 3, we will extend the translations to non-pure programs with non-trivial control flow by endowing the DS calculus with control operators.

3 Non-Trivial Control Flow and Control Operators

In Chapter 2 we have seen translations into continuation-passing style and back to direct style on pure programs. We have presented the pure DS language $\lambda_{\mathsf{D}}^{\mathsf{pure}}$ and the CPS language λ_{C} along with its pure fragment $\lambda_{\mathsf{C}}^{\mathsf{pure}}$. However, in λ_{C} continuations are first-class citizens and we cannot express the resulting possibilities for non-pure usage of continuations in $\lambda_{\mathsf{D}}^{\mathsf{pure}}$. Therefore, to be able to translate the full CPS calculus back to DS, we need to extend the DS calculus with control operators that enable non-trivial control flow in direct style. We call the resulting language λ_{D}. This also allows programmers to structure their DS programs using these control operators. Translating CPS programs back to DS then requires the insertion of control operators in some cases. Our goal is to insert them only where necessary, i.e., where continuations are used non-trivially.

As an example, consider the programs in Figure 3.1. For better discriminability, we again use colors for DS programs and boldface for keywords in CPS programs. The programs use the following type of results that may either be a success or a failure.

```
data Result { Success(String), Failure(String) }
```

Furthermore, they use a primitive `#readFile` operation for asynchronous file access. It receives a file path and two callbacks: one for success and one for failure.

Programming with control operators In Subfigure 3.1a we use control operators to define a function `readFile` that returns a `Result`, but internally uses the asynchronous operation `#readFile`. In the implementation of `readFile`, we use the control operator `suspend` to *capture and remove* the current call stack and make it available under the name `onReturn`. We then define two *new processes* `onSuccess` and `onFailure`, and invoke `#readFile` with these. Both processes resume the original computation, one with `Success` and the other with `Failure`. It is important to emphasize: The body of `suspend` does *not* run in the context of a call stack. This means that we can neither return a value nor call a function. All we can do is call primitive operations or resume processes. The same is true for the bodies of processes. The function `tryReadingFile` can now be written straightforwardly in direct style. The function `readFile` acts as a shield that protects the rest of the program from being infected by the asynchronous nature of the operation `#readFile`.

Translating control operators to CPS We translate this program to CPS. The result is the program in Subfigure 3.1b. This is roughly what we would have to write manually in a language without control operators. The whole program, including the function *tryReadingFile*, must be written in CPS. A CPS translation automates this process and can thus serve as an implementation technique for control operators.

Transforming control operators We now transform the CPS program. We use some standard inlining and reduction. The transformed version of the previous program is shown in Subfigure 3.1d. Concretely, we have inlined *readFile*, inlined k_0, and reduced the exposed matches on known constructors. While this transformation is trivial in CPS, it would be difficult to achieve the same result directly in DS due to the use of control operators. An optimizer performing this transformation in DS would have to see through the control operators.

```
def readFile(path: Path): Result {
  suspend { onReturn ⇒
    process onSuccess(content: String) {
      resume onReturn(Success(content))
    };
    process onFailure(error: String) {
      resume onReturn(Failure(error))
    };
    #readFile(path, onSuccess, onFailure)
  }
};

def tryReadingFile(path: Path): String {
  val result = readFile(path);
  match result {
    case Success(_) ⇒ ret "success"
    case Failure(_) ⇒ ret "failure"
  }
}
```

(a) Original λ_{D} program.

```
let readFile(path : Path ⫞ onReturn : ¬Result) {
  cnt onSuccess(content) {
    onReturn(Success(content))
  };
  cnt onFailure(error) {
    onReturn(Failure(error))
  };
  #readFile(path, onSuccess, onFailure)
};

let tryReadingFile(path : Path ⫞ cont : ¬String) {
  cnt k0(result) {
    match result {
      case Success(_) ⇒ cont("success")
      case Failure(_) ⇒ cont("failure")
    }
  };
  readFile(path ⫞ k0)
}
```

(b) CPS translation into λ_{C}.

```
def tryReadingFile(path: Path): String {
  suspend { cont ⇒
    process onSuccess(_) {
      resume cont("success") };
    process onFailure(_) {
      resume cont("failure") };
    #readFile(path, onSuccess, onFailure)
  }
}
```

(c) DS translation of transformed version into λ_{D}.

```
let tryReadingFile(path : Path ⫞ cont : ¬String) {
  cnt onSuccess(_) {
    cont("success") };
  cnt onFailure(_) {
    cont("failure") };
  #readFile(path, onSuccess, onFailure)
}
```

(d) Transformed λ_{C} program.

Figure 3.1: Example program using control operators to program with asynchronous IO.

Translating control operators back to DS Unfortunately, the resulting program is still in CPS and does not use the call stack anymore. On platforms that have call stacks, it is often beneficial to avoid programs in CPS since they might allocate continuations on the heap, rather than using the stack. Luckily, we can translate this program back to direct style. Our translation results in approximately[2] the program in Subfigure 3.1c. In this example, our translation had to insert a use of the control operator `suspend`. There is no way around this, as the continuation `cont` is used twice. The resulting function `tryReadingFile` has the same type as the original one, and it can be used in direct style wherever the original function was used. Indeed, this is always the case for our translations (Theorem 26). Moreover, of course, our translations guarantee that the resulting programs have the same semantics (Corollaries 29 and 32).

In general, composing our translations between λ_{C} and λ_{D} results in the same program when starting at λ_{C}. In the other direction, this is not always the case. Consider the following contrived example on the left, which uses control operators to define the identity function.

```
def identity(a: A): A {
  suspend { k ⇒
    resume k(a)
  }
}
```

```
let identity(a : A ! k : ¬A) {
  k(a)
}
```

```
def identity(a: A) {
  ret a
}
```

Our CPS translation yields the term in λ_{C} in the middle. When we convert this program back to direct style, we obtain the result on the right. Indeed, since the continuation is used trivially in this example, the use of control operators is not needed. In such cases, the composition of our translations will remove them. However, a round trip like this is not always a simplification, even if it does not result in the same program. This is because different combinations of control operators in a DS term that are not simplifications of one another can yield the same CPS term. Nevertheless, a round trip will not insert an unreasonable amount of control operators, as can be seen from the results on operational correspondence.

In this chapter we extend the pure DS language $\lambda_{\mathsf{D}}^{\mathsf{pure}}$ from Section 2.1 with a set of control operators in Section 3.1. We call the resulting calculus λ_{D}.

- The extended language also extends the type system, which means that it also admits a logical interpretation, which we will look at in Chapter 4.
- We also extend the abstract machine semantics to the full language. This allows us to prove an operational correspondence with the full CPS language with tight bounds on the number of steps.
- The extended language still satisfies the properties of progress and preservation (Theorems 23 and 24).

In Section 3.2 we also extend the CPS translation and the DS translation from Section 2.3.

- Both translations are now defined on the whole language, in other words, they work on all programs. This is important if we want to transform programs before translating back.
- Both are compositional, first-order, and one-pass.
- Both preserve well-typedness (Theorem 26).
- Both preserve semantics (Theorems 27 and 30). Indeed, while there is no step-by-step correspondence anymore, we prove a step-wise correspondence with a global upper bound on the number of steps.

[2]The actual result uses a more complicated combination of control operators in order to make the theorem of operational correspondence true.

- Neither duplicates code.
- Both are implemented in the total programming language Idris 2 (Brady, 2020) to make sure all cases are covered. The DS translation has a lot of corner cases and side conditions.

Moreover, when taken together, these two translations have the following desirable properties.

- While the translations are not full inverses anymore, the DS translation is still a syntactic right inverse of the CPS translation. When we translate a term to DS and back, we arrive at the same term (Theorem 33).
- We again extend both translations to machine states, which allows us to freely translate back and forth between arbitrary intermediate states.

None of the existing pairs of CPS and DS translations in the literature have all of these properties at the same time. A thorough comparison can be found in Section 3.4. We further consider an extension of the calculi with a construct for conditional branching, and discuss how the translations can deal with such constructs in Section 3.3. In Section 3.5 we conclude and briefly lay out future work.

3.1 The DS Language with Control Operators λ_{D}

We start by extending the pure DS language $\lambda_{\mathsf{D}}^{\mathsf{pure}}$ with control operators to enable non-trivial control flow. After briefly discussing the set of control operators we use, we present the extensions of the syntax and typing rules and of the abstract machine.

3.1.1 Control Operators

There are several different control operators for undelimited continuations[3], of which Landin's operator J (Landin, 1998, 1965a; Felleisen, 1987; Thielecke, 1998) likely is the oldest. The probably most well-known such control operators are `call/cc` or its variation `let/cc` (which Reynolds (1972) called **escape**) and Felleisen et al. (1987)'s $\mathcal{C}$. They capture the current continuation and bind it to a variable that is made available in the body of the construct. The only difference between `call/cc` and `let/cc` is that the former binds the continuation via a lambda, while in the latter the binder for the continuation is built into the construct. The main difference between `call/cc` and $\mathcal{C}$ is that the former leaves the captured continuation in place, while the latter does not. Therefore, if the captured continuation is never invoked in the body, `call/cc` behaves as a no-op, while $\mathcal{C}$ discards the continuation. This makes $\mathcal{C}$ a bit more powerful and flexible than `call/cc` since the latter cannot discard the current continuation, but $\mathcal{C}$ can easily express `call/cc` by simply wrapping a call to the just captured continuation around the body.

$$\texttt{call/cc}\,(\lambda k \Rightarrow s) \equiv \mathcal{C}\,(\lambda k \Rightarrow k(s))$$

Modeling $\mathcal{C}$ with `call/cc` necessitates an abortion operator that discards the current continuation surrounding it. Using the abortion operator $\mathcal{A}$ of Felleisen et al., one can define

$$\mathcal{C}\,(\lambda k \Rightarrow s) \equiv \texttt{call/cc}\,(\lambda k \Rightarrow \mathcal{A}(s))$$

suspend and run

The control operator **suspend** $\{\, k \Rightarrow s \,\}$ that we use in our language λ_{D} for capturing the current continuation, has the binding k for the continuation built-in, similar to `let/cc`. Moreover, it

[3] There are yet more for delimited continuations, but we do not consider those here.

is like $\mathcal{C}$ in that it is abortive, that is, it does not leave the captured continuation in place. As we will see, this nicely fits the design of our translation back to DS.

The continuation captured by `call/cc` or $\mathcal{C}$ can be applied to an argument in the same way as a function. However, as we have discussed in Section 2.2.1 already, continuations and functions are different concepts, philosophically and practically. Continuations reify the rest of the computation, representing the stack. In direct style, an invocation of a continuation should hence rather be seen as installing the stack of the process it represents. Moreover, for the purposes of our translation into CPS, it is particularly helpful to be able to distinguish an invocation of a continuation syntactically. We therefore include a construct $\mathbf{run}(k)\,\{\,s\,\}$ that runs the statement in its body in the context of the process bound to k. The previously used form $\mathbf{resume}\,k(v)$ is simply syntactic sugar for $\mathbf{run}(k)\,\{\,\mathbf{ret}\,v\,\}$, which first installs the stack of k and then immediately returns the value v to it.

Together, these two constructs **suspend** and **run** allow us to flexibly adapt the stack during the translation back to direct style. Similar pairs of control operators appear in Materzok and Biernacki (2012), Biernacka et al. (2011), and Kiselyov and Shan (2007).

process and exit

We include two more constructs in λ_{D}. First, we include an **exit** statement to mirror the **exit** term in the CPS language λ_{C}. As **suspend** is abortive, we could also model this with **suspend** and **run** by capturing the initial stack (consisting of a done frame) at the very beginning of the program s.

$$\mathbf{suspend}\,\{\,k_{\mathbf{exit}} \Rightarrow \mathbf{run}(k_{\mathbf{exit}})\,\{\,s\,\}\,\}$$

Then we could use $\mathbf{run}\,(k_{\mathbf{exit}})\,\{\,\mathbf{ret}\,e\,\}$ instead of using $\mathbf{exit}\,e$ directly to end a program. In a similar vein, in λ_{C} we could also use the initial free continuation variable instead of using **exit**. However, in both cases, it is simpler and feels clearer to have **exit** directly available as a language construct, and to define the initial stack or continuation in terms of it.

Second, we include a construct $\mathbf{process}\,k(x)\,\{\,s\,\};\,s_0$ to construct a process and bind it to a name k for the remaining statement s_0. Being able to have named processes simplifies the translation back to DS, because the continuations in our CPS language are explicitly named as well. Again, we could also model this with **suspend** and **run** via something like

$$\mathbf{run}(k_{\mathbf{exit}})\,\{\,\mathbf{val}\,x\;=\;\mathbf{suspend}\,\{\,k \Rightarrow s_0\,\};\,\mathbf{suspend}\,\{\,\_ \Rightarrow s\,\}\,\}$$

This installs an initial stack $k_{\mathbf{exit}}$ and pushes a frame for the process that, when invoked, ignores the rest of the stack and runs its body s. The new stack is then captured and bound to variable k in s_0. This encoding, however, is arguably less clear and readable.

Hence, while in principle **suspend** and **run** suffice to allow for a translation back to direct style, including **exit** and **process** makes the correspondence between the DS language and the CPS language more apparent.

3.1.2 Syntax

Figure 3.2 defines the syntax of our DS language λ_{D}. As we have seen for the pure fragment $\lambda_{\mathsf{D}}^{\mathsf{pure}}$ in Section 2.1, we syntactically distinguish between statements, which can have control effects, and expressions, which cannot. The syntax of expressions is still exactly the same as for $\lambda_{\mathsf{D}}^{\mathsf{pure}}$, which we indicate by having it grayed out.

Syntax of Statements

The syntax of statements in $\lambda_{\mathsf{D}}^{\mathsf{pure}}$ is extended with control operators. The control operator $\mathbf{suspend}\,\{\,k \Rightarrow s\,\}$ captures the current continuation, i.e., the current stack, and makes it

Syntax of λ_{D}

Statements	s	::=	$\mathsf{val}\, x = s;\, s$	sequence
		|	$\mathsf{ret}\, e$	return
		|	$\mathsf{def}\, f(x : \tau)\, \{\, s \,\};\, s$	define
		|	$f(e)$	call
		|	$\mathsf{process}\, k(x : \tau)\, \{\, s \,\};\, s$	process
		|	$\mathsf{suspend}\, \{\, k \Rightarrow s \,\}$	suspend
		|	$\mathsf{run}(e)\, \{\, s \,\}$	run
		|	$\mathsf{exit}\, e$	exit
Variables	v	::=	x, f, k	variables
Expressions	e	::=	v	variables
		|	$0 \mid 1 \mid \ldots$	integers

Syntax of Types

Types	τ	::=	$\tau \to \tau$	function type
		|	$\neg\, \tau$	stack type
		|	Int	base type
Environment Type	Γ	::=	$\Gamma,\, x : \tau$	extended environment
		|	$\emptyset$	empty environment
Context Type	ξ	::=	τ	delimited context
		|	$\#$	undelimited context

Figure 3.2: The direct-style language λ_{D}.

available as a process in its body s by binding it to variable k. As we have discussed in Section 3.1.1 and will see shortly in the typing rules and the operational semantics, it is more akin to Felleisen's $\mathcal{C}$ than to `call/cc`, since it does not leave the stack in place. The counterpart $\mathsf{run}(e)\, \{\, s \,\}$ runs statement s in the context of a process e. Using the $\mathsf{process}\, k(x : \tau)\, \{\, s_1 \,\};\, s_2$ statement, such a process can be constructed and bound to k in the environment for s_2. A program can be terminated with an expression e using $\mathsf{exit}\, e$.

Syntax of Types and Context Types

The syntax of types is extended with a special type $\neg\, \tau$ for stacks expecting an argument of type τ. This means that reified stacks can now be passed to functions as arguments. Stack types correspond to continuation types in λ_{C}. Typing environments are not changed.

In contrast to the pure DS language, we now have two kinds of contexts, and consequently two kinds of context types. Besides the types τ indicating that a statement runs in the delimiting context of a stack it returns to, there is also the context type $\#$, which signifies that a statement is undelimited and does not return at all.

3.1.3 Typing

The typing rules for λ_{D} are given in Figure 3.3. The judgment and rules for expressions stay exactly the same. Statements, however, are now typed with a context type ξ, i.e., either with an ordinary type τ or with $\#$. That is, some of them require a stack to be present, others do not, yet others are polymorphic. The rules for sequencing, returning, function definition, and application are still mostly unsurprising. But note that since the only purpose of the **val**-construct is to sequence statements, it cannot be typed with context type $\#$. It would, however, be straightforward to add another construct that allows us to bind an expression to a variable in a subsequent statement having an arbitrary context type. Moreover, in rule Define, the

Statement Typing $\boxed{\Gamma \vdash s : \xi}$

$$\frac{\Gamma \vdash s_0 : \tau_0 \quad \Gamma, x_0 : \tau_0 \vdash s : \tau}{\Gamma \vdash \mathbf{val}\, x_0 = s_0;\, s : \tau}\ [\textsc{Sequence}] \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathbf{ret}\, e : \tau}\ [\textsc{Return}]$$

$$\frac{\Gamma, x : \tau \vdash s_0 : \tau_0 \quad \Gamma, f : \tau \to \tau_0 \vdash s : \xi}{\Gamma \vdash \mathbf{def}\, f(x : \tau)\,\{\, s_0 \,\};\, s : \xi}\ [\textsc{Define}] \qquad \frac{f : \tau \to \tau_0 \in \Gamma \quad \Gamma \vdash e : \tau}{\Gamma \vdash f(e) : \tau_0}\ [\textsc{Call}]$$

$$\frac{\Gamma, x : \tau \vdash s_0 : \# \quad \Gamma, k : \neg\tau \vdash s : \xi}{\Gamma \vdash \mathbf{process}\, k(x : \tau)\,\{\, s_0 \,\};\, s : \xi}\ [\textsc{Process}] \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathbf{exit}\, e : \#}\ [\textsc{Exit}]$$

$$\frac{\Gamma, k : \neg\tau \vdash s : \#}{\Gamma \vdash \mathbf{suspend}\,\{\, k \Rightarrow s \,\} : \tau}\ [\textsc{Suspend}] \qquad \frac{\Gamma \vdash e : \neg\tau \quad \Gamma \vdash s : \tau}{\Gamma \vdash \mathbf{run}(e)\,\{\, s \,\} : \#}\ [\textsc{Run}]$$

Expression Typing $\boxed{\Gamma \vdash e : \tau}$

$$\frac{v : \tau \in \Gamma}{\Gamma \vdash v : \tau}\ [\textsc{Var}] \qquad \frac{}{\Gamma \vdash 19 : \mathsf{Int}}\ [\textsc{Int}]$$

Figure 3.3: Typing rules for λ_{D}.

whole statement must have the same context type ξ as the rest of the statement s. This means that such statements have context type # exactly when they occur in a context where no stack is available.

The same is true for newly defined processes in rule Process. Processes themselves never return and as such the body of a process definition has context type #. Exiting a program does of course not return either, as can be seen in rule Exit. In rule Suspend, the overall statement has type τ, thus the captured stack has type $\neg\tau$. It is available in the body s under the name k. The body does not return, and the captured stack is undelimited, so **suspend** can be understood as shifting to the undelimited world. Dually, as rule Run shows, **run** shifts back to the delimited world. The delimiting process e has type $\neg\tau$ and consequently the statement s has to return a value of type τ. In the following, we will usually assume the expression e to be a variable.

3.1.4 Operational Semantics

For the operational semantics of λ_{D} we extend the abstract machine of $\lambda_{\mathsf{D}}^{\mathsf{pure}}$.

Machine States

Figure 3.4 defines the syntax of the machine. There are now two different modes of machine states, one for delimited execution and one for undelimited execution. The delimited mode is the same as for $\lambda_{\mathsf{D}}^{\mathsf{pure}}$. The statement in the undelimited mode, however, is non-returning and thus there is no stack. Environments still map variables to values, but values can now also be stacks. A stack is still a list of frames whose last frame is an underflow frame, but underflow frames are now general frames, not only done frames. In fact, done is now defined to be the underflow frame $\{\, \bullet, (x : \tau) \Rightarrow \mathbf{exit}\, x \,\}$, just as in λ_{C}. Note that while the syntax of closures and underflow frames is the same, the body of a closure is a returning statement, whereas the body of an underflow frame is undelimited. The final states of the machine are now of the form $\langle \mathsf{E} \parallel \mathbf{exit}\, e \rangle$, again just as in λ_{C}. We still use done as a bottom frame to start evaluation of a closed program s in state $\langle\, \bullet \parallel s \parallel \mathsf{done} \rangle$. Typing of the machine is unsurprising (Figure 3.5

Syntax of Machine States for λ_{D}

Values	V	::=	$\{\,\mathsf{E},\,(x\,:\,\tau)\Rightarrow s\,\}$	closure
		|	K	stack
		|	$0 \mid 1 \mid \ldots$	integer
Environments	E	::=	$\mathsf{E},\,x\mapsto\mathsf{V}$	binding
		|	$\bullet$	empty
Stacks	K	::=	$\{\,\mathsf{E},\,(x\,:\,\tau)\Rightarrow s\,\}$	underflow
		|	$\{\,\mathsf{E},\,(x\,:\,\tau)\Rightarrow s\,\}\,::\,\mathsf{K}$	frame
Configurations	M	::=	$\langle\,\mathsf{E}\parallel s\parallel\mathsf{K}\,\rangle$	delimited execution
		|	$\langle\,\mathsf{E}\parallel s\,\rangle$	undelimited execution

Figure 3.4: Operational semantics of λ_{D}.

Configuration Typing $\boxed{\vdash \mathsf{M}}$

$$\frac{\mathsf{E}\vdash_{\mathrm{env}}\Gamma \qquad \Gamma\vdash s:\tau \qquad \tau\vdash_{\mathrm{stk}}\mathsf{K}}{\vdash\langle\,\mathsf{E}\parallel s\parallel\mathsf{K}\,\rangle}\;[\textsc{Delim}] \qquad \frac{\mathsf{E}\vdash_{\mathrm{env}}\Gamma \qquad \Gamma\vdash s:\#}{\vdash\langle\,\mathsf{E}\parallel s\,\rangle}\;[\textsc{Undelim}]$$

Figure 3.5: Typing rules for machine states of λ_{D}.

defines the rules for configurations; the full set of rules is given in Müller et al. (2023)). The rule for delimited machine states is still the same as in $\lambda_{\mathsf{D}}^{\mathsf{pure}}$. For undelimited states, there is no stack, so the statement in focus must be of type #.

Machine Steps

The evaluation steps of the abstract machine are defined in Figure 3.6. Rules *push* for pushing a frame and *call* for calling a function are as in $\lambda_{\mathsf{D}}^{\mathsf{pure}}$. But the rule for returning to the stack now comes in two flavors, depending on whether there is more than one frame left on the stack or not. Rule *ret-1* is the same as in the pure DS language. However, if the frame returned to was the last one, in rule *ret-0*, then the machine continues in undelimited mode. Similarly, there are two rules for defining functions, *def-1* and *def-0*, depending on whether the remaining statement returns or not. We have omitted the rules for returning integers and calling functions on integers, as they are essentially the same as the rules with variables. Rules *proc-1* and *proc-0* for defining a process are essentially the same as the ones for defining functions, but note that

(push)	$\langle\,\mathsf{E}\parallel\mathbf{val}\,x_0\,=\,s_0;\,s\parallel\mathsf{K}\,\rangle$	$\to$	$\langle\,\mathsf{E}\parallel s_0\parallel\{\,\mathsf{E},\,(x_0\,:\,\tau_0)\Rightarrow s\,\}\,::\,\mathsf{K}\,\rangle$
(ret-1)	$\langle\,\mathsf{E},\,v\mapsto\mathsf{V}\parallel\mathbf{ret}\,v\parallel\{\,\mathsf{E}_0,\,(x\,:\,\tau)\Rightarrow s\,\}\,::\,\mathsf{K}\,\rangle$	$\to$	$\langle\,\mathsf{E}_0,\,x\mapsto\mathsf{V}\parallel s\parallel\mathsf{K}\,\rangle$
(ret-0)	$\langle\,\mathsf{E},\,v\mapsto\mathsf{V}\parallel\mathbf{ret}\,v\parallel\{\,\mathsf{E}_0,\,(x\,:\,\tau)\Rightarrow s\,\}\,\rangle$	$\to$	$\langle\,\mathsf{E}_0,\,x\mapsto\mathsf{V}\parallel s\,\rangle$
(def-1)	$\langle\,\mathsf{E}\parallel\mathbf{def}\,f(x\,:\,\tau)\,\{\,s_0\,\};\,s\parallel\mathsf{K}\,\rangle$	$\to$	$\langle\,\mathsf{E},\,f\mapsto\{\,\mathsf{E},\,(x\,:\,\tau)\Rightarrow s_0\,\}\parallel s\parallel\mathsf{K}\,\rangle$
(def-0)	$\langle\,\mathsf{E}\parallel\mathbf{def}\,f(x\,:\,\tau)\,\{\,s_0\,\};\,s\,\rangle$	$\to$	$\langle\,\mathsf{E},\,f\mapsto\{\,\mathsf{E},\,(x\,:\,\tau)\Rightarrow s_0\,\}\parallel s\,\rangle$
(call)	$\langle\,\mathsf{E},\,f\mapsto\{\,\mathsf{E}_0,\,(x\,:\,\tau)\Rightarrow s\,\},\,v\mapsto\mathsf{V}\parallel f(v)\parallel\mathsf{K}\,\rangle$	$\to$	$\langle\,\mathsf{E}_0,\,x\mapsto\mathsf{V}\parallel s\parallel\mathsf{K}\,\rangle$
(proc-1)	$\langle\,\mathsf{E}\parallel\mathbf{process}\,k(x\,:\,\tau)\,\{\,s_0\,\};\,s\parallel\mathsf{K}\,\rangle$	$\to$	$\langle\,\mathsf{E},\,k\mapsto\{\,\mathsf{E},\,(x\,:\,\tau)\Rightarrow s_0\,\}\parallel s\parallel\mathsf{K}\,\rangle$
(proc-0)	$\langle\,\mathsf{E}\parallel\mathbf{process}\,k(x\,:\,\tau)\,\{\,s_0\,\};\,s\,\rangle$	$\to$	$\langle\,\mathsf{E},\,k\mapsto\{\,\mathsf{E},\,(x\,:\,\tau)\Rightarrow s_0\,\}\parallel s\,\rangle$
(sus)	$\langle\,\mathsf{E}\parallel\mathbf{suspend}\,\{\,k\Rightarrow s\,\}\parallel\mathsf{K}\,\rangle$	$\to$	$\langle\,\mathsf{E},\,k\mapsto\mathsf{K}\parallel s\,\rangle$
(run)	$\langle\,\mathsf{E},\,v\mapsto\mathsf{K}\parallel\mathbf{run}(v)\,\{\,s\,\}\,\rangle$	$\to$	$\langle\,\mathsf{E},\,v\mapsto\mathsf{K}\parallel s\parallel\mathsf{K}\,\rangle$

Figure 3.6: Machine steps of λ_{D}.

such a process always consists of one undelimited underflow frame instead of a closure. Rule *sus* captures the current stack, binds it as a process to k in the environment, and executes the undelimited statement in the body. In particular, if k is not used in the statement, then the stack is discarded. Rule *run* looks up the process bound to variable v in the environment, and runs the body s with this process installed as a stack. Note that the binding for the process in the environment persists, as s may still contain v free.

Type Safety

λ_{D} still satisfies the usual theorems of progress (Theorem 23) and preservation (Theorem 24). These theorems have been shown in Idris 2 by an intrinsically typed implementation and the totality of the step-function on machine states.

Theorem 23 (Progress (DS))**.**
For DS machine state M, if $\vdash M$ then either M is of the form $\langle E \parallel \textbf{\textit{exit}}\, e \rangle$ or there exists a unique machine state M' such that $M \to M'$.

Theorem 24 (Preservation (DS))**.**
For DS machine state M, if $\vdash M$ and $M \to M'$ then $\vdash M'$.

Moreover, if a closed program with type τ does not contain **exit**, then the resulting expression has the same type. Note that if a program does contain **exit**, it may end with an expression having a different type.

Theorem 25 (Type preservation (DS))**.**
*If $\vdash s : \tau$ and s does not contain **exit** and $\langle \bullet \parallel s \parallel \mathsf{done} \rangle \to^* \langle E \parallel \textbf{\textit{exit}}\, e \rangle$, then $\Gamma \vdash e : \tau$ where $E \vdash_{env} \Gamma$.*

Proof. None of the machine steps can add an **exit**, so the final **exit** comes from returning to the initial done. Thus, e has the type that done expects, i.e., τ. □

3.2 To CPS and Back to DS with Control

Now that we have enabled our DS calculus to express non-trivial control flow, we can extend our translations to CPS and back to DS from Section 2.3 to the full languages. We again state how the translations act on typing derivations, but note that typing information is still only used for type annotations of parameters, which we often leave out in the definition of the translations. That is, the same translations still work in an untyped setting.

3.2.1 From Direct Style to Continuation-Passing Style

We first describe the CPS translation on statements and then show how to extend it to the abstract machine. Expressions are still translated trivially.

CPS Translation of Statements

Figure 3.7 defines the CPS translation on statements. The translation still has an additional input, which stands for the current continuation. But since there are non-returning statements now, this continuation input is either a continuation variable as before or no continuation (indicated with $\bullet$). In the first case, the continuation variable is added to the typing context for the translated statement, while in the second case, the typing context stays exactly the same.

The cases present in the pure translation (grayed out here) are still the same. This includes returning, sequencing, function application, and function definition in the case where the remaining statement returns. For function definitions, there is a second case for a non-returning remaining statement. In that case, the remaining statement is translated with no continuation.

$(\Gamma_1, \Gamma_2 \vdash s : \tau \times k) \longrightarrow \Gamma_1, k : \neg\tau, \Gamma_2 \vdash t$ $\qquad$ $(\Gamma \vdash s : \# \times \bullet) \longrightarrow \Gamma \vdash t$

Translation of Statements

$$
\begin{array}{lcll}
\mathcal{C}[\![\,\textsf{ret}\ e\,]\!]_k & = & k(e) & \\
\mathcal{C}[\![\,\textsf{val}\ x = s_0;\ s\,]\!]_k & = & \textsf{cnt}\ k_0(x)\,\{\,\mathcal{C}[\![\,s\,]\!]_k\,\};\ \mathcal{C}[\![\,s_0\,]\!]_{k_0} & \text{where } k_0 \text{ fresh} \\
\mathcal{C}[\![\,f(e)\,]\!]_k & = & f(e \mathbin{!} k) & \\
\mathcal{C}[\![\,\textsf{def}\ f(x)\,\{\,s_0\,\};\ s\,]\!]_k & = & \textsf{let}\ f(x \mathbin{!} k_0)\,\{\,\mathcal{C}[\![\,s_0\,]\!]_{k_0}\,\};\ \mathcal{C}[\![\,s\,]\!]_k & \text{where } k_0 \text{ fresh} \\
\mathcal{C}[\![\,\textsf{def}\ f(x)\,\{\,s_0\,\};\ s\,]\!]_\bullet & = & \textsf{let}\ f(x \mathbin{!} k_0)\,\{\,\mathcal{C}[\![\,s_0\,]\!]_{k_0}\,\};\ \mathcal{C}[\![\,s\,]\!]_\bullet & \text{where } k_0 \text{ fresh} \\
\mathcal{C}[\![\,\textsf{process}\ k_0(x)\,\{\,s_0\,\};\ s\,]\!]_k & = & \textsf{cnt}\ k_0(x)\,\{\,\mathcal{C}[\![\,s_0\,]\!]_\bullet\,\};\ \mathcal{C}[\![\,s\,]\!]_k & \\
\mathcal{C}[\![\,\textsf{process}\ k_0(x)\,\{\,s_0\,\};\ s\,]\!]_\bullet & = & \textsf{cnt}\ k_0(x)\,\{\,\mathcal{C}[\![\,s_0\,]\!]_\bullet\,\};\ \mathcal{C}[\![\,s\,]\!]_\bullet & \\
\mathcal{C}[\![\,\textsf{suspend}\,\{\,k_0 \Rightarrow s\,\}\,]\!]_k & = & \mathcal{C}[\![\,s\,]\!]_\bullet \quad \text{with } k_0 := k & \\
\mathcal{C}[\![\,\textsf{run}(k_0)\,\{\,s\,\}\,]\!]_\bullet & = & \mathcal{C}[\![\,s\,]\!]_{k_0} & \\
\mathcal{C}[\![\,\textsf{exit}\ e\,]\!]_\bullet & = & \textsf{exit}\ e &
\end{array}
$$

Figure 3.7: Translation to continuation-passing style.

Translation of Values $\qquad$ $\vdash_{\text{val}} \mathsf{V} : \tau \longrightarrow \vdash_{\text{val}} \mathsf{V} : \tau$

$$
\begin{array}{lcll}
\mathcal{C}_V[\![\,\{\,\mathsf{E},\ (x : \tau) \Rightarrow s\,\}\,]\!] & = & \{\,\mathcal{C}_E[\![\,\mathsf{E}\,]\!],\ (x : \tau \mathbin{!} k : \neg\tau_0) \Rightarrow \mathcal{C}[\![\,s\,]\!]_k\,\} & \text{where } k \text{ fresh} \\
\mathcal{C}_V[\![\,\mathsf{K}\,]\!] & = & \mathcal{C}_K[\![\,\mathsf{K}\,]\!] & \\
\mathcal{C}_V[\![\,19\,]\!] & = & 19 &
\end{array}
$$

Translation of Environments $\qquad$ $\mathsf{E} \vdash_{\text{env}} \Gamma \longrightarrow \mathsf{E} \vdash_{\text{env}} \Gamma$

$$
\begin{array}{lcl}
\mathcal{C}_E[\![\,\bullet\,]\!] & = & \bullet \\
\mathcal{C}_E[\![\,\mathsf{E},\ x \mapsto \mathsf{V}\,]\!] & = & \mathcal{C}_E[\![\,\mathsf{E}\,]\!],\ x \mapsto \mathcal{C}_V[\![\,\mathsf{V}\,]\!]
\end{array}
$$

Translation of Stacks $\qquad$ $\tau \vdash_{\text{stk}} \mathsf{K} \longrightarrow \vdash_{\text{val}} \mathsf{V} : \neg\tau$

$$
\begin{array}{lcll}
\mathcal{C}_K[\![\,\{\,\mathsf{E},\ (x : \tau) \Rightarrow s\,\}\,]\!] & = & \{\,\mathcal{C}_E[\![\,\mathsf{E}\,]\!],\ (x : \tau) \Rightarrow \mathcal{C}[\![\,s\,]\!]_\bullet\,\} & \\
\mathcal{C}_K[\![\,\{\,\mathsf{E},\ (x : \tau) \Rightarrow s\,\} :: \mathsf{K}\,]\!] & = & \{\,\mathcal{C}_E[\![\,\mathsf{E}\,]\!],\ k \mapsto \mathcal{C}_K[\![\,\mathsf{K}\,]\!],\ (x : \tau) \Rightarrow \mathcal{C}[\![\,s\,]\!]_k\,\} & \text{where } k \text{ fresh}
\end{array}
$$

Translation of Configurations $\qquad$ $\vdash \mathsf{M} \longrightarrow \vdash \mathsf{M}$

$$
\begin{array}{lcll}
\mathcal{C}_M[\![\,\langle \mathsf{E} \parallel s \rangle\,]\!] & = & \langle \mathcal{C}_E[\![\,\mathsf{E}\,]\!] \parallel \mathcal{C}[\![\,s\,]\!]_\bullet \rangle & \\
\mathcal{C}_M[\![\,\langle \mathsf{E} \parallel s \parallel \mathsf{K} \rangle\,]\!] & = & \langle \mathcal{C}_E[\![\,\mathsf{E}\,]\!],\ k \mapsto \mathcal{C}_K[\![\,\mathsf{K}\,]\!] \parallel \mathcal{C}[\![\,s\,]\!]_k \rangle & \text{where } k \text{ fresh}
\end{array}
$$

Figure 3.8: Translation of machine to continuation-passing style.

The translation of defining a new process also has these two cases, i.e., the translation of the remaining statement is the same as for function definition. The definition of a new process is translated to a definition of a continuation with the same name, and its body is translated with no continuation, as it is non-returning.

In the translation of **suspend**, the body is translated with no continuation, but the continuation variable bound by **suspend** is replaced with the input continuation of the translation, as the latter is the current continuation. Running a statement s in a process bound to k_0 is translated by translating s with continuation input k_0. The **exit** statement is translated to its counterpart in λ_{C}.

CPS Translation of Machine

The extension of the CPS translation to the abstract machine is shown in Figure 3.8. As there are two different modes of machine configurations now, there are two cases in the translation. The case for delimited configurations is still the same as in the pure case, using the binding

for the translated stack as input continuation for the statement. For undelimited execution, where there is no stack, the statement is translated with no input continuation in the translated environment.

Environments are still translated by pointwise translation of the bound values. For values, there is a third case now for the translation of stacks, since there can now be continuation bindings in the environment. For stacks, instead of the case for done frames, there is a case for general underflow frames now. These are essentially translated in the same way as definitions of a new process but, similar to the translation of closures, put together with the translated environment. The translation of a frame on top of a remaining stack stays the same.

3.2.2 From Continuation-Passing Style Back to Direct Style

Let us now consider the opposite direction. We first describe how the DS translation behaves on terms, and then extend it to the abstract machine. Again, expressions are translated trivially. The DS translation is bottom-up, and we give it in the style of inference rules. This way, we can easily state the side conditions we often have for different cases.

DS Translation of Terms

Figure 3.9 defines the translation on terms. Where the CPS translation has an additional input, the DS translation has an additional output, as we have already seen for the pure fragments. If the translated term is delimited, as is always the case for the pure translation, there is an additional variable as output standing for the stack the translated term returns to. This variable is then removed from the environment. If the translated term is non-returning, then there is no variable as output, again indicated by $\bullet$, and the environment stays the same. Remember that we refer to this additional output as continuation output. The translation is bottom-up in that it first recursively translates the subterms. In contrast to the translation for the pure fragments, we then decide what to do depending on the continuation outputs of the translated subterms. In the non-pure cases, we have to use control operators, in particular the constructs **suspend** and **run** in λ_{D}, which allow us to flexibly adapt the stack where necessary. With the bottom-up approach, we try to recognize where to insert these control operators in order to minimize their use.

There are three base cases without subterms. The **exit**-term is translated to its counterpart in λ_{D} and does not return. For the translation of a function application $f(e \mid k)$ we use the function resetIfFree to first check whether the continuation parameter k appears free in $f(e)$ (which by typing means $e = k$).

$$\mathsf{resetIfFree}(k)(s) = \textbf{IF } k \in \mathsf{freeVars}(s) \textbf{ THEN } \textbf{run}(k)\{ s \} \rightsquigarrow \bullet \textbf{ ELSE } s \rightsquigarrow k$$

If not, the translation has k as its output continuation, as this represents the current continuation, i.e., the stack the translated term returns to. This is the pure case. Otherwise, we need a control operator, since in a pure statement, the current continuation cannot be the argument of a function. Specifically, we reset $f(e)$ to run in the process bound to k, that is, $\textbf{run}(k)\{ f(e) \}$. Note that this statement then has no output continuation as it does not return. In the case of calling a continuation $k(e)$, typing makes it impossible that k is free in e (which would mean $e = k$) since $\neg\tau \neq \tau$. Thus, the translation is still the same as for the pure languages; it simply returns e with continuation output k. Again, this is the stack that the translated term returns to. Without typing, we could again reset with k upon free occurrence.

For function definitions, there are now quite a few cases to consider, depending on the continuation outputs of translating the subterms. In any case, the result is again a function definition in λ_{D}, but we have to insert control operators in some cases.

Rules DLet1, DLet2, and DLet3 define a helper function that determines what happens with the actual function definition, as this is independent of the remaining term. In abuse of

$\boxed{\Gamma_1,\, k : \neg\tau,\, \Gamma_2 \vdash t \longrightarrow (\Gamma_1,\, \Gamma_2 \vdash s : \tau \rightsquigarrow k)}$ $\qquad$ $\boxed{\Gamma \vdash t \longrightarrow (\Gamma \vdash s : \# \rightsquigarrow \bullet)}$

Translation of Terms

$$\frac{}{\mathcal{D}[\![\, f(e \mid k) \,]\!] = \mathsf{resetIfFree}(k)(f(e))}\ [\textsc{DApp}] \qquad \frac{}{\mathcal{D}[\![\, k(e) \,]\!] = \mathbf{ret}\ e \rightsquigarrow k}\ [\textsc{DJmp}]$$

$$\frac{\mathcal{D}[\![\, t_0 \,]\!] = s_0 \rightsquigarrow k_0}{\mathcal{D}[\![\, \mathbf{let}\, f(x \mid k_0)\, \{\, t_0 \,\} \,]\!] = \mathbf{def}\, f(x)\, \{\, s_0 \,\}}\ [\textsc{DLet1}]$$

$$\frac{\mathcal{D}[\![\, t_0 \,]\!] = s_0 \rightsquigarrow \bullet}{\mathcal{D}[\![\, \mathbf{let}\, f(x \mid k_0)\, \{\, t_0 \,\} \,]\!] = \mathbf{def}\, f(x)\, \{\, \mathbf{suspend}\, \{\, k_0 \Rightarrow s_0 \,\} \,\}}\ [\textsc{DLet2}]$$

$$\frac{\mathcal{D}[\![\, t_0 \,]\!] = s_0 \rightsquigarrow v_0 \qquad v_0 \neq k_0}{\mathcal{D}[\![\, \mathbf{let}\, f(x \mid k_0)\, \{\, t_0 \,\} \,]\!] = \mathbf{def}\, f(x)\, \{\, \mathbf{suspend}\, \{\, k_0 \Rightarrow \mathbf{run}(v_0)\, \{\, s_0 \,\} \,\} \,\}}\ [\textsc{DLet3}]$$

$$\frac{\mathcal{D}[\![\, t \,]\!] = s \rightsquigarrow \bullet}{\mathcal{D}[\![\, \mathbf{let}\, f(x \mid k_0)\, \{\, t_0 \,\};\ t \,]\!] = \mathcal{D}[\![\, \mathbf{let}\, f(x \mid k_0)\, \{\, t_0 \,\} \,]\!];\ s \rightsquigarrow \bullet}\ [\textsc{DLetN}]$$

$$\frac{\mathcal{D}[\![\, t \,]\!] = s \rightsquigarrow k}{\mathcal{D}[\![\, \mathbf{let}\, f(x \mid k_0)\, \{\, t_0 \,\};\ t \,]\!] = \mathsf{resetIfFree}(k)(\mathcal{D}[\![\, \mathbf{let}\, f(x \mid k_0)\, \{\, t_0 \,\} \,]\!];\ s)}\ [\textsc{DLetR}]$$

$$\frac{\mathcal{D}[\![\, t_0 \,]\!] = s_0 \rightsquigarrow \bullet}{\mathcal{D}[\![\, \mathbf{cnt}\, k_0(x)\, \{\, t_0 \,\} \,]\!] = \mathbf{process}\, k_0(x)\, \{\, s_0 \,\}}\ [\textsc{DCnt1}]$$

$$\frac{\mathcal{D}[\![\, t_0 \,]\!] = s_0 \rightsquigarrow x}{\mathcal{D}[\![\, \mathbf{cnt}\, k_0(x)\, \{\, t_0 \,\} \,]\!] = \mathbf{process}\, k_0(x)\, \{\, \mathbf{run}(x)\, \{\, s_0 \,\} \,\}}\ [\textsc{DCnt2}]$$

$$\frac{\mathcal{D}[\![\, t_0 \,]\!] = s_0 \rightsquigarrow v_0 \qquad v_0 \neq x \qquad \mathcal{D}[\![\, t \,]\!] = s \rightsquigarrow \bullet}{\mathcal{D}[\![\, \mathbf{cnt}\, k_0(x)\, \{\, t_0 \,\};\ t \,]\!] = \mathsf{resetIfFree}(v_0)(\mathbf{val}\, x = \mathbf{suspend}\, \{\, k_0 \Rightarrow s \,\};\ s_0)}\ [\textsc{DCntNV}]$$

$$\frac{\mathcal{D}[\![\, t_0 \,]\!]\ \mathbf{else} \qquad \mathcal{D}[\![\, t \,]\!] = s \rightsquigarrow \bullet}{\mathcal{D}[\![\, \mathbf{cnt}\, k_0(x)\, \{\, t_0 \,\};\ t \,]\!] = \mathcal{D}[\![\, \mathbf{cnt}\, k_0(x)\, \{\, t_0 \,\} \,]\!];\ s \rightsquigarrow \bullet}\ [\textsc{DCntN}]$$

$$\frac{\mathcal{D}[\![\, t_0 \,]\!] = s_0 \rightsquigarrow v_0 \qquad v_0 \neq x \qquad \mathcal{D}[\![\, t \,]\!] = s \rightsquigarrow k \qquad k \neq k_0}{\mathcal{D}[\![\, \mathbf{cnt}\, k_0(x)\, \{\, t_0 \,\};\ t \,]\!] = \mathsf{resetIfFree}(v_0)(\mathbf{val}\, x = \mathbf{suspend}\, \{\, k_0 \Rightarrow \mathbf{run}(k)\, \{\, s \,\} \,\};\ s_0)}\ [\textsc{DCntROV}]$$

$$\frac{\mathcal{D}[\![\, t_0 \,]\!]\ \mathbf{else} \qquad \mathcal{D}[\![\, t \,]\!] = s \rightsquigarrow k \qquad k \neq k_0}{\mathcal{D}[\![\, \mathbf{cnt}\, k_0(x)\, \{\, t_0 \,\};\ t \,]\!] = \mathsf{resetIfFree}(k)(\mathcal{D}[\![\, \mathbf{cnt}\, k_0(x)\, \{\, t_0 \,\} \,]\!];\ s)}\ [\textsc{DCntRO}]$$

$$\frac{\mathcal{D}[\![\, t_0 \,]\!] = s_0 \rightsquigarrow v_0 \qquad v_0 \neq x \qquad \mathcal{D}[\![\, t \,]\!] = s \rightsquigarrow k_0}{\mathcal{D}[\![\, \mathbf{cnt}\, k_0(x)\, \{\, t_0 \,\};\ t \,]\!] = \mathsf{resetIfFree}(v_0)(\mathbf{val}\, x = s;\ s_0)}\ [\textsc{DCntRTV}]$$

$$\frac{\mathcal{D}[\![\, t_0 \,]\!]\ \mathbf{else} \qquad \mathcal{D}[\![\, t \,]\!] = s \rightsquigarrow k_0}{\mathcal{D}[\![\, \mathbf{cnt}\, k_0(x)\, \{\, t_0 \,\};\ t \,]\!] = \mathcal{D}[\![\, \mathbf{cnt}\, k_0(x)\, \{\, t_0 \,\} \,]\!];\ \mathbf{run}(k_0)\, \{\, s \,\} \rightsquigarrow \bullet}\ [\textsc{DCntRT}]$$

$$\frac{}{\mathcal{D}[\![\, \mathbf{exit}\, e \,]\!] = \mathbf{exit}\ e \rightsquigarrow \bullet}\ [\textsc{DExit}]$$

Figure 3.9: Translation back to direct style.

notation, we use the same name for this helper function as for the translation function itself. The pure case is in DLET1 where the translated body of the function definition returns to the continuation parameter k_0, which is the current continuation. Note that this means in particular that k_0 is not free in the translated body, and we can just drop the continuation parameter. If the translated body does not return, as in DLET2, the current continuation represented by the function parameter must have been used in a non-pure way, so we have to insert a **suspend** to capture the current stack. If the stack v_0 the translated body returns to is different from the current continuation (DLET3), we also have to capture the latter, but we additionally have to insert a **run**(v_0) to reset the translated body with the proper stack.

Rules DLETN and DLETR then determine how to translate the whole term for function definitions based on the result of translating the remaining term. If the translation of the remaining term does not return, then neither does the whole statement. If it does return to the stack k, we again have to check whether k occurs free in the translated term and reset with k if it does. We are in the pure case if k does not occur free.

For the definition of a continuation, there are even more cases to consider. Depending on the continuation output of the translation of the body s_0, we either obtain a sequencing statement or a definition of a new process.

The latter is the case if s_0 does not return or if it returns to the parameter x of the continuation. In this case, the process just defined is non-pure and hence cannot represent the current continuation. We have again defined a helper function for these two cases, DCNT1 and DCNT2, which only acts on the actual definition of the continuation. In the case where the translated body returns to x, we insert a **run**(x). In either of these two cases, we then distinguish three cases for the translation of the remaining term t to a statement s. The non-returning case DCNTN and the case DCNTRO where s returns to a stack k different from the process k_0 just defined, are similar to the corresponding cases of function definition, i.e., in the second case we reset with k if it occurs free. In the third case DCNTRT where s returns to k_0, we insert a **run**(k_0), as the process just defined is not the current continuation.

The case left to consider is when the translated body s_0 returns to a stack v_0 different from the parameter x. In this case, we translate to a sequencing statement that binds x and has s_0 as its second statement. Depending on the result s of translating the remaining term t, we have to insert control operators. The pure case is when s returns to k_0 as in DCNTRTV. If s does not return, as in DCNTNV, we have to insert a **suspend** to capture the current stack and bind it to k_0. Finally, in DCNTROV where s returns to a stack k different from k_0, we capture the current stack to k_0, and then reset with k. In any case, we have to check whether v_0 appears free, in which case we have to reset the whole statement with v_0.

DS Translation of Machine

Figure 3.10 shows how the DS translation is extended to the abstract machine. For machine configurations, there are two cases now, depending on the continuation output of the translation s of the term in focus. If s is non-returning, it is executed in an undelimited configuration with the translated environment. If s returns to k, the value bound to k is removed from the translated environment, and its translation is used as stack for the delimited machine configuration, just as in the case of the pure translations. In contrast to the pure case, there can now be an arbitrary number of continuation bindings in the environment E. This also means that in the pointwise translation of environments, the bound values can now also be stacks. We write $\mathsf{E}.\mathsf{rm}(k)$ for removing the binding to k from E.

For values, we still distinguish between returning and non-returning ones. Those that return are integers, which are still translated trivially, and function closures, which are again translated in essentially the same way as function definitions put together with the translated environment. The pure case is rule DFCLO1 where the translated body returns to the continuation variable bound by the function. The non-returning values, i.e., continuation closures, are translated to stacks. The translation of these is similar to the translation of continuation definitions, in

Translation of Values $\boxed{\vdash_{\text{val}} \mathsf{V} : \tau \longrightarrow \vdash_{\text{val}} \mathsf{V} : \tau}$

$$\frac{\mathcal{D}[\![\, t_0 \,]\!] = s_0 \rightsquigarrow k_0}{\mathcal{D}_V[\![\, \{\, \mathsf{E},\, (x : \tau \mid k_0 : \neg\, \tau_0) \Rightarrow t_0 \,\} \,]\!] = \{\, \mathcal{D}_E[\![\, \mathsf{E} \,]\!],\, (x : \tau) \Rightarrow s_0 \,\}} \;[\textsc{DFCLo1}]$$

$$\frac{\mathcal{D}[\![\, t_0 \,]\!] = s_0 \rightsquigarrow \bullet}{\mathcal{D}_V[\![\, \{\, \mathsf{E},\, (x : \tau \mid k_0 : \neg\, \tau_0) \Rightarrow t_0 \,\} \,]\!] = \{\, \mathcal{D}_E[\![\, \mathsf{E} \,]\!],\, (x : \tau) \Rightarrow \mathbf{suspend}\, \{\, k_0 \Rightarrow s_0 \,\} \,\}} \;[\textsc{DFCLo2}]$$

$$\frac{\mathcal{D}[\![\, t_0 \,]\!] = s_0 \rightsquigarrow v_0 \qquad v_0 \neq k_0}{\mathcal{D}_V[\![\, \{\, \mathsf{E},\, (x : \tau \mid k_0 : \neg\, \tau_0) \Rightarrow t_0 \,\} \,]\!] = \{\, \mathcal{D}_E[\![\, \mathsf{E} \,]\!],\, (x : \tau) \Rightarrow \mathbf{suspend}\, \{\, k_0 \Rightarrow \mathbf{run}(v_0)\, \{\, s_0 \,\} \,\} \,\}} \;[\textsc{DFCLo3}]$$

$$\frac{}{\mathcal{D}_V[\![\, \{\, \mathsf{E},\, (x : \tau) \Rightarrow t_0 \,\} \,]\!] = \mathcal{D}_K[\![\, \{\, \mathsf{E},\, (x : \tau) \Rightarrow t_0 \,\} \,]\!]} \;[\textsc{DCCLo}] \qquad \frac{}{\mathcal{D}_V[\![\, 19 \,]\!] = 19} \;[\textsc{DVInt}]$$

Translation of Environments $\boxed{\mathsf{E} \vdash_{\text{env}} \Gamma \longrightarrow \mathsf{E} \vdash_{\text{env}} \Gamma}$

$$\frac{}{\mathcal{D}_E[\![\, \bullet \,]\!] = \bullet} \;[\textsc{DEnv1}] \qquad \frac{}{\mathcal{D}_E[\![\, \mathsf{E},\, x \mapsto \mathsf{V} \,]\!] = \mathcal{D}_E[\![\, \mathsf{E} \,]\!],\, x \mapsto \mathcal{D}_V[\![\, \mathsf{V} \,]\!]} \;[\textsc{DEnv2}]$$

Translation of Continuation Closures $\boxed{\vdash_{\text{val}} \mathsf{V} : \neg\, \tau \longrightarrow \tau \vdash_{\text{stk}} \mathsf{K}}$

$$\frac{\mathcal{D}[\![\, t_0 \,]\!] = s_0 \rightsquigarrow \bullet}{\mathcal{D}_K[\![\, \{\, \mathsf{E},\, (x : \tau) \Rightarrow t_0 \,\} \,]\!] = \{\, \mathcal{D}_E[\![\, \mathsf{E} \,]\!],\, (x : \tau) \Rightarrow s_0 \,\}} \;[\textsc{DCCLo1}]$$

$$\frac{\mathcal{D}[\![\, t_0 \,]\!] = s_0 \rightsquigarrow x}{\mathcal{D}_K[\![\, \{\, \mathsf{E},\, (x : \tau) \Rightarrow t_0 \,\} \,]\!] = \{\, \mathcal{D}_E[\![\, \mathsf{E} \,]\!],\, (x : \tau) \Rightarrow \mathbf{run}(x)\, \{\, s_0 \,\} \,\}} \;[\textsc{DCCLo2}]$$

$$\frac{\mathcal{D}[\![\, t_0 \,]\!] = s_0 \rightsquigarrow k \qquad k \neq x}{\mathcal{D}_K[\![\, \{\, \mathsf{E},\, (x : \tau) \Rightarrow t_0 \,\} \,]\!] = \{\, \mathcal{D}_E[\![\, \mathsf{E} \,]\!].\mathsf{rm}(k),\, (x : \tau) \Rightarrow s_0 \,\} :: \mathcal{D}_K[\![\, \mathsf{E}(k) \,]\!]} \;[\textsc{DCCLo3}]$$

Translation of Configurations $\boxed{\vdash \mathsf{M} \longrightarrow \vdash \mathsf{M}}$

$$\frac{\mathcal{D}[\![\, t \,]\!] = s \rightsquigarrow \bullet}{\mathcal{D}_M[\![\, \langle \mathsf{E} \parallel t \rangle \,]\!] = \langle \mathcal{D}_E[\![\, \mathsf{E} \,]\!] \parallel s \rangle} \;[\textsc{DMch1}]$$

$$\frac{\mathcal{D}[\![\, t \,]\!] = s \rightsquigarrow k}{\mathcal{D}_M[\![\, \langle \mathsf{E} \parallel t \rangle \,]\!] = \langle \mathcal{D}_E[\![\, \mathsf{E} \,]\!].\mathsf{rm}(k) \parallel s \parallel \mathcal{D}_K[\![\, \mathsf{E}(k) \,]\!] \rangle} \;[\textsc{DMch2}]$$

Figure 3.10: Translation of machine back to direct style.

particular in the cases where the translated body does not return or returns to the parameter of the continuation closure. The case where the translated body returns to a stack k different from the parameter (DCCLO3) is more akin to the translation of a delimited machine configuration in that the binding for k is removed from the translated environment, and the translation of the value bound to k is used as the remaining stack on top of which the frame is put. This is as in the pure case.

3.2.3 Properties

With the full languages and translations in place, we want to show that they are still well-behaved. To this end, we consider again the meta-theoretical properties we have shown for the restriction to the pure fragments in Section 2.3.4.

Typeability Preservation

Typeability preservation still holds as before. This has been proven in Idris 2 by the intrinsic typing of the languages and the various translation functions on them.

Theorem 26 (Typeability preservation)**.**
All translations take typing derivations to typing derivations as stated above their definitions.

Semantics Preservation

Next, we consider semantics preservation. We do not have a step-by-step correspondence anymore, but since we use a small-step abstract machine semantics, we can still be very precise about how semantics is preserved, including upper and lower bounds for the number of steps the translated machine takes. In the following, we again identify machine configurations and frames which can be identified by weakening, contraction, or reordering of environments. Moreover, we consider equality always modulo α-equivalence. The proofs are mostly straightforward and are given in Müller et al. (2023).

We start with the CPS translation. For every step the DS machine takes, the CPS machine either takes no step or one step.

Theorem 27 (Simulation of $\mathcal{C}_M$)**.**
If $\vdash M$ *and* $M \to M'$*, then* $\mathcal{C}_M[\![M]\!] \to^? \mathcal{C}_M[\![M']\!]$*, where* $\to^?$ *means* 0 *or* 1 *machine step.*

The reason why the correspondence is not 1-to-1 is that there are steps for the control operators **suspend** and **run** in the DS machine, but the CPS translation already includes those steps. For example, in the translation of $\mathbf{suspend}\,\{\, k \Rightarrow s \,\}$, we simply translate the body s and replace k by the variable in the environment to which the current continuation is bound upon translating the machine state. In the DS version, this capturing of the stack into the environment is done in a machine step.

As a corollary, we obtain that the CPS machine takes *at most* as many steps as the DS machine.

Corollary 28 (Operational Reduction (CPS))**.**
If $\vdash M$ *and* $M \to^n M'$*, then* $\mathcal{C}_M[\![M]\!] \to^m \mathcal{C}_M[\![M']\!]$ *where* $m \leq n$*.*

Proof. By induction on n using Theorem 27. □

In particular, a closed term that evaluates to a result in DS evaluates to the same result in CPS in at most as many steps.

Corollary 29 (Evaluation (CPS))**.**
If $\vdash s : \tau$ *and* $\langle \bullet \parallel s \parallel \mathsf{done} \rangle \to^n \langle E \parallel \mathbf{exit}\, e \rangle$
then $\langle k \mapsto \mathsf{done} \parallel \mathcal{C}[\![s]\!]_k \rangle \to^m \langle \mathcal{C}_E[\![E]\!] \parallel \mathbf{exit}\, e \rangle$ *where* k *is fresh and* $m \leq n$*.*

Let us now look at the DS translation. The CPS machine needs fewer steps, but we can give an upper bound for the DS machine: for every step the CPS machine takes, the DS machine takes at least one and at most four steps. The reason is that we insert up to three control operators for one construct during translation, each of which needs its own step in the DS machine.

Theorem 30 (Simulation of $\mathcal{D}_M$)**.**
If $\vdash M$ *and* $M \to M'$*, then* $\mathcal{D}_M[\![M]\!] \to^{[1-4]} \mathcal{D}_M[\![M']\!]$*, where* $\to^{[1-4]}$ *means* 1*,* 2*,* 3*, or* 4 *machine steps.*

We obtain corollaries similar to the CPS translation above.

Corollary 31 (Operational Reduction (DS))**.**
If $\vdash M$ *and* $M \to^n M'$*, then* $\mathcal{D}_M[\![M]\!] \to^m \mathcal{D}_M[\![M']\!]$ *where* $m \leq 4n$*.*

Proof. By induction on n using Theorem 30. □

Corollary 32 (Evaluation (DS))**.**
If $\vdash t$ *with* $\mathcal{D}[\![t]\!] = s \rightsquigarrow k$ *and* $\langle k \mapsto \mathsf{done} \,\|\, t \rangle \to^n \langle E \,\|\, \mathbf{exit}\, e \rangle$
then $\langle \bullet \,\|\, s \,\|\, \mathsf{done} \rangle \to^m \langle \mathcal{D}_E[\![E]\!] \,\|\, \mathbf{exit}\, e \rangle$ *where* $m \leq 4n$*.*

Round Trips

Let us look at the behavior of the compositions of the two translations. The translations are not full syntactic inverses anymore, but after the round trip starting in λ_{C} we still obtain the syntactically same term.

Theorem 33 ($\mathcal{D}$ is right inverse of $\mathcal{C}$)**.**
If $\Gamma \vdash t$ *and* $\mathcal{D}[\![t]\!] = s \rightsquigarrow k$*, then* $\mathcal{C}[\![s]\!]_k = t$*.*
If $\Gamma \vdash t$ *and* $\mathcal{D}[\![t]\!] = s \rightsquigarrow \bullet$*, then* $\mathcal{C}[\![s]\!]_\bullet = t$*.*

We can also still extend this property to the abstract machines.

Theorem 34 (Right inverse for machines)**.**
If $\vdash M$*, then* $\mathcal{C}_M[\![\mathcal{D}_M[\![M]\!]]\!] = M$*.*
If $\vdash_{val} V : \tau$*, then* $\mathcal{C}_V[\![\mathcal{D}_V[\![V]\!]]\!] = V$*.*
If $E \vdash_{env} \Gamma$*, then* $\mathcal{C}_E[\![\mathcal{D}_E[\![E]\!]]\!] = E$*.*
If $\vdash_{val} V : \neg\tau$*, then* $\mathcal{C}_K[\![\mathcal{D}_K[\![V]\!]]\!] = V$*.*

This means, in particular, that the CPS translation is surjective and the DS translation is injective. As we have seen at the beginning of this chapter, the converse is not true, since in general, there are multiple DS statements, i.e., with different combinations of control operators, that are mapped to the same CPS term. Therefore, we do not in general insert the same control operators during the DS translation that are removed when CPS-translating. This also means that there is some choice of which combination of control operators to insert in the DS translation. The choices here are made in such a way that we can easily give the tight bounds for the number of machine steps when evaluating.

Moreover, as the round trip also affects the environment and the stack, we cannot expect that the original machine reduces to the one after the round trip. But semantics preservation at least gives us that both machine configurations eventually evaluate to the same result (if they terminate) with the given upper bound for the number of steps.

Syntax of λ_{D}

Statements	s	::=	$\ldots \mid \textsf{if}\ e\ \textsf{then}\ s\ \textsf{else}\ s$	conditionals

Syntax of λ_{C}

Terms	t	::=	$\ldots \mid \textsf{if}\ e\ \textsf{then}\ t\ \textsf{else}\ t$	conditionals

Syntax of Expressions

Expressions	e	::=	$\ldots \mid \mathsf{true} \mid \mathsf{false}$	booleans

Syntax of Types

Types	τ	::=	$\ldots \mid \mathsf{Bool}$	base type

Figure 3.11: Syntax for the extension of the languages with conditionals.

Semantics Reflection

From the round-trip result for λ_{C}, we still get a corollary about reflection of machine steps for the DS translation. If there is a step in the DS machine between translated machine configurations, then there must be a step in the CPS machine between the original states. For the CPS translation, we have no such result.

Corollary 35 ($\mathcal{D}_M$ is step-reflecting)**.**
Given $\vdash M$ *and* $\vdash M'$*, then* $\mathcal{D}_M[\![\, M \,]\!] \to \mathcal{D}_M[\![\, M' \,]\!]$ *implies* $M \to M'$*.*

3.3 Extension with Conditionals

The languages considered so far are rather minimalistic. To make them more practically usable, we now show how they can be extended with conditional statements. Consider a DS translation of an **if** statement,

$$\mathcal{D}[\![\, \textsf{if}\ e\ \textsf{then}\ t_1\ \textsf{else}\ t_2 \,]\!] = \textsf{if}\ e\ \textsf{then} \ldots s_1 \ldots \textsf{else} \ldots s_2 \ldots \rightsquigarrow ?$$

where $\mathcal{D}[\![\, t_1 \,]\!] = s_1 \rightsquigarrow k_1$ and $\mathcal{D}[\![\, t_2 \,]\!] = s_2 \rightsquigarrow k_2$. Both branches have to return to the same stack as the whole statement. Thus, if $k_1 = k_2$, we do not need any control operators, and the whole statement returns to this stack. This is always the case for the pure fragment. However, if $k_1 \neq k_2$, we have to insert control operators to adapt the stacks the two branches return to.

Biased Solution

An easy solution is to pick one branch and treat it as the preferred branch, adapting the other one. Let us pick the second branch as the default. The translation is

$$\mathcal{D}[\![\, \textsf{if}\ e\ \textsf{then}\ t_1\ \textsf{else}\ t_2 \,]\!] = \textsf{if}\ e\ \textsf{then suspend}\ \{\, k_2 \Rightarrow \textsf{run}(k_1)\ \{\, s_1 \,\}\,\}\ \textsf{else}\ s_2 \rightsquigarrow k_2$$

Figure 3.11 shows the straightforward syntax extensions for the two languages with conditionals. As expressions and types are extended identically, they are shown only once. The scrutinee of the conditionals is an expression, since all intermediate results have to be named. The typing and operational semantics are straightforward and can be found in Müller et al. (2023).

The CPS translation for conditionals is unsurprising, as either both branches are returning or both are non-returning. In the first case, we translate both branches with the input continuation of the whole statement; in the second case, we translate both branches without an input continuation. The CPS translation is shown in Figure 3.12.

Translation of Statements

$$\begin{array}{lcl} \mathcal{C}[\![\, \textsf{if}\ e\ \textsf{then}\ s_1\ \textsf{else}\ s_2 \,]\!]_k & = & \textsf{if}\ e\ \textsf{then}\ \mathcal{C}[\![\, s_1 \,]\!]_k\ \textsf{else}\ \mathcal{C}[\![\, s_2 \,]\!]_k \\ \mathcal{C}[\![\, \textsf{if}\ e\ \textsf{then}\ s_1\ \textsf{else}\ s_2 \,]\!]_\bullet & = & \textsf{if}\ e\ \textsf{then}\ \mathcal{C}[\![\, s_1 \,]\!]_\bullet\ \textsf{else}\ \mathcal{C}[\![\, s_2 \,]\!]_\bullet \end{array}$$

Figure 3.12: Translation of conditionals to continuation-passing style.

Translation of Terms

$$\frac{\mathcal{D}[\![\, t_1 \,]\!] = s_1 \rightsquigarrow k \qquad \mathcal{D}[\![\, t_2 \,]\!] = s_2 \rightsquigarrow k}{\mathcal{D}[\![\, \textsf{if}\ e\ \textsf{then}\ t_1\ \textsf{else}\ t_2 \,]\!] = \textsf{if}\ e\ \textsf{then}\ s_1\ \textsf{else}\ s_2 \rightsquigarrow k}\ [\textsc{DIfRR}]$$

$$\frac{\mathcal{D}[\![\, t_1 \,]\!] = s_1 \rightsquigarrow \bullet \qquad \mathcal{D}[\![\, t_2 \,]\!] = s_2 \rightsquigarrow \bullet}{\mathcal{D}[\![\, \textsf{if}\ e\ \textsf{then}\ t_1\ \textsf{else}\ t_2 \,]\!] = \textsf{if}\ e\ \textsf{then}\ s_1\ \textsf{else}\ s_2 \rightsquigarrow \bullet}\ [\textsc{DIfNN}]$$

$$\frac{\mathcal{D}[\![\, t_1 \,]\!] = s_1 \rightsquigarrow \bullet \qquad \mathcal{D}[\![\, t_2 \,]\!] = s_2 \rightsquigarrow k}{\mathcal{D}[\![\, \textsf{if}\ e\ \textsf{then}\ t_1\ \textsf{else}\ t_2 \,]\!] = \textsf{if}\ e\ \textsf{then suspend}\ \{\, k \Rightarrow s_1 \,\}\ \textsf{else}\ s_2 \rightsquigarrow k}\ [\textsc{DIfRN}]$$

$$\frac{\mathcal{D}[\![\, t_1 \,]\!] = s_1 \rightsquigarrow k \qquad \mathcal{D}[\![\, t_2 \,]\!] = s_2 \rightsquigarrow \bullet}{\mathcal{D}[\![\, \textsf{if}\ e\ \textsf{then}\ t_1\ \textsf{else}\ t_2 \,]\!] = \textsf{if}\ e\ \textsf{then}\ s_1\ \textsf{else suspend}\ \{\, k \Rightarrow s_2 \,\} \rightsquigarrow k}\ [\textsc{DIfNR}]$$

$$\frac{\mathcal{D}[\![\, t_1 \,]\!] = s_1 \rightsquigarrow k_1 \qquad \mathcal{D}[\![\, t_2 \,]\!] = s_2 \rightsquigarrow k_2 \qquad k_1 \neq k_2}{\mathcal{D}[\![\, \textsf{if}\ e\ \textsf{then}\ t_1\ \textsf{else}\ t_2 \,]\!] = \textsf{if}\ e\ \textsf{then suspend}\ \{\, k_2 \Rightarrow \textsf{run}(k_1)\ \{\, s_1 \,\} \,\}\ \textsf{else}\ s_2 \rightsquigarrow k_2}\ [\textsc{DIfRRD}]$$

Figure 3.13: Translation of conditionals back to direct style.

As explained above, the interesting part of the extension is the DS translation. There are different cases to consider. If both branches return to the same stack or no stack, then we do not have to insert control operators, and the whole translated term returns to the common stack in the former case or is non-returning in the latter case. The more interesting cases are if the branches return to different stacks, or if one is returning and the other one is not. We have explained the first case above. In the second case, we have several options: a) we can again always adapt the branch that is not the preferred one, or b) we insert a **run** for the returning branch, resulting in a non-returning statement for the whole term, or c) we insert a **suspend** for the non-returning branch, resulting in a returning statement for the whole term. We choose option c) since we aim to obtain a returning statement, as would be the case for a pure term. In particular, in the case that the term surrounding the **if** is pure and the returning branch is too, the only place where control operators are inserted is in the non-returning branch. The DS translation is displayed in Figure 3.13.

The extension fits in seamlessly with our languages. All the properties we have given about our languages and translations still hold exactly as stated.

Other Alternative Designs

While the above solution works without any changes to other parts of the DS translation, it is a bit unsatisfactory with regard to avoiding the insertion of unnecessary control operators. For example, if the continuation output of the preferred branch is not the current continuation, then the whole **if** statement has to be adapted with control operators again. But in the case that the current continuation is the one that the non-preferred branch returns to, this is unnecessary.

We actually could have adapted the non-preferred branch instead, resulting in an overall **if** statement that already returns to the correct stack. To do so, however, we would have to know which continuation is the current continuation.

Passing the current continuation down One option is to pass this information downwards during the DS translation by having an additional continuation variable as input for the DS translation. This works well for the translation of terms. However, when extending the translation to the machine, things become more difficult. For intermediate machine states that are not closed, it is not always clear which continuation is the current one at the top level of the term in focus and for the bindings in the environment. A potential solution could be to annotate machine states with the current continuation, and adapt this annotation appropriately when taking a machine step. This approach thus appears to require some changes to the existing languages, and more investigation is needed to understand whether it is actually feasible.

Non-deterministic translation There is another option that avoids passing down the current continuation in the DS translation. We could instead try to defer the decision of which branch of the conditional to adapt. Later, when the current continuation is known, we choose the right one. To do so, we have to remember all possible translations for a conditional. Then the translation for the case when the branches return to different continuations, as in our initial example above, returns a list as follows

$$
\begin{aligned}
&\mathcal{D}[\![\, \textbf{if}\ e\ \textbf{then}\ t_1\ \textbf{else}\ t_2 \,]\!] \\
&= [\textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else suspend}\ \{\, k_1 \Rightarrow \textbf{run}(k_2)\ \{\, s_2 \,\}\,\} \rightsquigarrow k_1, \\
&\quad\ \textbf{if}\ e\ \textbf{then suspend}\ \{\, k_2 \Rightarrow \textbf{run}(k_1)\ \{\, s_1 \,\}\,\}\ \textbf{else}\ s_2 \rightsquigarrow k_2, \\
&\quad\ \textbf{if}\ e\ \textbf{then suspend}\ \{\, k? \Rightarrow \textbf{run}(k_1)\ \{\, s_1 \,\}\,\}\ \textbf{else suspend}\ \{\, k? \Rightarrow \textbf{run}(k_2)\ \{\, s_2 \,\}\,\} \rightsquigarrow k?]
\end{aligned}
$$

Here $k?$ has to be replaced by the current continuation when it is known, if that case is chosen. Now, if the translation of a subexpression yields a non-singleton list, we either pick the correct option and return that as a singleton list if the current continuation is known, or we do the further translation for all elements in the list and return the resulting list. This of course bears some danger of exponential blowup if conditionals are nested deeply without information about the current continuation in between.

At the top level, we then pick the right option in accordance with the current continuation. However, similar to the solution that passes the current continuation downwards, for non-closed machine states it is not always clear which is the current continuation at the top level, and hence which option to pick if the translation yields multiple possible results.

3.4 Related Work

There is a huge body of work on translations back to direct style, and in this section we discuss how they relate to what we have presented here. Figure 3.14 shows several properties and whether they are shown to hold (✓) by the different translations or not (✗), or are conjectured (❍).

3.4.1 Back to Direct Style I and II

A direct-style translation was first considered by Danvy (1992). They only work on a pure direct-style language, but extend their translation to a language with `call/cc` in a follow-up paper (Danvy and Lawall, 1992). Both papers use higher-order translations to reduce administrative redexes at translation time. To deal with this in proofs, they factor out the administrative reductions for both translations and obtain staged versions, which is more thoroughly investigated in Lawall and Danvy (1993). In the pure case, where the CPS language is restricted

	Typed	One-Pass First-Order Compositional	Small-Step Preserv.	Reduction Theory	Right Inverse DS	Detects Pure Terms
*Calculi **without** control operators*						
Danvy (1992)	✗	✗	✗	✗	✓	—
Sabry and Felleisen (1992)	✓	✗	✗	✓	✓	—
Flanagan et al. (1993)	✗	✗	✓	✗	✓	—
Sabry and Wadler (1997)	❍	✓	❍	✓	✓	—
Danvy and Pfenning (1995)	✗	✗	✗	✗	✗	—
Nielsen (2002)	✗	✓	✗	✗	✓	—
Hatcliff and Danvy (1994)	✓	✗	✗	✗	✗	—
*Calculi with **undelimited** control*						
Danvy and Lawall (1992)	✗	✗	✗	✗	✗	✓
Sabry and Felleisen (1993)	❍	✗	✗	✗	✗	✗
This work	✓	✓	✓	✗	✓	✓
*Calculi with **delimited** control*						
Kameyama and Hasegawa (2003)	✗	✗	✗	✗	✗	✗
Kameyama (2007)	✗	✗	✗	✗	✗	✗
Biernacki et al. (2020)	✗	✓	✗	✓	✓	✓
Biernacki et al. (2021)	✗	✓	✗	✓	✓	✓

Figure 3.14: Summary of related work.

to the image of the CPS translation via an attribute grammar, they obtain as a result that the DS translation is inverse to the CPS translation. For the language extended with `call/cc`, they use a counting analysis for continuation identifiers to decide where to insert a control operator. This analysis checks whether a continuation identifier occurs free in the scope of another continuation identifier. This is sufficient as they do not have an abortion operator. We instead rely on a bottom-up translation to also capture the case when no continuation identifier occurs. For these extended languages, the translations only satisfy the weaker property of forming a Galois connection with respect to a kind of normalization, which is to be expected, since the CPS translation removes superfluous control operators, which are not inserted again by the DS translation. The normalization consists of performing a round trip, so on terms after a round trip, the translations are inverse. As our DS translation is the right inverse of the CPS translation, we have a Galois connection in this sense, too. Note, however, that this is different from a Galois connection with respect to reduction (see below), and neither paper above considers reduction or equational theories at all.

3.4.2 Development of ANF

Another direct-style translation was studied by Sabry and Felleisen (1993). Their goal is to derive a set of rewrite rules for the computational λ_c-calculus (Moggi, 1989). They achieve this goal and present an equational theory that proves the same equations as the CPS counterpart. This eventually leads to the system of A-normal forms for pure terms. They also give an extension to a calculus with `call/cc` and an abortion operator, which together are equivalent to the control operator $\mathcal{C}$ of Felleisen et al. (1987). The translations used in either case are based on evaluation contexts, making them non-compositional. In the pure case, the DS translation works on a CPS language that is closed under β-η-reduction and thus a bit larger than just the image of the CPS translation. On the full languages, the translations are not inverse of each other, but terms after a round trip are connected to the original terms with respect to an equational theory. A round trip on the DS language brings terms into ANF. When restricting

the DS language to ANF, the CPS translation becomes injective, so a restriction of the CPS language to the image of the CPS translation makes the two translations inverses. Moreover, with the correct rewrite rules for ANF, both translations preserve reduction. An argument for ANF over CPS as a compiler intermediate language was made by Flanagan et al. (1993). There, the result is extended to abstract machines for the two languages, similar to our result for the pure fragment. A difference, however, is that the pure fragment of our language is not in ANF, as we allow for nesting of **val** statements. For their extension with control operators, they do not consider reduction, but only an equational theory. For this reason, their DS translation inserts a `call/cc` for every binding of a continuation and explicit calls to the continuation, even when unnecessary, in contrast to Danvy and Lawall's and this work. In the end, they state that their results also hold in a typed setting, but do not elaborate on that point much further.

3.4.3 Further Developments for Calculi Without Control Operators

In subsequent years, there were further developments in different directions for direct-style calculi without control operators. Hatcliff and Danvy (1994) consider Moggi's monadic meta language λ_{ml} (Moggi, 1991) in a typed setting and give a translation from an appropriate CPS language back to the meta language, forming an equational correspondence. As the source language is λ_{ml}, this forms a core for DS translations for different evaluation strategies, which can be obtained by giving translations of different calculi into λ_{ml} and back.

Sabry and Wadler (1997) improve upon previous results by considering a reduction theory for λ_c instead of merely equational correspondences. They show that their translations back and forth form a reflection with respect to reduction, i.e., a Galois-connection with the DS translation being the right inverse. This is a stronger result than that of Sabry and Felleisen (1993), who only have a correspondence with respect to an equational theory for the composition of the DS translation with the CPS translation (but already prove the remaining properties of a Galois-connection). The CPS translation used in the paper is not quite compositional, but this could be remedied easily by unfolding the translation a bit further. The authors moreover conjecture that their results hold in a typed setting, and also when reduction in arbitrary contexts is replaced by deterministic evaluation. Note that our work does not consider a reduction theory, but only evaluation with an abstract machine. We have shown that the translations on the pure fragment are two-sided inverses. Perhaps, since everything is named in our calculi, this is not too surprising. It would be interesting to also consider a reduction theory for our approach in general.

A somewhat different direct-style translation is given by Danvy and Pfenning (1995), who aim to mechanically prove claims in Danvy (1992) about continuation parameters. It relies on a stack of intermediate results maintained during the translation, promising more efficiency and simpler proofs. In particular, Nielsen (2002) builds on the same technique to give a very simple proof by structural induction that the first-order, one-pass and compositional CPS translation (Danvy and Nielsen, 2003) is the left inverse of the DS translation. This kind of DS translation is partly bottom-up in the sense that it uses the stack produced by subexpressions in some cases.

3.4.4 Back to Direct Style with Delimited Control

A bit later, some papers on direct-style translations for calculi with delimited control operators appeared. Kameyama and Hasegawa (2003) consider a calculus with one level of `shift` and `reset`, but as they deal with an equational theory, they insert a control operator for every occurrence of a continuation binding, similar to Sabry and Felleisen (1993). They show that their translations form an equational correspondence in an untyped setting. This result was later extended to the CPS hierarchy (Kameyama, 2007).

Very recently, Biernacki et al. (2020) improved upon this result in a similar way Sabry and Wadler (1997) improved upon earlier work (Sabry and Felleisen, 1993). They consider a reduction theory for an untyped calculus with `shift` and `reset`, and show that their translations back

and forth form a reflection with respect to reduction. A similar result for `shift0` and `reset0` has been shown (Biernacki et al., 2021), too. To do so, they use a CPS calculus with a mildly context-sensitive grammar tracking which continuation variable is the current continuation, and reduction rules that are specifically designed to keep the calculus closed under reduction. This makes it easy to recognize the combinators that are translated to control operators when going back to direct style. It would be interesting to see whether our approach can be extended to delimited control operators, too.

3.5 Conclusion and Future Work

We have presented a translation from continuation-passing style back to direct style in a typed setting with undelimited control operators. It is the right inverse of our CPS translation, and, as we have seen in Chapter 2, on the pure fragment even the two-sided inverse. Both translations are total and preserve the small-step evaluation of the abstract machine semantics of the languages with tight bounds. Moreover, both translations are first-order, one-pass, and compositional, facilitating simple proofs by structural induction. Our DS translation is bottom-up, in order to recognize where to insert control operators and minimize their use.

It is possible to choose between different equivalent possibilities of inserting control operators for the DS translation in some cases. We have designed our translation in such a way that it fits particularly nicely with the abstract machine. It would be interesting to investigate whether different choices yield better results in other regards, such as the behavior of round trips starting at the DS language. Passing down the current continuation might open up even more possibilities in that respect. Moreover, it would be useful to also consider reduction theories for our calculi, in particular, to be able to reason better about optimizations in the CPS language (Torrens et al., 2024).

We have also shown an extension of our basic calculi with conditionals in such a way that all theorems still hold as stated. However, in some cases the DS translation inserts more control operators than necessary. We have sketched two more sophisticated approaches that should be investigated further. Another interesting question is whether our translations can be extended to delimited control operators by iterating the translations to CPS and back to DS. In that regard, it might also be particularly interesting to investigate partial translations back to direct style, that is, leaving the non-pure parts of the program in CPS. This way, no control operators are needed anymore to translate back, while still having some of the advantages of executing programs in direct style. Yet another interesting question is the interaction of continuations with local mutable state. Having the possibility to execute the same continuation multiple times requires to carefully ensure the correct backtracking behavior for state. A standard way for CPS languages is to employ a kind of state-passing style, which stores the current state in the closure of the captured continuation (Schuster et al., 2022). This yields a pure program again that can be easily optimized. A translation back to direct style might then try to restore the use of native local mutable state.

4 Correspondence to Logical Calculi

Typed computational systems exhibit a direct correspondence to logical proof calculi via the Curry-Howard isomorphism. Formulae correspond to types and proofs of a formula correspond to terms of the corresponding type. In this chapter, we illustrate how our calculi from Chapters 2 and 3 relate to logical systems. The languages we have seen so far, in particular those in direct style, are based on natural deduction, which was introduced by Gentzen (1935a). The term language corresponding to natural deduction, in particular in the intuitionistic variant NJ, is the lambda calculus. For the classical variant NK, the lambda calculus must be augmented with control operators to account for classical reasoning, such as the law of excluded middle. Classical logic can be embedded into intuitionistic logic by a double-negation translation, which corresponds to a CPS translation.

In the same paper, Gentzen also introduced another calculus, the sequent calculus, for studying natural deduction and for proving his famous Hauptsatz (Gentzen, 1935b), the cut-elimination theorem. A satisfying term assignment for the sequent calculus has only been found relatively recently with the $\lambda\mu\tilde{\mu}$-calculus (Curien and Herbelin, 2000). The $\lambda\mu\tilde{\mu}$-calculus corresponds to the classical sequent calculus LK. But with a restriction similar to the one we have made for our CPS calculus λ_{C} to arrive at its pure fragment, one also obtains a term assignment for the intuitionistic variant LJ. As it turns out, in some respects, λ_{C} actually appears to be closer to classical sequent calculus than to natural deduction.

Therefore, after illustrating how our DS language corresponds to classical natural deduction, we discuss how our CPS language compares to classical sequent calculus and the $\lambda\mu\tilde{\mu}$-calculus. In this chapter, we hence segue from Part I of this thesis, which was about continuation-passing style and translating back to direct style, into Part II, which will be about compilation with the sequent calculus.

4.1 Classical Logic and Natural Deduction

The pure fragment $\lambda_{\mathsf{D}}^{\mathsf{pure}}$ of our language in direct style is essentially a simply typed lambda calculus, which corresponds to intuitionistic natural deduction (Howard, 1980), as embodied by Gentzen's NJ. While being blurred by the requirement that all functions in $\lambda_{\mathsf{D}}^{\mathsf{pure}}$ must be named, the correspondence is still visible.

$$\frac{\Gamma, x : \tau \vdash s_0 : \tau_0 \quad \Gamma, f : \tau \to \tau_0 \vdash s : \tau_r}{\Gamma \vdash \mathbf{def}\, f(x : \tau)\{\, s_0 \,\};\, s : \tau_r}\ [\textsc{Define}] \qquad \frac{f : \tau \to \tau_0 \in \Gamma \quad \Gamma \vdash e : \tau}{\Gamma \vdash f(e) : \tau_0}\ [\textsc{Call}]$$

Ignoring the remaining statement, the definition of a function in rule Define corresponds to the introduction rule of implication, where the left premiss in Define corresponds to the premiss of the implication rule and the right premiss essentially corresponds to the conclusion of the implication rule. Function application in rule Call corresponds to the elimination rule of implication.

The classical calculus NK is an extension of the intuitionistic NJ with an axiom such as the law of excluded middle or double negation elimination. These axioms correspond to control operators that extend the lambda calculus (Griffin, 1989), such as the control operators of our full DS calculus λ_{D}, which extend the pure fragment. Particularly, the control operator **suspend** can be viewed as the axiom of double negation elimination, just as Felleisen's $\mathcal{C}$.

$$\frac{\Gamma,\, k : \neg\tau \vdash s : \#}{\Gamma \vdash \textsf{suspend}\,\{\, k \Rightarrow s \,\} : \tau}\ [\textsc{Suspend}]$$

In the judgment $\Gamma \vdash s : \#$, the statement s does not return anything, because there is nowhere to return to. Logically speaking, this means that it proves a contradiction. The **suspend** statement is of type τ when its body uses the current continuation of negation type $\neg\tau$ to prove a contradiction. That is, the double negation of τ is eliminated.

We can also give a logical interpretation for the typing rules of **process** and **run** as introduction and elimination of the negation connective, respectively.

$$\frac{\Gamma,\, x : \tau \vdash s_0 : \#\quad \Gamma,\, k : \neg\tau \vdash s : \xi}{\Gamma \vdash \textsf{process}\, k(x : \tau)\,\{\, s_0 \,\};\, s : \xi}\ [\textsc{Process}] \qquad \frac{\Gamma \vdash e : \neg\tau\quad \Gamma \vdash s : \tau}{\Gamma \vdash \textsf{run}(e)\,\{\, s \,\} : \#}\ [\textsc{Run}]$$

Again, ignoring the remaining statement, the typing rule for **process** creates a proof of the negation $\neg\tau$ by deriving a contradiction from a proof of τ. The rule for **run** proves a contradiction from a proof of $\neg\tau$ and a proof of τ. In a system where we define negation by means of a bottom type as $\neg\tau \equiv \tau \to \bot$, which is more standard in natural deduction, we could also view the above rules as a special case of implication introduction and elimination, when replacing $\#$ by $\bot$. This would correspond to modeling continuations as functions. Looking at the typing rule of the **run** operator again, we see that it also resembles the cut rule of sequent calculus if we interpret a proof of $\neg\tau$ as a refutation of τ. We will discuss the cut rule in more depth in Section 4.2.

The typing rule for **exit** derives a contradiction from an arbitrary proposition.

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \textsf{exit}\ e : \#}\ [\textsc{Exit}]$$

This is not in general a sound logical rule and should rather be seen as an extra-logical rule to terminate the computation.

With the above rules, it is possible to construct the classical proof of the law of excluded middle using the axiom of double negation elimination, embodied by `suspend`, (example left), and to recover `callcc`, whose type corresponds to Peirce's law (example on the right). Remember that **resume** $k(v)$ is simply syntactic sugar for **run**$(k)\,\{\,$**ret** $v\,\}$.

```
def lawOfExcludedMiddle(): ¬A + A {
  suspend { k ⇒
    process k0(a: A) {
      resume k(Inr(a)) };
    resume k(Inl(k0))
  }
}
```

```
def callcc(program: (A → B) → A): A {
  suspend { k ⇒
    def escape(a: A): B {
      suspend { _ ⇒ resume k(a) } };
    run(k) { program(escape) }
  }
}
```

Operationally, `lawOfExcludedMiddle` captures and removes the current stack `k` and uses it twice. First, to construct a new process `k0` of type `¬A`. Then, to delimit a statement that returns this new process injected as the left alternative `Inl(k0)`. So in a sense, `lawOfExcludedMiddle` tries to continue with the assumption that the proposition is not true, but when presented with a proof, it backtracks and continues with that proof instead. `callcc` captures and removes the current stack `k`, and then defines a function `escape` that, when called, will remove and discard the current stack and resume with `k`, instead. `callcc` then reinstalls the stack `k` and calls the given `program` with `escape`. The definition of `escape` accounts for the fact that we do not model continuations as functions and have a separate construct to invoke them. Otherwise, this definition of `callcc` is as the definition in terms of $\mathcal{C}$ we have seen in Section 3.1.1.

4.2 Classical Sequent Calculus and the $\lambda\mu\tilde{\mu}$-calculus

The sequent calculus was introduced by Gentzen as a more symmetric alternative to natural deduction. Both the premises and conclusions consist of *sequents* $\Gamma \vdash \Delta$, where both sides can contain an arbitrary number of formulae in the classical version. Restricting the right-hand side to at most one formula yields a system for intuitionistic logic. Thus, while in natural deduction the basic system is intuitionistic and we have to add an axiom to make it classical, in sequent calculus the basic system is classical and we have to restrict it to make intuitionistic. This restriction is similar to the one we have imposed when restricting our CPS calculus λ_{C} to its pure fragment. In Section 4.3 we will investigate the relation of λ_{C} and the sequent calculus in more depth.

Natural deduction can also be framed with sequents having always exactly one formula on the right-hand side, and this is usually done when drawing the comparison to type systems, as in the previous section. The difference between sequent calculus and natural deduction is that natural deduction uses right introduction and right elimination rules to act on the one formula on the right, while sequent calculus has right introduction rules and left introduction rules, but no elimination rules. Hence, natural deduction is biased towards proofs; it tries to prove the formula on the right-hand side. In contrast, sequent calculus is symmetric in proofs and refutations. There can be at most one *active* formula being currently under consideration. If the active formula is on the right, we currently try to prove that formula; if it is on the left, then we currently try to refute it.

For logical considerations, it is often left implicit which formula is currently active, but term assignment systems for the sequent calculus (Curien and Herbelin, 2000; Wadler, 2003; Zeilberger, 2008; Downen, 2017) make use of this. An active formula on the right types a *producer* p (variously called program, term, proof, etc.), while an active formula on the left types a *consumer* c (variously called context, coterm, refutation, etc.). The other formulae represent bindings in the typing context. There is a third syntactic category of *statements* s (variously called commands, contradictions, etc.) for the case where no formula is currently active. The first satisfying term assignment for classical sequent calculus was given by Curien and Herbelin (2000) with the $\lambda\mu\tilde{\mu}$-calculus. The syntax (of a slight variation) can be defined as follows.

Syntax of $\lambda\mu\tilde{\mu}$

Producers	p	$::=$	x	Consumers	c	$::=$	α
		$\mid$	$\mu\alpha.s$			$\mid$	$\tilde{\mu}x.s$
		$\mid$	$(x \cdot \alpha) \Rightarrow s$			$\mid$	$(p \cdot c)$
Statements	s	$::=$	$\langle p \mid c \rangle$				

Producers are either variables, μ-abstractions, which allow to abstract over a consumer in a statement, or function abstractions. They construct an element of some type and correspond to expressions in most programming languages. Dually, consumers are either covariables, $\tilde{\mu}$-abstractions, which allow to abstract over a producer in a statement, or function-application frames. They are not typically found as first-class entities in programming languages. Consumers destruct an element of some type, and one can approximately think of them as continuations or program contexts. Producers can only be bound to variables and consumers can only be bound to covariables. For better distinguishability, we usually use small Greek letters for covariables and small Latin letters for variables. The only statements in $\lambda\mu\tilde{\mu}$ are cuts, where a producer and a consumer meet and interact. For example, to apply a function, it is cut with an application frame.

Our definition of functions is slightly different from but equivalent to that of Curien and Herbelin to make the similarity with our CPS calculus more apparent. Similar to how functions in CPS bind an additional continuation parameter, our functions in $\lambda\mu\tilde{\mu}$ bind an additional consumer parameter α. The program context is always explicit in $\lambda\mu\tilde{\mu}$ as a consumer, and can

Producer Typing $\boxed{\Gamma \vdash p :^{\mathsf{prd}} \tau}$

$$\frac{x :^{\mathsf{prd}} \tau \in \Gamma}{\Gamma \vdash x :^{\mathsf{prd}} \tau}\ [\textsc{Ax-R}] \qquad \frac{\Gamma, \alpha :^{\mathsf{cns}} \tau \vdash s}{\Gamma \vdash \mu\alpha.s :^{\mathsf{prd}} \tau}\ [\textsc{Act-R}]$$

$$\frac{\Gamma, x :^{\mathsf{prd}} \tau, \alpha :^{\mathsf{cns}} \tau_0 \vdash s}{\Gamma \vdash (x \cdot \alpha) \Rightarrow s :^{\mathsf{prd}} \tau \to \tau_0}\ [\textsc{Fun-R}]$$

Consumer Typing $\boxed{\Gamma \vdash c :^{\mathsf{cns}} \tau}$

$$\frac{\alpha :^{\mathsf{cns}} \tau \in \Gamma}{\Gamma \vdash \alpha :^{\mathsf{cns}} \tau}\ [\textsc{Ax-L}] \qquad \frac{\Gamma, x :^{\mathsf{prd}} \tau \vdash s}{\Gamma \vdash \tilde{\mu}x.s :^{\mathsf{cns}} \tau}\ [\textsc{Act-L}]$$

$$\frac{\Gamma \vdash p :^{\mathsf{prd}} \tau \quad \Gamma \vdash c :^{\mathsf{cns}} \tau_0}{\Gamma \vdash (p \cdot c) :^{\mathsf{cns}} \tau \to \tau_0}\ [\textsc{Fun-L}]$$

Statement Typing $\boxed{\Gamma \vdash s}$

$$\frac{\Gamma \vdash p :^{\mathsf{prd}} \tau \qquad \Gamma \vdash c :^{\mathsf{cns}} \tau}{\Gamma \vdash \langle p \mid c \rangle}\ [\textsc{Cut}]$$

Figure 4.1: Typing for $\lambda\mu\tilde{\mu}$.

be given a name with this additional consumer parameter, but also using μ-abstractions. The latter can hence be thought of as another form of control operator for undelimited continuations that captures and reifies its surrounding context as a first-class consumer. It was originally introduced by Parigot (1992) before it became a part of the $\lambda\mu\tilde{\mu}$-calculus. The $\tilde{\mu}$-abstractions dually reify a surrounding producer and name it, just as typical let-bindings found in many programming languages.

Typing rules for $\lambda\mu\tilde{\mu}$ There are three different typing judgments in $\lambda\mu\tilde{\mu}$, one for each syntactic category. While producers correspond to expressions in typical programming languages, their judgment $\Gamma \vdash p :^{\mathsf{prd}} \tau$ is slightly different to make it explicit in the judgment that we are typing a producer. Similarly, the judgment $\Gamma \vdash c :^{\mathsf{cns}} \tau$ for consumers makes it explicit that we are typing a consumer. The judgment $\Gamma \vdash s$ for statements indicates that statements do not have a type by omitting a type annotation. Statements do not return anything, but rather represent computations, as we will see below. We deviate from the standard notation in the literature by collecting all bindings for producers and consumers in a single typing context Γ instead of partitioning them into two separate contexts Γ and Δ on the left- and right-hand side of the turnstile (Binder et al., 2024). We therefore annotate whether a binding is a producer or a consumer with $x :^{\mathsf{prd}} \tau$ or $\alpha :^{\mathsf{cns}} \tau$, respectively. We call this annotation the *chirality* of the expression, which is derived from the Greek word for hand. Our typing judgments hence deviate from those of Curien and Herbelin as summarized in the following table.

Judgment Form	Our notation	Curien and Herbelin (2000)
Typing Producers	$\Gamma \vdash p :^{\mathsf{prd}} \tau$	$\Gamma \vdash p : \tau \mid \Delta$
Typing Consumers	$\Gamma \vdash c :^{\mathsf{cns}} \tau$	$\Gamma \mid c : \tau \vdash \Delta$
Typing Statements	$\Gamma \vdash s$	$s : (\Gamma \vdash \Delta)$

The typing rules are given in Figure 4.1. While it is common to make the structural rules of weakening, contraction, and exchange explicit in logical considerations of the sequent calculus,

we leave them implicit for now. We will come back to these rules during the presentation of our compilation pipeline in Section 7.3. The axiom rules Ax-R and Ax-L introduce variables and covariables, respectively. They are the leaves of a derivation tree. The activation rules Act-R and Act-L type μ- and $\tilde{\mu}$-bindings, respectively. The right introduction rule for functions Fun-R adds the bound variable for its input type and the bound covariable for its output type to the context for typing the statement in the function body. This shows that functions do not return a result, as statements do not return. Instead, the result is cut with a consumer, similar to how the result in CPS is fed into a continuation. The left introduction rule for functions Fun-L creates an application frame from a producer of the input type and a consumer of the output type. In rule Cut, we see that the producer and the consumer of the cut must have the same type.

Operational semantics of $\lambda\mu\tilde{\mu}$ When a producer and a consumer of the same type meet in a cut, they interact. Operationally, cuts correspond to redexes, but do not return a result. For example, when cutting a function abstraction with an application frame, we can reduce the application:

$$\langle (x \cdot \alpha) \Rightarrow s \mid (p \cdot c) \rangle \to s[x \mapsto p,\, \alpha \mapsto c]$$

Here $v \mapsto t$ means standard capture-avoiding substitution of variable v by term t. The result is another statement that we can evaluate further. μ- and $\tilde{\mu}$-bindings are reduced in a similar way.

$$\langle \mu\alpha.\, s \mid c \rangle \to s[\alpha \mapsto c]$$
$$\langle p \mid \tilde{\mu}x.\, s \rangle \to s[x \mapsto p]$$

From a logical perspective, cuts correspond to contradictions where a proof and a refutation of the same proposition meet, and rewriting is done to eliminate cuts. Computation or reduction hence corresponds to *cut elimination*.

There are two problems with the above evaluation rules, however. First, if, for example, the producer in the application frame in the rule for function application is a μ-abstraction, $p = \mu\beta.s_0$, then it contains a statement, i.e., an unevaluated computation. This is not a value, and if we want to proceed by call-by-value evaluation, then we have to evaluate it first. To do so, we can lift this computation to the top level and give its result a name that we use in its place. This is called *focusing* (Andreoli, 1992; Curien and Munch-Maccagnoni, 2010), and we have already seen this in the discussion of our DS calculus in Section 2.1.1. We can either add additional evaluation rules, usually called ς-rules, to lift sub-computations to the outside of the statement we are evaluating, which is called dynamic focusing (Wadler, 2003). Or we can perform a transformation on the code before we start evaluating it. This is called static focusing (Curien and Herbelin, 2000) and results in a focused normal form that is a subset of the syntax we have described so far. In the above example, (static) focusing would yield (here $\mathcal{F}$ is the focusing function)

$$\mathcal{F}(\langle (x \cdot \alpha) \Rightarrow s \mid (\mu\beta.s_0 \cdot c) \rangle) = \langle \mu\beta.s_0 \mid \tilde{\mu}y.\langle (x \cdot \alpha) \Rightarrow s \mid (y \cdot c) \rangle \rangle \quad (y \text{ fresh})$$

That is, we obtain a cut of a μ- and a $\tilde{\mu}$-binding, the famous *critical pair*. This is the second problem, the so-called *fundamental dilemma of computation*. Which side of the cut should be reduced? The dilemma is resolved by choosing an evaluation strategy. For call-by-value, we have to reduce the μ-abstraction (otherwise we are back to where we started before focusing). For call-by-name, we have to reduce the $\tilde{\mu}$-abstraction, as we do not want to evaluate the computation s_0 inside the μ-abstraction then. The two evaluation strategies are dual to each other (Wadler, 2003). The evaluation strategy thus also determines which producers and consumers we have to lift out of argument positions during focusing, as we only have to lift the ones we want to reduce. Instead of fixing a global evaluation strategy, it is also easily possible

to make the decision how to reduce a critical pair locally, dependent on the type (Munch-Maccagnoni, 2009). We will come back to this point below, and we will see this again when we investigate our compiler with intermediate languages based on classical sequent calculus in Part II.

4.3 Comparing λ_{C} and $\lambda\mu\tilde{\mu}$

While the typing rules of our DS calculus λ_{D} have a rather clear correspondence to natural deduction, the typing rules in our CPS calculus λ_{C} look a bit different. In particular, terms are not associated with a type, since they do not return. Instead, the result type of a function is given by the type that the continuation of the function expects. This corresponds to a double-negation embedding of propositions. In our types, this is not directly visible, since we use an explicit negation connective instead of a bottom type and thus do not change the function type. For a logical interpretation, the function type $\tau \to \tau_0$ in λ_{C} should hence rather be read as $\tau \to \neg\neg\tau_0$. Classically, these are equivalent since double negations can be eliminated. The typing judgment of terms $\Gamma \vdash t$ can be read as deriving a contradiction. This is more in line with the typing judgment for (cut) statements in the $\lambda\mu\tilde{\mu}$-calculus, although with an explicit bottom type $\bot$, terms could be typed with $\bot$. In the typing rule for function definition, we accordingly derive a contradiction from a proof of τ and $\neg\tau_0$, which is again blurred by the requirement that functions must be bound to a name.

$$\frac{\Gamma,\, x : \tau,\, k : \neg\tau_0 \vdash t_0 \qquad \Gamma,\, f : \tau \to \tau_0 \vdash t}{\Gamma \vdash \mathsf{let}\, f(x : \tau \mid k : \neg\tau_0)\,\{\, t_0 \,\};\, t}\ [\textsc{Let}]$$

When reading negation as refutation, i.e., the continuation as a consumer, this rule looks more like the rule Fun-R of function definition in sequent calculus than the natural deduction rule of λ_{D}, in particular when giving the function a name via a cut with a $\tilde{\mu}$-binding.

$$\dfrac{\dfrac{\Gamma, x :^{\mathsf{prd}} \tau, \alpha_k :^{\mathsf{cns}} \tau_0 \vdash s_0}{\Gamma \vdash (x \cdot \alpha_k) \Rightarrow s_0 :^{\mathsf{prd}} \tau \to \tau_0}\,\textsc{Fun-R} \qquad \dfrac{\Gamma, f :^{\mathsf{prd}} \tau \to \tau_0 \vdash s}{\Gamma \vdash \tilde{\mu} f.s :^{\mathsf{cns}} \tau \to \tau_0}\,\textsc{Act-L}}{\Gamma \vdash \langle (x \cdot \alpha_k) \Rightarrow s_0 \mid \tilde{\mu} f.s \rangle}\,\textsc{Cut}$$

Here we have written α_k to indicate how this covariable corresponds to the continuation k. Despite this similarity, λ_{C} is still based on natural deduction, as is more clearly visible in the rule for function application, which is a (right) elimination rule.

$$\frac{f : \tau \to \tau_0 \in \Gamma \qquad \Gamma \vdash p : \tau \qquad \Gamma \vdash c : \neg\tau_0}{\Gamma \vdash f(p \mid c)}\ [\textsc{App}]$$

In the sequent calculus, this hence becomes a left introduction rule for application frames combined with a cut (Gentzen, 1935a,b; Ostermann et al., 2022).

$$\dfrac{\dfrac{f :^{\mathsf{prd}} \tau \to \tau_0 \in \Gamma}{\Gamma \vdash f :^{\mathsf{prd}} \tau \to \tau_0}\,\textsc{Ax-R} \qquad \dfrac{\Gamma \vdash p :^{\mathsf{prd}} \tau \quad \Gamma \vdash c :^{\mathsf{cns}} \tau_0}{\Gamma \vdash (p \cdot c) :^{\mathsf{cns}} \tau \to \tau_0}\,\textsc{Fun-L}}{\Gamma \vdash \langle f \mid (p \cdot c) \rangle}\,\textsc{Cut}$$

We can still see that the function application rule App in λ_{C} is a double negation embedding, as we derive a contradiction from a proof of τ and $\neg\tau_0$, hence proving $\neg\neg\tau_0$. And when reading the continuation as consumer, this again looks very similar to the above derivation for function application in the $\lambda\mu\tilde{\mu}$-calculus.

Continuations vs Consumers

However, continuations and consumers are not the same. In fact, a consequence of being based on natural deduction is that λ_{C} does not support consumers in the same way as producers. Rather, consumers must always be modeled more indirectly as continuations with a negation type. In particular, unlike in $\lambda\mu\tilde{\mu}$, it is not possible to directly bind an application frame to a (co)variable. Instead, it must be modeled as a continuation that abstracts over the function that will be applied to it. The following correspondence is thus rather approximate.

$$\textbf{cnt}\, k(f)\,\{\, f(p \mid c)\,\};\, s \mathrel{\widehat{\approx}} \langle \mu\alpha_k.s \mid (p \cdot c)\rangle$$

This shows that even though continuations are first-class in λ_{C}, there is no direct first-class support for more *structured* consumers. Languages based on classical sequent calculus are hence more flexible in this regard.

Instead of binding the application frame directly to a covariable, we can also use a $\tilde{\mu}$-abstraction in $\lambda\mu\tilde{\mu}$ to abstract over the function, corresponding more closely to continuations in λ_{C}.

$$\textbf{cnt}\, k(f)\,\{\, f(p \mid c)\,\};\, s \mathrel{\widehat{=}} \langle \mu\alpha_k.s \mid \tilde{\mu}f.\langle f \mid (p \cdot c)\rangle\rangle$$

To make this correspondence work, we have to employ call-by-value evaluation order, ensuring that the μ-binding in the critical pair is reduced first.

More generally, we can model continuations in λ_{C} with $\tilde{\mu}$-abstractions in $\lambda\mu\tilde{\mu}$ by binding them to a name with a μ-binding. Application of a continuation is again modeled with a cut.

$$\begin{array}{ll} \textbf{cnt}\, k(x)\,\{\, s_0\,\};\, s & \mathrel{\widehat{=}}\ \langle \mu\alpha_k.s \mid \tilde{\mu}x.s_0\rangle \\ k(p) & \mathrel{\widehat{=}}\ \langle p \mid \alpha_k\rangle \end{array}$$

There is a difference between continuations in λ_{C} and $\tilde{\mu}$-abstractions in $\lambda\mu\tilde{\mu}$, however. While in λ_{C} it is possible to pass a continuation as an argument to another continuation, this is not possible with a $\tilde{\mu}$-abstraction. The reason is that the $\tilde{\mu}$-abstraction is a consumer but abstracts over a producer, so we cannot pass it another $\tilde{\mu}$-abstraction. Similarly, the ordinary function argument of a function must be a producer in $\lambda\mu\tilde{\mu}$, but can be a continuation in λ_{C}. The latter restriction could be lifted by introducing a second function type in $\lambda\mu\tilde{\mu}$ that expects a consumer as its ordinary argument. This does not exactly correspond to the usual function type anymore, but rather to the so-called ⅋-connective (pronounced *par*) (Binder et al., 2024).

Another option to lift the above restrictions is to add a negation type to the $\lambda\mu\tilde{\mu}$-calculus. The negation connective turns a producer into a consumer.

Syntax

$$\text{Producers} \quad p ::= \ldots \mid \neg(x) \Rightarrow s \qquad\qquad \text{Consumers} \quad c ::= \ldots \mid \neg(p)$$

Typing

$$\frac{\Gamma,\, x :^{\mathsf{prd}} \tau \vdash s}{\Gamma \vdash \neg(x) \Rightarrow s :^{\mathsf{prd}} \neg\tau}\ [\textsc{Cnt-R}] \qquad\qquad \frac{\Gamma \vdash p :^{\mathsf{prd}} \tau}{\Gamma \vdash \neg(p) :^{\mathsf{cns}} \neg\tau}\ [\textsc{Cnt-L}]$$

Then we can model the continuations of λ_{C} as follows.

$$\begin{array}{ll} \textbf{cnt}\, k(x)\,\{\, s_0\,\};\, s & \mathrel{\widehat{=}}\ \langle \neg(x) \Rightarrow s_0 \mid \tilde{\mu}k.s\rangle \\ k(p) & \mathrel{\widehat{=}}\ \langle k \mid \neg(p)\rangle \end{array}$$

As k is a producer in $\lambda\mu\tilde{\mu}$, it can again be passed to another continuation or as the ordinary argument of a function. However, it cannot be passed as the consumer argument of a function. We could additionally change functions to take a producer of negation type instead of a consumer, which would allow us to faithfully embed λ_{C} into $\lambda\mu\tilde{\mu}$. This embedding then, of course,

does not make use of consumers anymore in $\lambda\mu\tilde{\mu}$. An alternative might be to instead convert between producers of negation type and consumers as needed via

$$\begin{array}{lll} \alpha :^{\mathsf{cns}} \tau & \longrightarrow \neg(x) \Rightarrow \langle x \mid \alpha \rangle & (x \text{ fresh}) \\ k :^{\mathsf{prd}} \neg\tau & \longrightarrow \tilde{\mu}x.\langle k \mid \neg(x) \rangle & (x \text{ fresh}) \end{array}$$

Either way, this discussion shows that while λ_{C} and the $\lambda\mu\tilde{\mu}$-calculus are quite similar in that they make the control flow of a program explicit, they differ in how they express consumers. In λ_{C} the only consumers are continuations, which are represented as producers of negation type. In $\lambda\mu\tilde{\mu}$ there is direct support for more structured consumers, symmetric to producers. In Part II we will see how this duality of producers and consumers can be used in a compiler that uses intermediate representations based on classical sequent calculus.

Evaluation Strategies and Focusing

There is another difference between λ_{C} and $\lambda\mu\tilde{\mu}$. In Section 2.1.1 we have seen that a core property of CPS is that the evaluation order is syntactically explicit. Translations into CPS hence change the structure of the original program. Repeating the example of a nested function call, a call-by-value translation would look as follows.

$$\mathcal{C}[\![f(g(3))]\!]_k = \mathbf{cnt}\, k_0(x)\, \{ f(x \shortmid k) \};\ g(3 \shortmid k_0)$$

This forces λ_{C} to be in a focused form. In our DS language λ_{D}, we therefore imposed a similar restriction to make the correspondence between DS and CPS clearer. For languages based on sequent calculus, focusing is a separate step, and the evaluation order is only determined by how critical pairs are reduced, as we have discussed in Section 4.2. A translation of a nested function call into $\lambda\mu\tilde{\mu}$ therefore might look as follows.

$$\mathcal{C}[\![f(g(3))]\!]_c = \langle f \mid (\mu\alpha.\langle g \mid (3 \cdot \alpha) \rangle \cdot c) \rangle$$

The nesting structure of the translated program is still the same as in the original version, independent of the evaluation strategy. We will see how such a translation from a non-focused DS language into a language based on sequent calculus can be defined in an efficient way in more depth in Chapter 6. After focusing, the above term looks more akin to the CPS term under call-by-value:

$$\mathcal{F}(\mathcal{C}[\![f(g(3))]\!]_c) = \langle \mu\alpha.\langle g \mid (3 \cdot \alpha) \rangle \mid \tilde{\mu}x.\langle f \mid (x \cdot c) \rangle$$

But in contrast to CPS, it still does not prescribe the evaluation order; both are possible, and the decision can also be made locally, based on the type.

4.3.1 Translating the $\lambda\mu\tilde{\mu}$-calculus back to direct style

To conclude this chapter, we briefly look at how the $\lambda\mu\tilde{\mu}$-calculus can be translated back to direct style, but in a less formal way than in the previous chapters. Such a translation can be defined in a very similar fashion to the one for λ_{C}, in particular if we consider a focused version of $\lambda\mu\tilde{\mu}$ where all arguments must be (co)variables and where all functions must have a name, similar to the requirements we have imposed on λ_{D} and λ_{C}. While this fragment would not be closed anymore under a standard substitution semantics, we could easily give an abstract machine semantics similar to λ_{C}. For a more in-depth discussion of translations forth and back, in particular with respect to evaluation order and with an unfocused version, we refer to the dissertation of Downen (2017) (Chapter IX). Downen's languages feature data and codata types, similar to the extended languages we will present in Chapter 6. Note, however, that the data and codata types discussed there are less general than ours, in that they are restricted to those that can be expressed in a pure DS language. This means that constructors of data types

cannot have consumer arguments, and destructors of codata types always have exactly one consumer argument. Moreover, Downen does not try to detect non-pure control flow and hence inserts a control operator for every binding of a consumer. We do not discuss data and codata types here, but will briefly come back to the additional challenges they present for translating back to direct style in Section 6.4, after we have properly introduced them.

Remember that the translation back to direct style for λ_{C} is bottom-up and has an additional continuation variable as output. The latter is now replaced by a covariable. Let us first assume that there is no negation type. We have to consider all possible combinations of producers and consumers that can meet in a cut, as allowed by the typing rules. However, due to our requirement that functions must have a name, we can additionally rule out cuts of a function abstraction with a covariable. Moreover, there cannot be a cut of a function abstraction with an application frame. Such a cut could be immediately reduced anyway by simple renaming, since application frames must consist of a variable and a covariable. Similarly, a cut of a variable with a $\tilde{\mu}$-binding and of a μ-binding with a covariable could be immediately reduced, so we do not consider them here. We also add one statement, however. To be able to write terminating programs, we use an **exit** statement, corresponding to the **exit** term in λ_{C}.

Thus, the base cases are the cut of a variable with an application frame, corresponding to function application, the cut of a variable with a covariable, corresponding to the invocation of a continuation, and the **exit** statement:

$$\mathcal{D}[\![\, \langle f \mid (x \cdot \alpha) \,]\!] = f(x) \rightsquigarrow \alpha \qquad \mathcal{D}[\![\, \langle x \mid \alpha \rangle \,]\!] = \mathbf{ret}\, x \rightsquigarrow \alpha \qquad \mathcal{D}[\![\, \mathbf{exit}\, x \,]\!] = \mathbf{exit}\, x \rightsquigarrow \bullet$$

Note that in contrast to λ_{C}, the ordinary argument x cannot be a covariable in the case of function application, so we do not have to check whether it occurs free in the translated term. The cases left to consider are the following:

$$\langle (x \cdot \alpha) \Rightarrow s_0 \mid \tilde{\mu} f.s \rangle \qquad \langle \mu\alpha.s \mid \tilde{\mu} x.s_0 \rangle \qquad \langle \mu\alpha.s \mid (x \cdot \beta) \rangle$$

The cut on the left of a function abstraction with a $\tilde{\mu}$-binding corresponds to function definition, and the critical pair in the middle corresponds to the definition of a continuation. The translation for them hence proceeds in the same way as for function definition and continuation definition in λ_{C}, respectively, deciding where to insert control operators based on the covariable output. In λ_{D} there is no way to bind an application frame directly to a variable, however, as is done in the cut on the right. But as we have already seen before, we can simply wrap the application frame into a $\tilde{\mu}$-abstraction that abstracts over the function to apply to the frame.

$$\langle \mu\alpha.s \mid (x \cdot \beta) \rangle \longrightarrow \langle \mu\alpha.s \mid \tilde{\mu} f.\langle f \mid (x \cdot \beta) \rangle \rangle \quad (f \text{ fresh})$$

Since the body of the $\tilde{\mu}$-binding in this case is a function application with output covariable β, we can always turn this into a **val**-binding for f.

The question left to consider is how to treat a negation construct. For this, the interesting cases are

$$\langle \neg(x) \Rightarrow s_0 \mid \tilde{\mu} k.s \rangle \qquad \langle k \mid \neg(x) \rangle \qquad \langle \mu\alpha.s \mid \neg(x) \rangle$$

As we have seen above, the cut on the left also corresponds to a continuation definition, and the cut in the middle corresponds to an invocation of a continuation, so they can be translated accordingly. Importantly, however, the additional output of the translation for the continuation invocation here is a continuation variable instead of a covariable, so there are two different kinds of additional output now. For the cut on the right, we can again wrap the right-hand side into a $\tilde{\mu}$-abstraction.

$$\langle \mu\alpha.s \mid \neg(x) \rangle \longrightarrow \langle \mu\alpha.s \mid \tilde{\mu} k.\langle k \mid \neg(x) \rangle \rangle \quad (k \text{ fresh})$$

This again corresponds to a continuation definition, one that expects another continuation as an argument, and is hence also translated in that way. With the negation construct, we now

have two different models of continuations that we have to consider when translating back to direct style. In fact, the negation type is a special case of codata types, as is the function type, and we will briefly come back to the additional difficulties general codata types present for translations back to direct style in Section 6.4.

Part II

Compiling with the Sequent Calculus

Abstract. Compiling a high-level functional programming language to machine code that can be executed efficiently on a modern machine is complicated, since we have to traverse many different levels of abstraction. This is particularly challenging if the language contains some form of control effects and a mix of different evaluation strategies, such as call-by-value data types and call-by-name codata types. In this part of the thesis, we tell the complete story, starting from a simple functional programming language with control effects and both data and codata types, and ending up with machine code for standard platforms. What distinguishes our compiler from all other existing compilers for functional programming languages is that, instead of natural-deduction-based languages like the lambda calculus, we use sequent-calculus-inspired languages throughout all intermediate stages. These sequent-calculus-based languages are characterized by the first-class nature of consumers, which represent program contexts. In this sense, we view this work as a continuation, and generalization, of Andrew Appel's landmark work on "Compiling with Continuations".

After providing an overview of the compilation pipeline in Chapter 5, we present our surface language and a high-level intermediate language based on classical sequent calculus in Chapter 6. Both languages feature general data and codata types. The surface language supports control effects by providing a set of control operators for undelimited continuations. This non-local control flow can be naturally expressed in a language based on classical sequent calculus due to the symmetry of producers and consumers. We show how our surface language can be translated into the high-level intermediate representation in an efficient way. To forge a bridge to the first part of this thesis, we also consider the challenges posed by general data and codata types in a translation back to direct style.

In Chapter 7 we derive a normal form for the terms of our intermediate language that is suitable for direct code generation. To do so, we first apply two transformations, each targeting a smaller fragment of the language. The focusing transformation binds all terms in a program to a name, and the shrinking transformation removes redundant constructs from the language. Then we translate the transformed version into a language with a slightly different structure, making the structural rules of exchange, weakening, and contraction explicit. All of these steps bring us closer to machine code. In Chapter 8 we then show how we can generate machine code in a surprisingly straightforward way from the final language derived in Chapter 7.

To evaluate our approach, we have implemented the complete compilation pipeline. Moreover, we have implemented a considerable number of benchmark programs. In Chapter 9 we discuss our implementation and present our benchmark results. We compare our implementation against a number of compilers for different industrial and research languages. The results demonstrate the viability of our approach.

This part is closely based on the following journal article, currently under consideration for publication:

Marius Müller, David Binder, Marco Tzschentke, Philipp Schuster, Klaus Ostermann, and Jonathan Immanuel Brachthäuser. 2026. Compiling with the Sequent Calculus. Submitted to ACM Trans. Program. Lang. Syst.

The journal article itself is based on the following publications (Schuster et al., 2025a; Binder et al., 2024):

Philipp Schuster, Marius Müller, Klaus Ostermann, and Jonathan Immanuel Brachthäuser. 2025. Compiling Classical Sequent Calculus to Stock Hardware: The Duality of Compilation. *Proc. ACM Program. Lang.* 9, OOPSLA1. DOI: https://doi.org/10.1145/3720507

David Binder, Marco Tzschentke, Marius Müller, and Klaus Ostermann. 2024. Grokking the Sequent Calculus (Functional Pearl). *Proc. ACM Program. Lang.* 8, ICFP. DOI: https://doi.org/10.1145/3674639

5 Overview of the Compilation Pipeline

Continuation-passing style has long been established as an intermediate representation for compilers (Steele, 1978; Appel, 1992). We argue – as other researchers have before us (Downen et al., 2016) – that we should also examine Gentzen (1935a)'s classical sequent calculus as a basis for an attractive IR that bridges the gap between high- and low-level languages. Sequent-calculus-based intermediate representations have specific advantages that distinguish them from IRs based on natural deduction, such as direct style or continuation-passing style. In this second part of the thesis, we will thus present a full compilation pipeline from an interesting functional surface language down to machine code, using IRs based on classical sequent calculus throughout all intermediate stages.

Expressing advanced control flow and first-class contexts We have already introduced sequent calculus in Chapter 4, where we have also drawn a comparison to CPS. We have seen that the symmetry between producers and consumers (Downen and Ariola, 2014; Ostermann et al., 2022) makes term assignments for classical sequent calculus particularly well-suited to expressing control effects, without the need for a low-level encoding, such as state machines, or support from a runtime system. Consumers are first-class: They can be named, and hence, they can also be stored, duplicated, and dropped, similar to continuations in CPS. In fact, consumers can model arbitrary program contexts directly, generalizing continuations. For example, consumers can be function-application frames, making them first-class entities. Similarly, consumers can be first-class pattern-matching contexts for algebraic data types. As an example, consider the list data type, whose producers are terms built from its constructors `Nil` and `Cons`. Consumers of that type are built by pattern matching on `Nil` and `Cons` and returning a statement in each branch, as in the following consumer:

$$\frac{\Gamma \vdash s_1 \quad \Gamma,\, x :^{\mathsf{prd}} \mathsf{i64},\, xs :^{\mathsf{prd}} \mathtt{List}[\mathsf{i64}] \vdash s_2}{\Gamma \vdash \mathbf{case}\,\{\,\mathtt{Nil} \Rightarrow s_1,\, \mathtt{Cons}(x,\, xs) \Rightarrow s_2\,\} :^{\mathsf{cns}} \mathtt{List}[\mathsf{i64}]}\ [\textsc{Case}]$$

The interesting thing to observe is that instead of a familiar case expression like $p.\mathbf{case}\,\{\ldots\}$ that *combines* the scrutinee and the branches in a single expression, the sequent calculus *factors* these case expressions into three parts: a producer p, the (first-class) consumer $\mathbf{case}\,\{\ldots\}$, and the cut $\langle p \mid \mathbf{case}\,\{\ldots\}\rangle$ combining them.

Controlling evaluation order We have further seen that sequent-calculus-based languages can naturally and uniformly express both call-by-value (CBV) and call-by-name (CBN) evaluation order, while retaining much of the original nesting structure of a source program in direct style. The two evaluation orders are both useful in different scenarios. While CBV is the well-known standard strategy used in most languages, CBN allows for demand-driven and streaming computation. The evaluation order in languages based on sequent calculus is determined by how critical pairs are reduced. This decision can be made locally, instead of prescribing a global evaluation order. While we can hence mix both evaluation strategies in the same program, we must do so consistently. Classical sequent calculus facilitates this, since every logical connective naturally comes in two polarities: positive and negative (Andreoli, 1992; Girard, 1991, 1993; Munch-Maccagnoni, 2009; Downen and Ariola, 2020, 2021; Downen, 2017), together with shifts that mediate between call-by-value and call-by-name (Zeilberger, 2008). To be able to mix evaluation strategies, we employ both (positive) data types and (negative) codata types (Hagino,

1989) in a surface language. For data types defined by their constructors, we use call-by-value, while for codata types, which are a form of object-oriented feature defined by their destructors, we use call-by-name. These choices ensure the validity of all η-laws (Curien and Herbelin, 2000; Wadler, 2003).

In other words, the symmetric nature of sequent calculus leads to support for algebraic data and codata types in one system in a perfectly dual way. We can take advantage of this during compilation because it allows us to treat data and codata types in exactly the same fashion. An IR based on sequent calculus hence retains the virtues of CPS, while being more general. All of these features make classical sequent calculus a natural basis for a high-level compiler intermediate representation for surface languages with control effects and codata types.

Compiling with the Sequent Calculus

An intermediate representation based on sequent calculus corresponds to a logical system quite directly, and the symmetry between structured producers and consumers makes it appear to be a high-level language. However, programs are run on real hardware where registers could contain anything at any point, memory is an unstructured sequence of bytes, and code is an unstructured sequence of instructions. A compiler IR should also make it easy to generate code for such machines. How is it possible to bridge this gap between theoretical logical systems and practical computer hardware?

As it turns out, with the right preparations, compilation from a modern high-level language to machine code is surprisingly easy. In this paper, we show that, in fact, an intermediate representation based on sequent calculus can serve as a bridge between high-level logical proofs and low-level machine code. We transform its terms into a normal form which, on the one hand, constitutes a fragment of classical sequent calculus and enjoys safety in the form of progress and preservation. On the other hand, it admits a surprisingly direct translation to machine code and has a small-step operational semantics with an abstract machine. We demonstrate how the concepts of logic, in the form of types, typing judgments, and typing rules, can be used to impose invariants, giving structure to machine code, registers, and memory. While the structure of an IR based on sequent calculus is also very promising with regards to optimizations (Downen et al., 2016; Binder et al., 2024), in this thesis we do not set out for an exploration of these. Rather, we present a complete compilation pipeline, starting with a surface language with support for different evaluation strategies, codata types, and control effects, and ending up with machine code. This work is thus supposed to form the basis for future investigations.

Additionally, we have implemented the complete compilation pipeline. While in this thesis we present code generation for RISC-V, our implementation can also generate code for the AArch64 and x86-64 instruction sets. In fact, we have executed all examples in this part of the thesis on the latter two architectures, as well as a considerable number of benchmark programs of varying size, which we compare against several other compilers for various languages. Even though the other compilers produce optimized code, compared to our rather naive compilation strategy, the benchmark results demonstrate that our approach is viable and produces code that performs roughly in the same league as existing ahead-of-time compilers.

While we mostly produce slower code than the optimizing compilers, our performance still falls within the same order of magnitude, and we expect substantial improvements after we implement optimizations in our compiler. We will discuss the results in detail in Chapter 9.

In this chapter, we give an overview of our complete compilation pipeline, which, as already shown in Chapter 1, can be illustrated as follows:

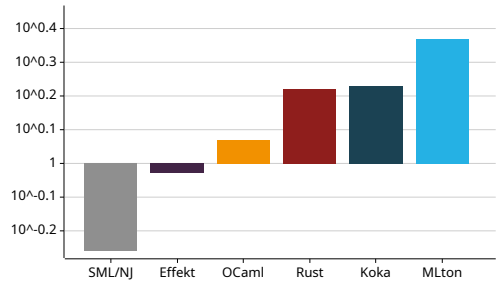
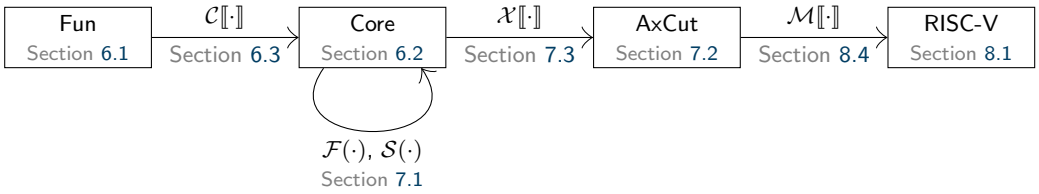


Figure 5.1: Geometric mean of relative speedups over all benchmark programs compared to our implementation, higher is better.



We will explain the characteristics of each stage, represented by one of the boxes, and show how each translation step, represented by the arrows, brings us from our high-level surface language closer to machine code. In the subsequent chapters, we will discuss all of these stages and steps in detail.

5.1 The Surface Language Fun

The surface language Fun is in many regards an ordinary functional programming language, with a simply typed lambda calculus such as $\lambda_{\mathsf{D}}^{\mathsf{pure}}$ from Section 2.1 at its core. However, it also includes features that broaden the scope of our compilation technique beyond traditional functional programming.

Top-level functions Programs in Fun are structured by top-level function definitions that are not first-class. That is, these functions can only be called, but they cannot be passed to or returned from other functions, distinguishing them from first-class functions. In return, these top-level functions can be arbitrarily recursive, even mutually so, and are the only way to express recursion in Fun. We want to stress, however, that Fun is not a first-order language; in fact, it features a generalization of first-class functions, as we will see next.

Data and codata types Besides algebraic data types and pattern matching, Fun also features codata types (Hagino, 1989) and copattern matching (Abel et al., 2013). While data types are defined by their *constructors*, codata types are defined by *destructors*, which specify how their inhabitants can be destructed or observed. Codata types are thus similar to interfaces defined by abstract methods, as in object-oriented programming, and copattern matches can be thought of as creating objects by implementing all methods. First-class functions are one important example, since they can be defined as a codata type with one `apply` destructor (Setzer, 2003) and with lambda abstraction corresponding to copattern matching on this destructor:

Example 36 (Functions as Codata)**.**

```
i64 → i64  ≙ codata Fun { apply(x : i64) : i64 }
λx.t       ≙ new { apply(x) ⇒ t }
t1 t2      ≙ t1.apply(t2)
```

Another typical example are streams defined by two destructors `head` and `tail`:

Example 37 (The Stream Codata Type)**.**

```
codata Stream { head : i64, tail : Stream }
def ascending(n : i64) : Stream { new { head ⇒ n, tail ⇒ ascending(n + 1) } }
def three() : i64 { ascending(1).tail.tail.head }
```

In this example, the top-level function ascending defines an infinite stream of ascending integers, and the definition three observes the third element of this stream.

Mixed evaluation strategies Fun is designed to maximize the number of valid reasoning principles, among them the important class of η-laws. But the validity of η-laws in languages with general recursion or side effects depends on the evaluation strategy: The η-laws for data types are only valid under call-by-value order, and the η-laws for codata types are only valid for call-by-name (Binder et al., 2024; Downen and Ariola, 2020). For this reason, Fun uses call-by-value for data types and call-by-name for codata types, which can be observed in programs with side effects, such as this one:

Example 38 (Codata Types are CBN)**.**

```
def zeros() : Stream { println_i64(0); new { head ⇒ 0, tail ⇒ zeros() } }
def printAndHead() : i64 { let s = zeros(); println_i64(42); s.head }
```

The top-level function zeros returns a stream of zeros and prints 0 when it is called. printAndHead calls this function and binds the stream to the variable s, prints 42, and then observes the first element of s. Since streams are a codata type, we evaluate the call of zeros by-name, and printAndHead therefore first prints 42 and then 0, not the other way around. This ensures that the definition of stream s can be η-expanded to

```
new { head ⇒ zeros().head, tail ⇒ zeros().tail }
```

without changing the semantics, enabling local reasoning about the behavior for the programmer.

Control operators Fun also supports non-local control effects using the control operators **label** and **goto**, which are similar to **suspend** and **run** in λ_{D} from Section 3.1. In contrast to **suspend**, however, **label** behaves more like `let/cc` (Reynolds, 1972) and not like Felleisen et al.'s $\mathcal{C}$, as it leaves the captured program context in place instead of removing it. Moreover, we use *covariables*, denoted by lower-case Greek letters and annotated with cns, to represent the captured program contexts. To illustrate the usage of these control operators, consider the multiplication of a list of integers with early return.

Example 39 (Multiplication with Early Return)**.**

```
def rmult(xs : List[i64]) : i64 { label α { rmultRec(xs, α) } }
def rmultRec(xs : List[i64], α :cns i64) : i64 {
  xs.case { Nil ⇒ 1, Cons(y, ys) ⇒ if y ≡ 0 { goto α (0) } else { y * rmultRec(ys, α) } }
}
```

Here we capture the current continuation in rmult, bind it to covariable α, and pass it as an argument to rmultRec, which performs the recursion over the list. If rmultRec encounters a 0 in the list, it immediately returns (without any stack unwinding) by throwing to the previously captured continuation.

In the rest of this section, we use the code from Example 39 to illustrate all intermediate stages of the compiler.

5.2 The High-Level Intermediate Language Core

In the first step (detailed in Section 6.3), we translate the surface language Fun into the high-level intermediate language Core. The foundation of Core is the $\lambda\mu\tilde{\mu}$-calculus introduced in Section 4.2, a term assignment system for classical sequent calculus. Similar to Fun, Core supports data and codata types (Downen and Ariola, 2020) and top-level functions. As we have seen for $\lambda\mu\tilde{\mu}$, the control flow in this language is made explicit using explicit *consumers*, similar to how continuation-passing style uses explicit continuations. In contrast to CPS, however, producers and consumers are completely symmetric in classical sequent calculus. Consider again our running example from Example 39 of multiplication of a list with early return, this time in Core.

Example 40 (Multiplication with Early Return in Core)**.**

```
def rmult(xs :prd List[i64], β :cns i64) { ⟨μα.rmultRec(xs, α, α) | β⟩ }
def rmultRec(xs :prd List[i64], α :cns i64, β :cns i64) {
    ⟨xs | case { Nil ⇒ ⟨1 | β⟩,
                 Cons(y, ys) ⇒ if y ≡ 0 { ⟨0 | α⟩ } else { ⟨y * μγ.rmultRec(ys, α, γ) | β⟩ } }⟩
}
```

Similar to how functions in CPS receive an additional continuation argument, both top-level functions have an additional covariable β, which stands for the current program context, instead of having a return type. This covariable is made available in the body of rmult, where it is bound to a covariable α by a μ-abstraction and then used twice when calling rmultRec. This implements the control operator **label**. As in the original program, the first covariable α abstracted by rmultRec is for returning early, which can be seen in the then branch of the conditional. There we cut α with 0 when encountering a 0 in the list, hence returning directly to the program context at the original call site of rmultRec. This implements the control operator **goto**. In the **else** branch, we cut the result of the multiplication with the second consumer parameter β for a normal return. In the recursive call of rmultRec, we still use α for early return, but abstract a fresh covariable γ to stand for the new program context. We will see how the latter is actually bound to γ in the next subsection.

The pattern match in the body of rmultRec is now a consumer term that is separated from its scrutinee xs. Instead, it meets the latter in a cut. This also facilitates giving pattern matches a name by binding them to covariables, making them first-class entities. The same is true for destructors, which are also separated from the object they are called on. We will describe this in more detail in Section 6.2.

The above example shows that top-level functions can have multiple consumer parameters, which allows us to express different complex control structures directly. While rmult has one consumer parameter, similar to standard CPS, rmultRec has two, one for normal return and one for early return, which is more akin to double-barreled CPS. Similar to top-level functions, destructors of codata types in Core also do not have return types, but instead receive an additional consumer argument. For example, the codata type of first-class functions from Example 36 becomes

```
codata Fun { apply(x :prd i64, α :cns i64) }
```

In general, destructors can also have multiple consumer parameters. In fact, top-level functions and destructors can even have zero consumer parameters. Such definitions are not readily expressed in Fun, where we always need a return type[4].

Example 41 (Coroutining Send and Receive)**.** As an example, consider the following two mutually recursive declarations. Each of them defines two destructors, respectively `stop` and `push`, and `end` and `pull`.

```
codata Receiver { stop, push(n :prd i64, rest :prd Sender) }
codata Sender { end, pull(next :prd Receiver) }
```

Together, they specify the protocol of interaction between a sender and a receiver of integers. When we push an integer, we yield control to the receiver, describing the rest of the sending behavior. Dually, when we pull, we describe the next receiving behavior. Potentially infinite processes like these are best modeled with codata types. In the following, we create a sender zeroes, which always responds with integer 0, and a receiver discard, which discards all integers pushed into it.

```
def zeroes(r :prd Receiver) {
  ⟨ r | push(0, new { end ⇒ exit 0, pull(next) ⇒ zeroes(next) }) ⟩
}
def discard(s :prd Sender) {
  ⟨ s | pull(new { stop ⇒ exit 0, push(n, rest) ⇒ discard(rest) }) ⟩
}
def connect() { discard(new { end ⇒ exit 0, pull(next) ⇒ zeroes(next) }) }
```

In the definition of zeroes, we push a 0 and continue to behave as zeroes on the next pull. In the definition of discard, we pull the next integer, and continue to behave as discard, ignoring the next integer pushed. Both definitions exit the program upon completion, returning control to the operating system. We connect the two processes, by calling the receiver discard with the sender behaving as zeroes. They will coroutine between each other, until one of them is finished, a potentially infinite process. There is no intermediary, no operating system, nor runtime system between the two processes. They directly interact with each other, following a common protocol that guarantees safety.

Having explicit control flow in the intermediate representation not only facilitates implementing control operators and expressing complex control structures, but is also helpful during compilation, as has been demonstrated for CPS (Appel, 1992; Kennedy, 2007). It makes the language closer to what happens on a real machine. In particular, every branch of the abstract syntax tree ends with a jump to another location, i.e., we can never fall through in branching constructs.

5.3 Transformations on Core

In the next step we apply two transformations on Core to bring it closer to a normal form that is suitable for code generation. The focusing and shrinking transformations each target a smaller fragment of Core.

5.3.1 Focusing

Static focusing (Andreoli, 1992; Curien and Herbelin, 2000) is a transformation that names all intermediate computations and terms using μ- and $\tilde{\mu}$-bindings so that only variables and covariables are allowed to occur in argument positions. We illustrate the idea using our running example from Example 40, the details are discussed in Section 7.1.1 (also cf. Section 4.2).

[4] To facilitate such definitions, we could add some kind of type expressing that a top-level function or destructor does not return. We will come back to this point in Section 6.4.

Example 42 (Focusing of Multiplication with Early Return)**.** The only part of the program in Example 40 that has a non-(co)variable term in argument position is the **else** branch in rmultRec. Focusing this statement yields (here $\mathcal{F}$ is the focusing function)

$$\mathcal{F}(\langle y * \mu\gamma.\mathsf{rmultRec}(ys, \alpha, \gamma) \mid \beta \rangle) = \langle \mu\gamma.\mathsf{rmultRec}(ys, \alpha, \gamma) \mid \tilde{\mu}r.\langle y * r \mid \beta \rangle \rangle$$

The μ-abstraction is lifted out of the multiplication by generating a fresh name r for it, which is bound by a $\tilde{\mu}$-abstraction, and then cutting the latter with the lifted term. The variable r is put in the place of the lifted term in the multiplication. Here we can see how the new program context for the recursive call, consisting of the multiplication with y and the outer context β, is bound to the covariable γ abstracted for it during the translation into Core.

Giving names to all terms is common in many intermediate representations, such as A-normal form (Flanagan et al., 1993), continuation-passing style, or SSA (Cytron et al., 1991). It corresponds to how a real machine stores computed results in registers (or memory) and then uses them in further computations.

5.3.2 Shrinking

Shrinking eliminates redundant constructs from the focused language by first inlining all combinations of producers and consumers into the cut statement, if allowed by typing, and then transforming away several combinations. The resulting fragment Shrunk Core only consists of statements. In particular, we eliminate *critical pairs*, i.e., cuts of a μ- with a $\tilde{\mu}$-binding, such as the one introduced by the focusing transformation in Example 42 above. As we have seen in Section 4.2, these critical pairs play an important role in the reduction theory of our sequent-calculus-based languages, determining the evaluation order. Eliminating them ensures that no variable is ever bound to a μ-abstraction and no covariable is ever bound to a $\tilde{\mu}$-abstraction, that is, these constructs only ever occur directly in cuts to give names to other constructs. This way, when (co)variables occur in cuts, it is always clear from their type what they stand for. Similarly, we eliminate completely unknown cuts $\langle x \mid \alpha \rangle$, i.e., cuts of a variable with a covariable. This leaves us only with statements for which there is a surprisingly straightforward way to generate machine code. The details can be found in Section 7.1.2.

The way we eliminate unknown cuts and critical pairs is different for data types, codata types, and built-in types like integers. For data and codata types, we use η-expansion, which is guaranteed to be valid due to our choice of evaluation strategies. This is type-preserving and only requires local changes. For built-in types, we need a different way, which is illustrated by applying shrinking to our running example:

Example 43 (Shrinking of Multiplication with Early Return)**.**

```
def rmult(xs :prd List[i64], β :cns Cont) { rmultRec(xs, β, β) }
def rmultRec(xs :prd List[i64], α :cns Cont, β :cns Cont) {
    ⟨xs | case {
      Nil ⇒ ⟨1 | μ̃z.⟨Ret(z) | β⟩⟩,
      Cons(y, ys) ⇒
          if y ≡ 0 {
            ⟨0 | μ̃z.⟨Ret(z) | α⟩⟩
          } else {
            ⟨μγ.rmultRec(ys, α, γ) | case { Ret(r) ⇒ ⟨y * r | μ̃z.⟨Ret(z) | β⟩⟩ }⟩
          }
    }⟩
}
```

First note that compared to the version in Example 40, the cut in the body of rmult has been reduced by a trivial renaming, which illustrates another kind of statement eliminated by our

transformation. Moreover, the type of the covariable β in the parameter list has been changed to a data type `Cont`. The same is true for the covariables α and β in the parameter list of rmultRec. The reason is that we now model the $\tilde{\mu}$-abstractions of integers in critical pairs, i.e., the first-class consumers of integers, with a continuation data type `Cont` having one constructor `Ret`(x :$^{\text{prd}}$ i64). That is, a pattern match of this type is a continuation for integers. This is visible in the critical pair in the **else** branch in the body of rmultRec that was generated by focusing, as discussed in Example 42. The $\tilde{\mu}$-abstraction, which can also be viewed as a continuation for integers, has been turned into a pattern match of type `Cont`. Consequently, all covariables for consumers of integers are now of type `Cont`. Hence, the integers in cuts with such covariables must be wrapped into a constructor `Ret`, as is visible in the cut with the multiplication and also in the cuts with the integer literals in the other branches. To ensure that the argument of the `Ret` constructor is a variable, we bind the result of the multiplication and the integer literals to a name with a $\tilde{\mu}$-binding first. We will see that wrapping integers into constructors does not mean that we box integers, i.e., we do not allocate them on the heap.

5.4 The Lower-Level Intermediate Language AxCut

With the shrunk language, we are already close to a normal form of our intermediate representation for which we can directly generate machine code. The final step before doing so is to translate into a representation with a slightly different structure, which we call AxCut. There are two differences between Shrunk Core and AxCut. First, we eliminate a last redundancy from Shrunk Core. Since data types and codata types are perfectly symmetric in classical sequent calculus, we can use a unified syntax for them. For example, we can use the same notation for binding a constructor to a variable and for binding a destructor to a covariable.

$$\langle K(\sigma) \mid \tilde{\mu}x.s \rangle \quad \& \quad \langle \mu\alpha.s \mid D(\sigma) \rangle \quad \longrightarrow \quad \textbf{let}\ v\ =\ X(\sigma);\ s$$

Here we use v to stand for either a variable or a covariable, and X to stand for either a constructor or a destructor. We could also have used such a notation in Core already, but we decided not to do so, because we think that it is more intuitive to explicitly distinguish between terms of data and codata types at that stage. For code generation, however, we treat data and codata types in exactly the same way, and we think that the unified syntax is more appropriate for low-level concerns. In particular, there are fewer cases to consider during code generation and the syntax of the constructs is more suggestive. Still, for specific examples it may be helpful to consider them from the viewpoint of either data types or codata types to help intuition. In fact, we could take the unification one step further and also collapse data and codata types themselves, resulting in a fully one-sided variant (Girard, 1987) of our language. However, as this does not affect code generation in an essential way and we obtain virtually all benefits from the unified syntax alone, we refrain from doing so here.

The second, and more important difference is that while the structural rules of weakening, contraction, and exchange are implicit in all languages and fragments so far, i.e., variable usage is arbitrary, we make these rules explicit in AxCut. We do so by adding a construct for explicit substitutions (Abadi et al., 1991) **substitute** $[\Gamma := \sigma]$; s. Explicit substitutions define a new environment Γ for the remaining statement s based on the variables in scope. This construct includes reordering variables by using them once in σ in the desired position, corresponding to exchange, duplicating variables by using them multiple times, corresponding to contraction, and dropping variables by not using them at all, corresponding to weakening. The other constructs have to use variables linearly and in the correct order. The details will become clear when we discuss the typing rules of AxCut in 7.2.1.

With the unified syntax of AxCut and after inferring explicit substitutions, our running example of multiplication of a list with early return from Example 43 looks as follows (ignore the colors for now, they only become relevant in 5.5).

Example 44 (Multiplication with Early Return in AxCut)**.**

```
def rmult(xs :prd List[i64], β :cns Cont) {
    substitute [xs := xs, β := β, γ := β]; rmultRec(xs, β, γ)
}
def rmultRec(xs :prd List[i64], α :cns Cont, β :cns Cont) {
    substitute [β := β, α := α, xs := xs];
    switch xs {
        Nil ⇒ substitute [β := β]; lit z ← 1; substitute [z := z, β := β]; invoke β Ret(z),
        Cons(y, ys) ⇒
            if y ≡ 0 {
                substitute [α := α]; lit z ← 0; substitute [z := z, α := α]; invoke α Ret(z)
            } else {
                substitute [ys := ys, α := α, y := y, β := β];
                create γ = (y, β) {
                    Ret(r) ⇒ z ← y * r; substitute [z := z, β := β]; invoke β Ret(z)
                };
                rmultRec(ys, α, γ)
            }
    }
}
```

It is visible how explicit substitutions are used to duplicate the context β in rmult, by using it in two right-hand sides, and to drop one of the contexts in the `Nil` case and the then branch, by not mentioning them on any of the right-hand sides. For the inference of explicit substitutions, we follow garbage-free reference counting (Reinking et al., 2021) in that we aim to drop unneeded variables as early as possible and duplicate variables as late as possible. This is why we drop the contexts in the `Nil` case and the then branch with the substitution before binding the integer (which is not necessary for correctness), instead of only doing so in the substitution afterwards. Moreover, explicit substitutions rearrange the context appropriately before direct jumps to top-level labels or indirect jumps via **invoke** but also for other constructs. This is the case, for example, for the closure environment consumed by the closure γ created by **create** or to bring xs into the right position before pattern matching on it with **switch**. We will explain the details of inferring substitutions in Section 7.3.

Explicit substitutions are helpful for code generation. The explicit information where variables are duplicated and dropped is useful for memory management, especially for the reference counting scheme we use. Explicit reordering of variables corresponds to register moves and allows us to keep register allocation trivial.

5.5 Code Generation

Statements in AxCut are essentially certain normal forms of terms for classical sequent calculus, as we have derived in the previous subsections. Nevertheless, it is surprisingly simple to directly generate code from AxCut. For each statement, we generate a sequence of machine instructions. The variables in the (ordered) typing context of a statement directly correspond to registers (see Section 8.2). We simply assign them left to right, with two adjacent registers per variable. For memory management we use a reference counting scheme, facilitated by our explicit substitutions, which tell us exactly where variables have to be shared and erased. The details are explained in Chapter 8.

To illustrate the generated code, let us look at our running example again.

Example 45 (Code Generation for Multiplication with Early Return)**.** We generate the following machine code for the highlighted parts in Example 44.

```
rmult:
  SHAREBLOCK x6
  mv x8 x6
  mv x9 x7
  j rmultRec
```

```
Nil:
  ERASEBLOCK x6
  li x7 1
  mv x6 x4
  mv temp x7
  mv x7 x5
  mv x5 temp
  jr x7 0
```

```
then:
  ERASEBLOCK x4
  ERASEBLOCK x10
  mv x6 x4
  mv x7 x5
  li x7 0
  ...
  jr x7 0
```

```
Ret:
  RELEASE x6
  lw x9 20 x6
  lw x8 16 x6
  lw x7 12 x6
  mul x11 x7 x5
  mv x6 x8
  mv x7 x9
  mv x5 x11
  jr x7 0
```

The `mv` instruction performs a register move, `li` loads an integer into a register, `lw` loads from memory into a register, `j` jumps to a label, and `jr` jumps to an address in a register with an offset. The machine instructions will be explained in more detail in Section 8.1.

We can see how the context is duplicated in rmult by `SHAREBLOCK` and dropped in the `Nil` case and the then branch by `ERASEBLOCK`, which are macros for code that manipulates the reference count. In the then branch, we also drop the tail of the list (by the second `ERASEBLOCK`) and afterwards move the context that was not dropped into the two registers the erased context was in. At this point, the remaining code for the `Nil` case and for the then branch is the same, apart from the integer loaded (the ellipsis in the then branch stands for the same moves as in the `Nil` case). In the end, both invoke the correct consumer with an indirect jump. When entering the closure bound to γ, we first load the closure environment and release the corresponding memory block with the macro `RELEASE`, if there is no other reference, before performing the code for the body statement. The details will become clear in Sections 8.3 and 8.4.

We have seen how each step in the compilation pipeline brings us closer to the machine code generated above, bridging the gap between complex and structured high-level language constructs in the surface language, and simple unstructured low-level operations on a real machine. We will explain all of these steps in detail in the course of this part of the thesis.

6 Extending the Base Languages

In this chapter, we introduce our surface language Fun (Section 6.1) and our high-level intermediate language Core (Section 6.2). Fun is in many respects an ordinary, expression-oriented, functional programming language with a lambda calculus such as $\lambda_{\mathsf{D}}^{\mathsf{pure}}$ from Section 2.1 at its core. For example, we do not support mutable state or some mechanisms often found in object-oriented programming, such as inheritance. But the language is also more interesting than just the lambda calculus, from which it is distinguished by several features, as already illustrated in Section 5.1. In particular, Fun features control operators, similar to λ_{D} from Section 3.1, but also general data and codata types. These features are not only interesting from the point of view of programmers using them, but also from a compilation perspective. The intermediate language Core is directly based on classical sequent calculus. Its skeleton is the $\lambda\mu\tilde{\mu}$-calculus presented in Section 4.2, but extended in a similar way as for Fun, with arbitrary data and codata types as in System $\mathcal{CD}$ of Downen and Ariola (2020).

After showing how to translate Fun into Core (Section 6.3), we conclude the chapter by briefly considering what challenges the additional features pose when translating back to direct style (Section 6.4).

6.1 The Surface Language Fun

We start with the surface language Fun. We first give the syntax and then the typing rules. The operational semantics is mostly standard and we do not discuss it here, because we are mainly interested in presenting the compiler pipeline.

6.1.1 Syntax

The syntax is presented in Figure 6.1. Terms p, which we also call *producers* in anticipation of the languages that we are going to introduce later, include variables x, non-recursive let-expressions $\mathbf{let}\, x = p_1; p_2$, and calls $f(\sigma)$ to known top-level first-order functions. The latter are defined with a label in the global program Θ with the syntactic form $\mathbf{def}\, f(\Gamma) : \tau \{ p \}$. The top-level function named main, whose return type we assume to be i64, serves as the entry point into the program. All of these features are completely standard. In the declarations of top-level functions, we use typing contexts Γ to specify the number and types of the required arguments. Similarly, instead of writing function calls as $f(p, \ldots, c, \ldots)$, we use the syntactic category of *arguments* σ. Arguments are just lists of producer and consumer arguments; the latter will be introduced in more detail later. We sometimes write e for one argument if we do not want to distinguish producers and consumers. The use of contexts Γ in declarations and of arguments σ in calls allows for a cleaner specification of the typing rules and transformation algorithms.

Three constructs in the language of producers pertain to operations on integers. There are signed 64-bit integer literals n of type i64, arithmetic operations like addition $p_1 + p_2$ and conditional expressions like $\mathbf{if}\, p_1 \equiv 0 \{ p_2 \} \mathbf{else} \{ p_3 \}$. These conditionals can compare integers and evaluate to p_2 if the condition is true, e.g., if p_1 evaluates to 0 here, and to p_3 otherwise. As we will see, these primitives directly compile to corresponding machine instructions. Other primitives would be treated similarly.

Most statically-typed functional programming languages provide a mechanism to define algebraic data types, introduce terms using constructors, and eliminate terms using pattern matching. We write constructors as $K(\sigma)$ and pattern-matching expressions as $p.\mathbf{case} \{ K(\Gamma) \Rightarrow p, \ldots \}$.

Syntax

For variables and names we assume that x, y, $\ldots \in$ Variables, α, β, $\ldots \in$ Covariables, $T \in$ TypeNames, K, D, $X \in$ Tags, and $f \in$ Labels.

Producers/Terms	p	::=	$x \mid \textbf{let}\, x = p;\, p \mid f(\sigma)$
		$\mid$	$n \mid p + p \mid \textbf{if}\, p \equiv 0\, \{\, p \,\}\, \textbf{else}\, \{\, p \,\}$
		$\mid$	$K(\sigma) \mid p.\textbf{case}\, \{\, K(\Gamma) \Rightarrow p, \ldots \,\}$
		$\mid$	$p.D(\sigma) \mid \textbf{new}\, \{\, D(\Gamma) \Rightarrow p, \ldots \,\}$
		$\mid$	$\textbf{label}\, \alpha\, \{\, p \,\} \mid \textbf{goto}\, \alpha\, (p) \mid \textbf{exit}\, p$
Consumers	c	::=	α
Types	τ	::=	$\mathsf{i64} \mid T$
Arguments	σ	::=	$\bullet \mid \sigma,\, p \mid \sigma,\, c$
Typing Contexts	Γ	::=	$\bullet \mid \Gamma,\, x : \tau \mid \Gamma,\, \alpha :^{\mathsf{cns}} \tau$
Declarations	δ	::=	$\textbf{data}\, T\, \{\, K(\Gamma), \ldots \,\} \mid \textbf{codata}\, T\, \{\, D(\Gamma) : \tau, \ldots \,\} \mid \textbf{def}\, f(\Gamma) : \tau\, \{\, p \,\}$
Programs	Θ	::=	$\bullet \mid \Theta,\, \delta$

Figure 6.1: Syntax of Fun.

Data type declarations are specified on the top-level with the syntactic form **data** $T\, \{\, K(\Gamma), \ldots \,\}$. Computations whose result is of a data type are evaluated with call-by-value evaluation order. Similar to the declaration of top-level functions, we use typing contexts Γ to specify the number and types of the parameters of constructors and arguments σ for concrete arguments.

Codata types (Hagino, 1989; Downen et al., 2019) are the dual of data types described in the previous paragraph. In distinction to data types, which use constructors to produce elements of the type and pattern matching to consume elements, codata types use copattern matching (Abel et al., 2013) to produce and destructors to consume elements of codata types. We write invocations of destructors as $p.D(\sigma)$ and copattern-matching expressions as **new** $\{\, D(\Gamma) \Rightarrow p, \ldots \,\}$. Just as for data types, codata type declarations are specified on the top-level with the syntactic form **codata** $T\, \{\, D(\Gamma) : \tau, \ldots \,\}$. In contrast to data types, computations whose result is of a codata type are evaluated with call-by-name evaluation order. We have already seen the `Stream` codata type in Example 37 in Section 5.1, and we have also illustrated the CBN evaluation order in Example 38. We have also mentioned that first-class functions are just another codata type with one `apply` destructor and lambda abstraction corresponds to copattern matching on this destructor. This is illustrated in the following example, which maps a function over a list.

Example 46 (Mapping a Function over a List)**.**

```
data List[A] { Nil, Cons(x : A, xs : List[A]) }
codata Fun[A, B] { apply(x : A) : B }
def map(xs : List[i64], f : Fun[i64, i64]) : List[i64] {
  xs.case { Nil ⇒ Nil, Cons(y, ys) ⇒ Cons(f.apply(y), map(ys, f)) }
}
def increment(xs : List[i64]) : List[i64] { map(xs, new { apply(x) ⇒ x + 1 }) }
```

In this and the following examples, we often use type parameters A in the declaration of types. However, these are only used for convenience in type definitions and there is no polymorphism on the term level. In particular, in all types occurring in terms, these type parameters must be instantiated with monomorphic types in a coherent way. One can think of them as being expanded to one copy of the type definition for each instantiation before type checking. Therefore, type parameters are not included in the formal syntax.

Data and codata types are dual concepts, as will become obvious once we have translated them to our symmetric intermediate language. In fact, as we will see, we use exactly the same

code generation for data and codata types. If we want to refer to a constructor or a destructor, without distinguishing them, we also write X instead of K or D, and similarly, we write v to stand for either a variable or a covariable. The latter will be introduced next.

The two constructs **label** $\alpha\,\{\,p\,\}$ and **goto** $\alpha\,(p)$ are control operators that respectively capture and invoke a program context, or continuation, α. They are similar to **suspend** and **run** in λ_{D}, but in contrast to **suspend**, **label** behaves more like `let/cc` (Reynolds, 1972) and not like Felleisen et al.'s $\mathcal{C}$, as it leaves the captured program context in place instead of removing it. The additional control operator **exit** p, which simply terminates the program, is also as in λ_{D}. The language Fun contains these control operators to illustrate that the sequent calculus is naturally suited as a compilation target for languages that have complex non-local control flow. We use *covariables*, denoted by lower-case Greek letters, to represent the captured program contexts; in Fun they are the only available kind of *consumers* c. These consumers are first-class entities, so it is possible to use them as arguments to constructors and destructors or to pass them to top-level functions. We have already seen why this might be useful in Example 39 in Section 5.1, where we have implemented multiplication of a list of integers with early return.

6.1.2 Typing Rules

Typing in Fun is split into several different judgment forms defined in Figure 6.2. We forgo the formal definitions of most of the rules for well-formedness of programs, declarations, and contexts, only showing the rule for top-level functions. These well-formedness rules ensure that all types and names of top-level functions used in a program are defined and that there are no name clashes. Moreover, for top-level functions we require that the body of a top-level function is well-typed in the environment abstracted by the function. The body is typed with respect to the whole program, which means that top-level functions are allowed to be recursive, even mutually so.

For arguments (to functions, constructors, or destructors), we have the typing judgment $\Theta \mid \Gamma \vdash \sigma : \Gamma'$. Arguments σ are well-typed against an environment Γ', if every producer and consumer in σ is well-typed in context Γ with the type annotated for the corresponding variable or covariable in Γ'.

The forms $\Theta \mid \Gamma \vdash p : \tau$ and $\Theta \mid \Gamma \vdash c :^{\mathsf{cns}} \tau$ are used to type producers and consumers, respectively. As the program Θ is not changed by any rule, and sometimes not even referenced, we will often leave it implicit. Most of the rules are completely standard. Using the rules for arguments makes it particularly easy to express checking arguments for constructors, destructors, and calls of top-level functions. For these rules, the order of (co)variables in the context Γ' from the declaration is important, as we always want to keep the order of arguments in σ as determined by the declaration. The most interesting rules are Label and Goto. In rule Label, the covariable α is bound and made available in the body p. The type of α is the same as the type of p, which is also the type of the whole expression. This is because α represents the current continuation and hence is a consumer for the value produced by evaluating p. Similarly, in rule Goto, the type of the covariable α is the same as that of the body p, but the type τ' of the whole expression is arbitrary. This is because a continuation that was previously bound to α by a **label** is invoked here, and thus the current continuation of the expression, which would consume a value of type τ', is discarded. The type of an **exit** expression is also arbitrary because this terminates the program, but the type of the exit code must be i64. The rules of these control operators show the difference to the control operators **suspend**, **run**, and **exit** in λ_{D}. Since **label** leaves the continuation in place instead of removing it, in contrast to **suspend**, its body is not undelimited. Consequently, **goto** is not undelimited either, as there is a continuation in place when **goto** is invoked, which is ignored, however. Similarly, **exit** in Fun also is not undelimited. While this makes the control operators of Fun somewhat less flexible, they are probably more intuitive. One can often think of **label** as installing a label – embodied by a covariable – at some place in the program, to which one can jump with **goto** (hence the

Well-Formed Declarations $\boxed{\Theta \vdash \delta\ \mathrm{OK}}$

$$\frac{\Theta \mid \Gamma \vdash p : \tau}{\Theta \vdash \mathbf{def}\, f(\Gamma) : \tau\, \{\, p \,\}\ \mathrm{OK}}\ [\textsc{Wf-Def}] \qquad \ldots$$

Argument Typing $\boxed{\Theta \mid \Gamma \vdash \sigma : \Gamma'}$

$$\frac{}{\Theta \mid \Gamma \vdash \bullet : \bullet}\ [\textsc{Arg}_1]$$

$$\frac{\Theta \mid \Gamma \vdash \sigma : \Gamma' \qquad \Theta \mid \Gamma \vdash p : \tau}{\Theta \mid \Gamma \vdash \sigma, p : \Gamma', x : \tau}\ [\textsc{Arg}_2] \qquad \frac{\Theta \mid \Gamma \vdash \sigma : \Gamma' \qquad \Theta \mid \Gamma \vdash c :^{\mathsf{cns}} \tau}{\Theta \mid \Gamma \vdash \sigma, c : \Gamma', \alpha :^{\mathsf{cns}} \tau}\ [\textsc{Arg}_3]$$

Producer Typing $\boxed{\Theta \mid \Gamma \vdash p : \tau}$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ [\textsc{Var}] \qquad \frac{}{\Gamma \vdash n : \mathsf{i64}}\ [\textsc{Lit}] \qquad \frac{\Gamma \vdash p_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash p_2 : \tau_2}{\Gamma \vdash \mathbf{let}\, x = p_1;\, p_2 : \tau_2}\ [\textsc{Let}]$$

$$\frac{\Gamma \vdash p_1 : \mathsf{i64} \qquad \Gamma \vdash p_2 : \mathsf{i64}}{\Gamma \vdash p_1 + p_2 : \mathsf{i64}}\ [\textsc{Plus}] \qquad \frac{\Gamma \vdash p : \mathsf{i64} \qquad \Gamma \vdash p_1 : \tau \qquad \Gamma \vdash p_2 : \tau}{\Gamma \vdash \mathbf{if}\, p \equiv 0\, \{\, p_1 \,\}\, \mathbf{else}\, \{\, p_2 \,\} : \tau}\ [\textsc{IfZ}]$$

$$\frac{\Gamma, \alpha :^{\mathsf{cns}} \tau \vdash p : \tau}{\Gamma \vdash \mathbf{label}\, \alpha\, \{\, p \,\} : \tau}\ [\textsc{Label}] \qquad \frac{\Gamma \vdash p : \tau \qquad \Gamma \vdash \alpha :^{\mathsf{cns}} \tau}{\Gamma \vdash \mathbf{goto}\, \alpha\, (p) : \tau'}\ [\textsc{Goto}] \qquad \frac{\Gamma \vdash p : \mathsf{i64}}{\Gamma \vdash \mathbf{exit}\, p : \tau}\ [\textsc{Exit}]$$

$$\frac{\mathbf{data}\, T\, \{\ldots, K(\Gamma'), \ldots\} \in \Theta \qquad \Theta \mid \Gamma \vdash \sigma : \Gamma'}{\Theta \mid \Gamma \vdash K(\sigma) : T}\ [\textsc{Ctor}]$$

$$\frac{\mathbf{data}\, T\, \{\, K_1(\Gamma_1), \ldots\} \in \Theta \qquad \Gamma \vdash p : T \qquad \forall i : \Gamma, \Gamma_i \vdash p_i : \tau}{\Theta \mid \Gamma \vdash p.\mathbf{case}\, \{\, K_1(\Gamma_1) \Rightarrow p_1, \ldots\} : \tau}\ [\textsc{Case}]$$

$$\frac{\mathbf{codata}\, T\, \{\ldots, D(\Gamma') : \tau, \ldots\} \in \Theta \qquad \Gamma \vdash p : T \qquad \Theta \mid \Gamma \vdash \sigma : \Gamma'}{\Theta \mid \Gamma \vdash p.D(\sigma) : \tau}\ [\textsc{Dtor}]$$

$$\frac{\mathbf{codata}\, T\, \{\, D_1(\Gamma_1) : \tau_1, \ldots\} \in \Theta \qquad \forall i : \Gamma, \Gamma_i \vdash p_i : \tau_i}{\Theta \mid \Gamma \vdash \mathbf{new}\, \{\, D_1(\Gamma_1) \Rightarrow p_1, \ldots\} : T}\ [\textsc{New}]$$

$$\frac{\mathbf{def}\, f(\Gamma') : \tau\, \{\ldots\} \in \Theta \qquad \Theta \mid \Gamma \vdash \sigma : \Gamma'}{\Theta \mid \Gamma \vdash f(\sigma) : \tau}\ [\textsc{Call}]$$

Consumer Typing $\boxed{\Theta \mid \Gamma \vdash c :^{\mathsf{cns}} \tau}$

$$\frac{\alpha :^{\mathsf{cns}} \tau \in \Gamma}{\Gamma \vdash \alpha :^{\mathsf{cns}} \tau}\ [\textsc{Covar}]$$

Figure 6.2: Typing rules for Fun.

names). The typing rule for covariables, the only kind of consumer, simply checks whether the covariable is in the context.

6.2 The High-Level Intermediate Language Core

Next, we introduce the high-level intermediate language Core. As before, we present the syntax and typing rules, but leave out the operational semantics.

6.2.1 Syntax

The syntax of Core is presented in Figure 6.3. In our compilation pipeline, we perform two transformations on Core, each of which targets a smaller fragment of the calculus, before we translate the final fragment to a system called AxCut. The latter has a slightly different structure that is more well-suited for code generation. All of these languages share the part of the syntax given in the upper part of Figure 6.3.

This common part of the syntax consists of types and typing contexts, as well as the declarations making up programs. It coincides mostly with the corresponding syntax of the surface language Fun. A minor difference is that in typing contexts, we now annotate typing of producer bindings with prd, dually to how typing of consumer bindings is annotated with cns. We also call this annotation the *chirality* of the expression, which is derived from the Greek word for hand, since in the original notation of sequent calculus, producers are introduced on the right-hand side of a sequent, while consumers are introduced on the left-hand side. As in our presentation of the $\lambda\mu\tilde{\mu}$-calculus in Section 4.2, we do not follow this original notation, which uses separate contexts for producers and consumers, but rather collect all bindings in one context, as we did in Fun, and distinguish them with the chirality annotation. The crucial difference to Fun, however, is that top-level functions and destructors in declarations of codata types no longer have return types. This makes declarations of codata types exactly dual to declarations of data types. That is, data and codata types syntactically only differ in their *polarity*, where data types are also said to be positive and codata types are negative (Andreoli, 1992; Girard, 1991, 1993; Downen and Ariola, 2020, 2021; Downen, 2017).

The only non-terminals not defined in the upper part of 6.3 are statements s, which constitute the bodies of top-level functions. These statements vary with the language in each compilation phase, and the four different variants can be found in the lower part of 6.3, along with the additional syntactic categories needed to define statements in each case. We can observe the following differences between the languages, which we will study in detail in subsequent sections.

- The language Focused Core differs from Core at the positions marked with ▢ where we enforce that in argument positions only variables or covariables are used instead of arbitrary producers or consumers.
- The language Shrunk Core is obtained from Focused Core by inlining producers p and consumers c in the cut $\langle p \mid c \rangle$ and eliminating redundant constructs to shrink the language. Since we have six producers and four consumers in Focused Core, this would result in 24 combinations, but 14 of those are either precluded by typing or can be eliminated. The remaining ten combinations make up the majority of the statements of Shrunk Core.
- While Focused Core is a fragment of Core and Shrunk Core is a fragment of Focused Core, the calculus AxCut has a slightly different structure. In particular, dual term constructs in Shrunk Core are collapsed into one single form using a unified notation. For example, the two statements $\langle K(\sigma) \mid \tilde{\mu}x.s \rangle$ and $\langle \mu\alpha.s \mid D(\sigma) \rangle$ for binding constructors and destructors are combined into $\textbf{let}\ v = X(\sigma);\ s$. Moreover, there is an additional construct **substitute** to be explained later.

In the syntax of terms for Core, we can see that while Fun mostly consisted of producers with the only consumers being covariables, terms in Core are divided into the three categories of

Syntax of Types, Contexts, Declarations and Programs

Types	τ	::=	$\mathsf{i64} \mid T$
Chirality	χ	::=	$\mathsf{prd} \mid \mathsf{cns}$
Typing Contexts	Γ	::=	$\bullet \mid \Gamma,\, v :^{\chi} \tau$
Polarity	π	::=	$\mathbf{data} \mid \mathbf{codata}$
Declarations	δ	::=	$\pi\ T\,\{\, X(\Gamma), \ldots \,\} \mid \mathbf{def}\, f(\Gamma)\,\{\, s \,\}$
Programs	Θ	::=	$\bullet \mid \Theta,\, \delta$

Syntax of Terms

Core

Producers	p	::=	$x \mid \mu\alpha.s \mid K(\sigma) \mid \mathbf{new}\,\{\, D(\Gamma) \Rightarrow s, \ldots \,\} \mid n \mid p + p$
Consumers	c	::=	$\alpha \mid \tilde{\mu}x.s \mid D(\sigma) \mid \mathbf{case}\,\{\, K(\Gamma) \Rightarrow s, \ldots \,\}$
Statements	s	::=	$\langle p \mid c \rangle \mid \mathbf{if}\, p \equiv 0\,\{\, s \,\}\,\mathbf{else}\,\{\, s \,\} \mid f(\sigma) \mid \mathbf{exit}\, p$
Arguments	σ	::=	$\bullet \mid \sigma,\, p \mid \sigma,\, c$

Focused Core (Section 7.1.1)

Producers	p	::=	$x \mid \mu\alpha.s \mid K(\sigma) \mid \mathbf{new}\,\{\, D(\Gamma) \Rightarrow s, \ldots \,\} \mid n \mid x + x$
Consumers	c	::=	$\alpha \mid \tilde{\mu}x.s \mid D(\sigma) \mid \mathbf{case}\,\{\, K(\Gamma) \Rightarrow s, \ldots \,\}$
Statements	s	::=	$\langle p \mid c \rangle \mid \mathbf{if}\, x \equiv 0\,\{\, s \,\}\,\mathbf{else}\,\{\, s \,\} \mid f(\sigma) \mid \mathbf{exit}\, x$
Arguments	σ	::=	$\bullet \mid \sigma,\, x \mid \sigma,\, \alpha$

Shrunk Core (Section 7.1.2)

Statements	s	::=	$\langle K(\sigma) \mid \tilde{\mu}x.s \rangle \mid \langle \mu\alpha.s \mid D(\sigma) \rangle \mid \langle K(\sigma) \mid \alpha \rangle \mid \langle x \mid D(\sigma) \rangle$
		$\mid$	$\langle x \mid \mathbf{case}\,\{\, K(\Gamma) \Rightarrow s, \ldots \,\} \rangle \mid \langle \mu\alpha.s \mid \mathbf{case}\,\{\, K(\Gamma) \Rightarrow s, \ldots \,\} \rangle$
		$\mid$	$\langle \mathbf{new}\,\{\, D(\Gamma) \Rightarrow s, \ldots \,\} \mid \alpha \rangle \mid \langle \mathbf{new}\,\{\, D(\Gamma) \Rightarrow s, \ldots \,\} \mid \tilde{\mu}x.s \rangle$
		$\mid$	$\langle n \mid \tilde{\mu}x.s \rangle \mid \langle x + x \mid \tilde{\mu}x.s \rangle \mid \mathbf{if}\, x \equiv 0\,\{\, s \,\}\,\mathbf{else}\,\{\, s \,\} \mid f(\sigma) \mid \mathbf{exit}\, x$
Arguments	σ	::=	$\bullet \mid \sigma,\, x \mid \sigma,\, \alpha$

AxCut (Section 7.2)

(Co)Variables	v	::=	$x \mid \alpha$
Statements	s	::=	$\mathbf{let}\, v = X(\sigma);\, s \mid \mathbf{invoke}\, v\, X(\sigma)$
		$\mid$	$\mathbf{switch}\, v\,\{\, X(\Gamma) \Rightarrow s, \ldots \,\} \mid \mathbf{create}\, v = \Gamma\,\{\, X(\Gamma) \Rightarrow s, \ldots \,\};\, s$
		$\mid$	$\mathbf{lit}\, v \leftarrow n;\, s \mid v \leftarrow v + v;\, s \mid \mathbf{if}\, v \equiv 0\,\{\, s \,\}\,\mathbf{else}\,\{\, s \,\}$
		$\mid$	$f(\sigma) \mid \mathbf{substitute}\,[\Gamma := \sigma];\, s \mid \mathbf{exit}\, v$
Arguments	σ	::=	$\bullet \mid \sigma,\, v$

Figure 6.3: Syntax of intermediate languages.

producers, consumers, and statements, with producers and consumers being almost exactly symmetric. Producers are variables, μ-bindings, constructors of data types, and copattern matches for codata types, and moreover, integer literals and arithmetic expressions. Dually, consumers are covariables, $\tilde{\mu}$-bindings, destructors of codata types, and pattern matches for data types. The μ- and $\tilde{\mu}$-bindings allow us to abstract over a producer or a consumer in a statement. These constructs are new compared to Fun and, as we will see, can be used to express **let**-bindings and the control operators **label** and **goto**.

A crucial difference to Fun is that the scrutinee, i.e., the component of a construct scrutinized or acted upon, is no longer part of pattern matches and destructor calls. Rather, the two parts of such expressions, one consumer and one producer, can now meet in a cut where they interact. Cuts, and more generally statements, are thus the place where computation happens. They do not return anything, but rather drive the evaluation of programs. This is what makes a language based on sequent calculus surprisingly close to machine code, as we will see. In addition to cuts, we also treat conditionals and calls of top-level functions as statements.

As an example, consider again the mapping of a function over a list discussed in Example 46.

Example 47 (Mapping a Function over a List in Core)**.** Instead of having a return type B, the destructor `apply` has an additional parameter α for a consumer of B. Similarly, the top-level functions map and increment have a parameter β for a consumer of type `List`[i64] instead of a corresponding return type.

```
data List[A] { Nil, Cons(x :prd A, xs :prd List[A]) }
codata Fun[A, B] { apply(x :prd A, β :cns B) }
def map(xs :prd List[i64], f :prd Fun[i64, i64], α :cns List[i64]) {
    ⟨xs | case { Nil ⇒ ⟨Nil | α⟩,
                 Cons(y, ys) ⇒ ⟨Cons(μβ.⟨f | apply(y, β)⟩, μγ.map(ys, f, γ)) | α⟩ }⟩
}
def increment(xs :prd List[i64], α :cns List[i64]) {
    map(xs, new { apply(x, β) ⇒ ⟨x + 1 | β⟩ }, α)
}
```

Inside the copattern match in increment, the incremented number $x + 1$ is cut with the consumer β abstracted by the destructor. Inside the pattern match in map, the constructors of the list are cut with the outer consumer α abstracted by map. The two arguments of the `Cons` constructor themselves abstract a covariable, which they use in their respective body statement.

Apart from the fact that the computations of the `Cons` constructor are not lifted out of their positions to make the evaluation order explicit, this is quite similar to what the example would look like in CPS and we will see this scheme again when we discuss the translation from Fun to Core in Section 7.3. The lifting of computations out of argument position is a separate step, which we will discuss in 7.1.1. This is facilitated by the μ-abstractions, which allow us to capture the current context without changing the nesting structure, because, in contrast to lambdas, they are not generally values and hence do not hide the computation in their body.

6.2.2 Typing Rules

We will now discuss the typing rules for Core. As Focused Core and Shrunk Core are fragments of Core, there is no need to change the typing rules for them. However, the typing rules for AxCut are somewhat different and we will discuss them in Section 7.2.

As before, there are a number of judgment forms used to type expressions in Core, whose rules are presented in Figure 6.4. Most of them are analogous to Fun, but since we now have an additional syntactic category of statements, there is also a typing judgment for it, written $\Theta \mid \Gamma \vdash s$. Moreover, since producers and consumers are symmetric in Core, so are their typing rules. We will again often leave the program Θ implicit.

Well-Formed Declarations $\boxed{\Theta \vdash \delta\ \textsc{Ok}}$

$$\frac{\Theta \mid \Gamma \vdash s}{\Theta \vdash \mathbf{def}\, f(\Gamma)\,\{\, s \,\}\ \textsc{Ok}}\ [\textsc{Wf-Def}] \qquad \ldots$$

Producer Typing $\boxed{\Theta \mid \Gamma \vdash p :^{\mathsf{prd}} \tau}$

$$\frac{\Gamma,\, \alpha :^{\mathsf{cns}} \tau \vdash s}{\Gamma \vdash \mu\alpha.s :^{\mathsf{prd}} \tau}\ [\textsc{Act-R}] \qquad \frac{\mathbf{codata}\ T\,\{\, D_1(\Gamma_1), \ldots \,\} \in \Theta \qquad \forall i : \Gamma, \Gamma_i \vdash s_i}{\Theta \mid \Gamma \vdash \mathbf{new}\,\{\, D_1(\Gamma_1) \Rightarrow s_1, \ldots \,\} :^{\mathsf{prd}} T}\ [\textsc{New}]$$

The rules Var, Ctor, Lit and Plus are identical to the ones in Figure 6.2.

Consumer Typing $\boxed{\Theta \mid \Gamma \vdash c :^{\mathsf{cns}} \tau}$

$$\frac{\Gamma,\, x :^{\mathsf{prd}} \tau \vdash s}{\Gamma \vdash \tilde{\mu}x.s :^{\mathsf{cns}} \tau}\ [\textsc{Act-L}] \qquad \frac{\mathbf{data}\ T\,\{\, K_1(\Gamma_1), \ldots \,\} \in \Theta \qquad \forall i : \Gamma, \Gamma_i \vdash s_i}{\Theta \mid \Gamma \vdash \mathbf{case}\,\{\, K_1(\Gamma_1) \Rightarrow s_1, \ldots \,\} :^{\mathsf{cns}} T}\ [\textsc{Case}]$$

$$\frac{\mathbf{codata}\ T\,\{\ldots,\, D(\Gamma'), \ldots\} \in \Theta \qquad \Theta \mid \Gamma \vdash \sigma : \Gamma'}{\Theta \mid \Gamma \vdash D(\sigma) :^{\mathsf{cns}} T}\ [\textsc{Dtor}]$$

The rule Covar is identical to the one in Figure 6.2.

Statement Typing $\boxed{\Theta \mid \Gamma \vdash s}$

$$\frac{\Gamma \vdash p :^{\mathsf{prd}} \tau \qquad \Gamma \vdash c :^{\mathsf{cns}} \tau}{\Gamma \vdash \langle p \mid c \rangle}\ [\textsc{Cut}] \qquad \frac{\Gamma \vdash p :^{\mathsf{prd}} \mathsf{i64} \qquad \Gamma \vdash s_1 \qquad \Gamma \vdash s_2}{\Gamma \vdash \mathbf{if}\, p \equiv 0\,\{\, s_1 \,\}\,\mathbf{else}\,\{\, s_2 \,\} : \tau}\ [\textsc{IfZ}]$$

$$\frac{\mathbf{def}\, f(\Gamma')\,\{\ldots\} \in \Theta \qquad \Theta \mid \Gamma \vdash \sigma : \Gamma'}{\Theta \mid \Gamma \vdash f(\sigma)}\ [\textsc{Call}] \qquad \frac{\Gamma \vdash p :^{\mathsf{prd}} \mathsf{i64}}{\Gamma \vdash \mathbf{exit}\, p}\ [\textsc{Exit}]$$

Figure 6.4: Typing rules for Core.

As the rules for arguments are virtually identical to the ones for Fun in Figure 6.2, we leave them out here. The same is true for several of the rules for producers and the rule for covariables, as indicated in the figure. The only change in rule Wf-Def for well-formedness of top-level functions is that the body is a statement now instead of a producer. Similarly, the only change in rule New for copattern matches is that the bodies of the branches are statements now.

The only rule for producers that has no equivalent in Fun is the right activation rule Act-R, which types μ-abstractions. A μ-abstraction turns a statement into a producer by abstracting over a consumer in it. Dually, a $\tilde{\mu}$-abstraction turns a statement into a consumer by abstracting over a producer in it, as is visible in rule Act-L. In the rules Dtor for destructors and Case for pattern matches, we can see that since these consumers have been separated from their scrutinees, typing for destructors is now completely dual to constructors and typing for pattern matches is completely dual to copattern matches.

In the typing judgment for statements, we can observe that they are typed without a return type, as they represent computations. For typing of cuts in rule Cut, we require the producer and the consumer to have the same type, which ensures that they are compatible and can be reduced. As the branches of conditionals are now statements, so are the conditionals as a whole. Moreover, since top-level functions do not return anymore, calls to them are statements as well. Finally, termination of a program with **exit** is naturally a statement, as could already be seen in Fun where its return type was arbitrary.

6.3 Translating Fun to Core

In this section, we show how to translate programs from the surface language Fun into the intermediate language Core. The translation shares some similarities with translations into CPS. To avoid administrative redexes, we employ techniques found in one-pass, first-order, and compositional CPS translations (Danvy and Nielsen, 2003). As we need typing information in a few places, the translation is defined over typing derivations. However, for ease of presentation, we do not make this explicit in the definition. The translation preserves typeability (Theorem 49).

6.3.1 A One-Pass, First-Order, Compositional Translation

We start with the translation of top-level definitions, which works in essentially the same way as for a CPS translation. The translated top-level function abstracts a fresh covariable α whose type is the return type, and which is then used in the translation of the body, explained shortly.

$$\mathcal{C}[\![\,\mathbf{def}\,f(\Gamma)\,:\,\tau\,\{\,p\,\}\,]\!] \;=\; \mathbf{def}\,f(\Gamma,\,\alpha\,:^{\mathsf{cns}}\,\tau)\,\{\,\mathcal{C}[\![\,p\,]\!]_{\alpha}\,\} \qquad (\alpha\,\text{fresh})$$

Similar to the current continuation in a CPS translation, this covariable stands for the consumer into which the result of the function call is supposed to be fed, i.e., the reified program context at this place. The only exception is the main function, which is the entry point of the program. We do not abstract an additional consumer parameter for it, but translate its body with a top-level consumer $\tilde{\mu}x.\mathbf{exit}\,x$, which terminates the program with the value it receives.

$$\mathcal{C}[\![\,\mathbf{def}\,\mathsf{main}(\Gamma)\,:\,\mathsf{i64}\,\{\,p\,\}\,]\!] \;=\; \mathbf{def}\,\mathsf{main}(\Gamma)\,\{\,\mathcal{C}[\![\,p\,]\!]_{\tilde{\mu}x.\mathbf{exit}\,x}\,\}$$

The parts of the translation concerning typing are mostly trivial. More precisely, as types are either the built-in i64 or names of user-defined types, which are the same in both languages, these do not have to be translated at all. Similarly, variables and covariables are also the same in both languages, so we do not have to translate typing contexts either (apart from the trivial prd-annotation for producers). The same is true for declarations of data types, but codata declarations are different in Core, in that they no longer have a return type for destructors. Instead, each destructor is endowed with a fresh covariable argument α_i with this type during the translation, similar to top-level definitions.

$$\mathcal{C}[\![\,\mathbf{codata}\,T\,\{\,D_1(\Gamma_1)\,:\,\tau_1,\,\ldots\,\}\,]\!] \;=\; \mathbf{codata}\,T\,\{\,D_1(\Gamma_1,\,\alpha_1\,:^{\mathsf{cns}}\,\tau_1),\,\ldots\,\} \qquad (\alpha_1,\,\ldots\,\text{fresh})$$

This covariable again stands for the current context at the place of a destructor invocation.

The translation for terms is shown in Figure 6.5. There is no translation of consumers in the figure, as the only consumers in Fun are covariables, which do not have to be translated. For ease of presentation, we do not state freshness conditions explicitly, but rather always implicitly assume each (co)variable appearing in a translated term that is not already present in the source term to be fresh. To avoid administrative redexes, the translation for producers is split into the two mutually recursive functions $\mathcal{C}[\![\cdot]\!]$, which translates a producer in Fun to a producer in Core, and $\mathcal{C}[\![\cdot]\!]_{\odot}$, which takes a consumer in Core as an additional argument and returns a statement. The intuition is that $\mathcal{C}[\![p]\!]_c$ should be the same as $\langle \mathcal{C}[\![p]\!] \mid c \rangle$, but whenever this would result in a redex, for example, if $\mathcal{C}[\![p]\!]$ has the form $\mu\alpha.s$, we inline the consumer and hence avoid generating the administrative redex in the first place. For this to be sound, the consumer should be a covalue. This is similar to how one-pass, first-order, compositional translations into CPS avoid administrative redexes by taking the current continuation as an additional argument. As with naive CPS translations, administrative redexes obscure the structure of the translated program and increase its size significantly. This not only makes the code harder to read and understand, but also makes later optimization passes more difficult and expensive to perform. To illustrate how the translation avoids administrative redexes, consider the following example.

Example 48 (Translation of Let Bindings)**.** A naive translation might introduce too many μ-abstractions, such as in

$$\begin{aligned}&[\![\textsf{let}\, x \;=\; v.\textsf{case}\,\{\,\mathtt{Tup}(y_1,\, y_2) \Rightarrow y_1 \;+\; y_2\,\};\, f(x)\,]\!]\\ &=\; \mu\alpha.\langle \mu\beta.\langle v \mid \textsf{case}\,\{\,\mathtt{Tup}(y_1,\, y_2) \Rightarrow \langle y_1 \;+\; y_2 \mid \beta\rangle\,\}\rangle \mid \tilde{\mu}x.\langle \mu\gamma.f(x,\, \gamma) \mid \alpha\rangle\rangle\end{aligned}$$

Here, the outer μ-binding abstracts the outer consumer α. The **let** directly corresponds to a $\tilde{\mu}$-abstraction, which is cut with the bound pattern match. The pattern match needs this consumer in the body of its branch to continue execution. It is hence wrapped into a μ-abstraction that binds the consumer to the covariable β, turning the pattern match into a producer. Similarly, the call of the top-level function f also needs an additional consumer argument, as it does not return in Core. It is hence also wrapped into a μ-abstraction to turn it into a producer, which is then cut with the outer consumer α. However, these two cuts are administrative and could be reduced right away. In contrast, our optimized translation directly yields

$$\begin{aligned}&\mathcal{C}[\![\textsf{let}\, x \;=\; v.\textsf{case}\,\{\,\mathtt{Tup}(y_1,\, y_2) \Rightarrow y_1 \;+\; y_2\,\};\, f(x)\,]\!]\\ &=\; \mu\alpha.\langle v \mid \textsf{case}\,\{\,\mathtt{Tup}(y_1,\, y_2) \Rightarrow \langle y_1 \;+\; y_2 \mid \tilde{\mu}x.f(x,\, \alpha)\rangle\,\}\rangle\end{aligned}$$

which avoids the administrative cuts in the first place.

An important difference to CPS translations is that in most cases we do not lift terms out of argument positions, but rather translate them in place by the function $\mathcal{C}[\![\cdot]\!]$ without an additional consumer argument. To do so, we use μ-abstractions locally, i.e., around the term in argument position, to abstract over the current context. The bound covariable is then used as consumer input to translate the argument term and the resulting Core statement becomes the body of the μ-abstraction, turning the statement into a producer in Core. This also means that in contrast to CPS translations, we do not make the evaluation order explicit. The lifting of terms is performed later in a separate step, which we will discuss in Section 7.1.1.

While it may seem that we introduce lots of administrative redexes this way, it is important to note that we never introduce μ-abstractions for *trivial* terms, i.e., integer literals, arithmetic operations, variables, constructors, and copattern matches. These terms have a corresponding producer in Core, and we can in most cases just recursively translate the subterms without a consumer input, where the translation of arguments in constructors simply amounts to translating each argument individually. The only somewhat more interesting case is the one for copattern matches. Just as we saw in the translation of codata declarations, each destructor abstracts an additional consumer parameter α_i, standing for the program context when this

destructor is called. This α_i is then used to translate the body of the corresponding branch into a statement. We argue that by translating trivial terms without a μ-abstraction, we avoid most administrative redexes. To see why, we note that nested calls of top-level functions or nested destructor invocations do not really result in administrative redexes. As an example, consider the translation of the following nested call of top-level functions:

$$\mathcal{C}[\![\, f_1(f_2(x))\,]\!]_c \;=\; f_1(\mu\alpha.f_2(x,\, \alpha),\; c)$$

The μ-abstraction will later be lifted by our focusing transformation $\mathcal{F}(\,\cdot\,)$ (cf. 7.1.1), resulting in:

$$\mathcal{F}(\mathcal{C}[\![\, f_1(f_2(x))\,]\!]_c) \;=\; \langle \mu\alpha.f_2(x,\, \alpha) \mid \tilde{\mu}y.f_1(y,\, c)\rangle$$

A translation trying to inline the consumer would instead put the consumer containing the outer call into argument position:

$$\mathcal{C}[\![\, f_1(f_2(x))\,]\!]_c \;=\; f_2(x,\, \tilde{\mu}y.f_1(y,\, c))$$

But our focusing transformation would also lift the $\tilde{\mu}$-abstraction, as we will see in 7.1.1, resulting in the same statement as above. In general, inlining consumers in such nested calls only puts the consumers into a different scope, but does not really result in administrative redexes. For some other non-trivial terms, such as pattern matches, we might generate administrative redexes when they occur in argument positions, but we argue that real programs are not typically written this way. They instead bind such terms to a name first, for which we do avoid the administrative redex, as we have seen in Example 48.

The translation of the control operator **label** is a special case. While it does not seem to have an equivalent in Core, it can actually be translated directly to a μ-binding. Both constructs abstract a covariable α to which the reified program context is bound. The body of the **label** can then be translated with this α as consumer input to become the body of the μ-abstraction. This μ-abstraction is never administrative, however, as it directly expresses the control operator present in the original program. In fact, μ-abstractions had originally been conceived as control operators in Parigot (1992)'s $\lambda\mu$-calculus, before they became part of the $\lambda\mu\tilde{\mu}$-calculus.

When translating a producer in Fun that is not in argument position, we make use of the additional consumer input in function $\mathcal{C}[\![\,\cdot\,]\!]_{\odot}$ to translate it to a statement in Core. Those producers that have an equivalent in Core are translated in the same way as in argument position, i.e., without an additional consumer input, and are then simply cut with the given consumer. For the other constructs, we try to avoid cuts in order not to generate administrative redexes.

The cut that we generate for the control operator **label** may look as though it could also be reduced immediately by inlining the consumer input c for the covariable α, but there are reasons not to do so. For one, as explained above, this cut is not administrative. Moreover, reducing this cut may duplicate c since α may occur multiple times in the body. In the translation for the other control operators, we ignore the consumer input. For **exit**, we simply translate the body without consumer input, as it terminates the program anyway. Similarly, a **goto** jumps to a reified program context bound to its covariable α, so instead of the given consumer input, we use α to translate the body.

In the translation of a **let**-expression, the binding x and the remaining producer p_2 become a $\tilde{\mu}$-binding, which serves as a consumer for the translation of the bound producer p_1. Here, we have to distinguish two cases. As mentioned above, the consumer can only be inlined safely if it is a covalue. However, $\tilde{\mu}$-abstractions are only covalues if they are not of a codata type. Thus, if p_1 is of a codata type, we simply cut its translation with the consumer. Otherwise, we can avoid generating administrative μ-bindings by passing the $\tilde{\mu}$-abstraction as consumer input to the translation of p_1.

The translations of destructor invocations and pattern matches both proceed by translating the scrutinee, with the new consumer input being the other part of the construct. For destructor

$\mathcal{C}[\![\,\cdot\,]\!] : \mathit{Producer}_{\mathsf{Fun}} \to \mathit{Producer}_{\mathsf{Core}}$

$$
\begin{array}{lcllcl}
\mathcal{C}[\![\,n\,]\!] & = & n & \mathcal{C}[\![\,p_1 \;+\; p_2\,]\!] & = & \mathcal{C}[\![\,p_1\,]\!] + \mathcal{C}[\![\,p_2\,]\!] \\
\mathcal{C}[\![\,x\,]\!] & = & x & \mathcal{C}[\![\,\mathsf{new}\,\{\,D_1(\Gamma_1) \Rightarrow p_1,\,\ldots\,\}\,]\!] & = & \mathsf{new}\,\{\,D_1(\Gamma_1,\,\alpha_1) \Rightarrow \mathcal{C}[\![\,p_1\,]\!]_{\alpha_1},\,\ldots\,\} \\
\mathcal{C}[\![\,K(\sigma)\,]\!] & = & K(\mathcal{C}[\![\,\sigma\,]\!]) & \mathcal{C}[\![\,\mathsf{label}\,\alpha\,\{\,p\,\}\,]\!] & = & \mu\alpha.\mathcal{C}[\![\,p\,]\!]_{\alpha}
\end{array}
$$

$$\mathcal{C}[\![\,p\,]\!] \;=\; \mu\alpha.\mathcal{C}[\![\,p\,]\!]_{\alpha} \quad \text{for all other producers } p$$

$\mathcal{C}[\![\,\cdot\,]\!]_{\odot} : \mathit{Producer}_{\mathsf{Fun}} \times \mathit{Consumer}_{\mathsf{Core}} \to \mathit{Statement}_{\mathsf{Core}}$

$$
\begin{array}{lcllcl}
\mathcal{C}[\![\,n\,]\!]_c & = & \langle n \mid c \rangle & \mathcal{C}[\![\,p_1 \;+\; p_2\,]\!]_c & = & \langle \mathcal{C}[\![\,p_1\,]\!] + \mathcal{C}[\![\,p_2\,]\!] \mid c \rangle \\
\mathcal{C}[\![\,x\,]\!]_c & = & \langle x \mid c \rangle & & & \\
\mathcal{C}[\![\,f(\sigma)\,]\!]_c & = & f(\mathcal{C}[\![\,\sigma\,]\!],\, c) & \mathcal{C}[\![\,\mathsf{exit}\,p\,]\!]_c & = & \mathsf{exit}\,\mathcal{C}[\![\,p\,]\!] \\
\mathcal{C}[\![\,\mathsf{label}\,\alpha\,\{\,p\,\}\,]\!]_c & = & \langle \mu\alpha.\mathcal{C}[\![\,p\,]\!]_{\alpha} \mid c \rangle & \mathcal{C}[\![\,\mathsf{goto}\,\alpha\,(p)\,]\!]_c & = & \mathcal{C}[\![\,p\,]\!]_{\alpha} \\
\mathcal{C}[\![\,\mathsf{let}\,x \;=\; p_1;\,p_2\,]\!]_c & = & \langle \mathcal{C}[\![\,p_1\,]\!] \mid \tilde{\mu}x.\mathcal{C}[\![\,p_2\,]\!]_c \rangle & \text{if } p_1 : \mathsf{codata}\; T\,\{\,\ldots\,\} & & \\
\mathcal{C}[\![\,\mathsf{let}\,x \;=\; p_1;\,p_2\,]\!]_c & = & \mathcal{C}[\![\,p_1\,]\!]_{\tilde{\mu}x.\mathcal{C}[\![\,p_2\,]\!]_c} & \text{otherwise} & &
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{C}[\![\,p.D(\sigma)\,]\!]_c & = & \mathcal{L}_v(\mathcal{C}[\![\,\sigma\,]\!])[\lambda as.\mathcal{C}[\![\,p\,]\!]_{D(as,\,c)}] \\
\mathcal{C}[\![\,\mathsf{new}\,\{\,D_1(\Gamma_1) \Rightarrow p_1,\,\ldots\,\}\,]\!]_c & = & \langle \mathsf{new}\,\{\,D_1(\Gamma_1,\,\alpha_1) \Rightarrow \mathcal{C}[\![\,p_1\,]\!]_{\alpha_1},\,\ldots\,\} \mid c \rangle \\
\mathcal{C}[\![\,p.\mathsf{case}\,\{\,K_1(\Gamma_1) \Rightarrow p_1,\,\ldots\,\}\,]\!]_c & = & \mathcal{C}[\![\,p\,]\!]_{\mathsf{case}\,\{\,K_1(\Gamma_1) \Rightarrow \mathcal{C}[\![\,p_1\,]\!]_{c_0},\,\ldots\,\}} \\
\qquad \text{where} & & c_0 \equiv \tilde{\mu}x.j(\Gamma) \\
\qquad \text{with} & & \mathsf{def}\; j(\Gamma)\,\{\,\langle x \mid c \rangle\,\} \\
\qquad \text{and} & & \Gamma \;:=\; \mathsf{freeVars}(c),\; x :^{\mathsf{prd}} \tau \qquad (\text{where } c :^{\mathsf{cns}} \tau) \\
\mathcal{C}[\![\,\mathsf{if}\; p_1 \equiv 0\,\{\,p_2\,\}\,\mathsf{else}\,\{\,p_3\,\}\,]\!]_c & = & \mathsf{if}\; \mathcal{C}[\![\,p_1\,]\!] \equiv \{\,\mathcal{C}[\![\,p_2\,]\!]_{c_0}\,\}\,\mathsf{else}\,\{\,\mathcal{C}[\![\,p_3\,]\!]_{c_0}\,\} \\
\qquad \text{where} & & c_0 \equiv \tilde{\mu}x.j(\Gamma) \\
\qquad \text{with} & & \mathsf{def}\; j(\Gamma)\,\{\,\langle x \mid c \rangle\,\} \\
\qquad \text{and} & & \Gamma \;:=\; \mathsf{freeVars}(c),\; x :^{\mathsf{prd}} \tau \qquad (\text{where } c :^{\mathsf{cns}} \tau)
\end{array}
$$

$\mathcal{C}[\![\,\cdot\,]\!] : \mathit{Arguments}_{\mathsf{Fun}} \to \mathit{Arguments}_{\mathsf{Core}}$

$$
\begin{array}{lcl}
\mathcal{C}[\![\,\bullet\,]\!] & = & \bullet \\
\mathcal{C}[\![\,\sigma,\,p\,]\!] & = & \mathcal{C}[\![\,\sigma\,]\!],\, \mathcal{C}[\![\,p\,]\!] \\
\mathcal{C}[\![\,\sigma,\,\alpha\,]\!] & = & \mathcal{C}[\![\,\sigma\,]\!],\, \alpha
\end{array}
$$

$\mathcal{B}_v(\,\cdot\,)[\cdot] : \mathit{Argument}_{\mathsf{Core}} \times ((\mathit{Co})\,\mathit{Value}_{\mathsf{Core}} \to \mathit{Statement}_{\mathsf{Core}}) \to \mathit{Statement}_{\mathsf{Core}}$

$$
\begin{array}{lcll}
\mathcal{B}_v(e)[k] & = & \langle e \mid \tilde{\mu}x.k(x) \rangle & \text{if } e \neq a \\
\mathcal{B}_v(a)[k] & = & k(a) &
\end{array}
$$

$\mathcal{L}_v(\,\cdot\,)[\cdot] : \mathit{Arguments}_{\mathsf{Core}} \times ((\mathit{Co})\,\mathit{Values}_{\mathsf{Core}} \to \mathit{Statement}_{\mathsf{Core}}) \to \mathit{Statement}_{\mathsf{Core}}$

$$
\begin{array}{lcl}
\mathcal{L}_v(\,\bullet\,)[k] & = & k(\bullet) \\
\mathcal{L}_v(e \;::\; \sigma)[k] & = & \mathcal{B}_v(e)[\lambda a.\mathcal{L}_v(\sigma)[\lambda as.k(a \;::\; as)]]
\end{array}
$$

Figure 6.5: Translation from Fun to Core.

invocations, this is the destructor, where the given consumer input becomes the additional argument a destructor requires. There is a problem, however: A destructor is only a covalue if all its arguments are (co)values. We define values and covalues as follows:

Values	v	$::=$	$n \mid x \mid K(\nu) \mid \mathbf{new}\,\{\dots\} \mid p$	where $p : {}^{\mathsf{prd}}\ \mathbf{codata}\ T\,\{\dots\}$
Covalues	q	$::=$	$\alpha \mid D(\nu) \mid \mathbf{case}\,\{\dots\} \mid c$	where $c :\!\!\not{}^{\mathsf{hs}}\ \mathbf{codata}\ T\,\{\dots\}$
Argument Values	a	$::=$	$v \mid q$	
	ν	$::=$	$\bullet \mid \nu,\, a$	

Argument values a are either values or covalues. We write ν for a list of argument values. Values consist of integers, constructors whose arguments are all argument values, copattern matches, and all producers that are of a codata type. Dually, covalues consist of destructors whose arguments are all argument values, pattern matches, and all consumers that are not of a codata type. To ensure that we can safely inline a destructor, we thus lift all translated non-(co)value arguments of the destructor before turning it into the consumer input of the scrutinee, i.e., we bring the destructor into focused form (cf. 4.2). This is accomplished by the mutually recursive functions $\mathcal{B}_v(\,\cdot\,)[\cdot]$, and $\mathcal{L}_v(\,\cdot\,)[\cdot]$[5], which both take a continuation as an additional argument. This is similar to how the translation $\mathcal{C}[\![\,\cdot\,]\!]_{\odot}$ has an additional consumer input, but here the continuation is a meta-level function abstracting the (co)value that is to replace a lifted term in the statement out of which the term was lifted. Thus, when such a continuation is applied to a (co)value, it returns a statement with this (co)value in the proper argument position. The function $\mathcal{B}_v(\,\cdot\,)[\cdot]$ distinguishes whether the given argument e is a (co)value or not. If not, e must be a producer – since the only consumers that can occur are covariables – so the function creates a fresh variable and uses a $\tilde{\mu}$-binding to bind e in a cut. The body of the $\tilde{\mu}$-binding consists of the given continuation applied to the freshly generated variable. If the argument is a (co)value a already, the function simply applies the continuation to a, putting the (co)value back into its place. The function $\mathcal{L}_v(\,\cdot\,)[\cdot]$ simply maps $\mathcal{B}_v(\,\cdot\,)[\cdot]$ over a list of arguments, but in continuation-passing style.

For pattern matches, the new consumer input is the matching part, where we translate the body of each branch with the given consumer input c. To avoid duplicating this consumer, and hence the risk of exponential blowup, we introduce a join point if there is more than one branch. CPS translations often do so by using second-class continuation bindings to avoid allocating closures for join points (Kennedy, 2007; Cong et al., 2019). Here, we could bind the consumer to a covariable via a μ-binding, but since our covariables are first-class, this would allocate a closure. To also avoid closure allocation, we instead introduce a join point by defining a top-level function j, again implicitly assumed to be fresh, whose parameters are the free variables of the conusmer c and an additional fresh producer parameter x of the same type as c. The body of j is a cut of x and c. The consumer input to translate the branches of the pattern match then consists of a $\tilde{\mu}$-abstraction which binds x and calls j with the free variables of c and x as arguments in its body. If c is a very simple consumer, such as a covariable, we can of course go without the join point. A similar duplication arises in the translation of conditionals, where both branches are translated with the given consumer input, so we also introduce a join point there. We note that the $\tilde{\mu}$-abstraction used here for join points is not necessarily a covalue, as it might be of a codata type. However, we can still safely inline it, because the $\tilde{\mu}$-abstraction is semantically equivalent to the given consumer c: It results from η-expanding c to a trivial $\tilde{\mu}$-abstraction $\tilde{\mu}x.\langle x \mid c \rangle$, which is always valid (Downen and Ariola, 2018), and then replacing the cut by a call to a top-level function that can be performed immediately since all arguments are (co)variables.

The above translation preserves typeability.

Theorem 49 (Typeability Preservation)**.**
Let p be a producer in Fun, Γ a typing context, and τ a type. Moreover, let c be a consumer in Core.

[5] standing for: creating value $\mathcal{B}$indings and creating $\mathcal{L}$ists of value bindings

If $\Theta \vdash \mathbf{def} f(\Gamma) : \tau \{ p \}$ *OK, then* $\Theta \vdash \mathcal{C}[\![\mathbf{def} f(\Gamma) : \tau \{ p \}]\!]$ *OK.*
If $\Gamma \vdash p : \tau$ *and* $\Gamma \vdash c :^{cns} \tau$*, then* $\Gamma \vdash \mathcal{C}[\![p]\!] :^{prd} \tau$ *and* $\Gamma \vdash \mathcal{C}[\![p]\!]_c$*.*

Proof Sketch. Straightforward induction. The only thing to keep in mind is that codata types and top-level functions obtain an additional consumer argument instead of having a return type. □

6.4 Translating the Extensions Back to DS

After having seen how our surface language Fun can be translated into our high-level intermediate representation Core, we briefly pause the discussion of our compilation pipeline and reconsider the translation back to direct style examined in Part I. Fun and Core feature general data and codata types, not present in the calculi discussed in Part I. We now outline what additional challenges these features present when translating back to direct style. For simplicity, we again assume a focused language, as in Section 4.3.1 (we will discuss focusing for Core in detail in Section 7.1.1). We refer to the dissertation of Downen (2017) (Chapter IX) for a discussion of an unfocused version, yet in a somewhat more restricted setting, as we have explained already in Section 4.3.1.

6.4.1 Data Types

We start with data types. Constructors of data types can be translated back trivially, since they are the same in direct style as they are in CPS or a language based on classical sequent calculus. Pattern matching for data types essentially behaves like conditionals, which we have discussed in Section 3.3. The main difference is that pattern matches are not restricted to exactly two branches, but can also have only one branch or more than two. The case of only one branch is simple: The consumer output for the whole translated pattern match simply is the covariable to which the translated body of the one branch returns.

$$\frac{\mathcal{D}[\![s]\!] = s' \rightsquigarrow \alpha}{\mathcal{D}[\![\langle x \mid \mathbf{case} \{ K(\ldots) \Rightarrow s \} \rangle]\!] = x.\mathbf{case} \{ K(\ldots) \Rightarrow s' \} \rightsquigarrow \alpha} \text{ [DPM1]}$$

In the case of two or more branches, we have to distinguish cases based on where the translated bodies of the branches return to. If all of them return to the same covariable, we are in the pure case, so we do not need control operators and the whole translated pattern match returns to this covariable as well. If the translated bodies of the branches return to different covariables, however, we need to insert control operators. For the two branches of conditionals, we have discussed different approaches for doing so. One of them was the biased strategy of picking one of the branches as preferred and adapt the other one. We could still do the same with more than two branches, but consider the case where we have three branches and two of them return to the same covariable, while the other one returns to a different covariable.

$$\mathcal{D}[\![s_1]\!] = s_1' \rightsquigarrow \alpha \qquad \mathcal{D}[\![s_2]\!] = s_2' \rightsquigarrow \beta \qquad \mathcal{D}[\![s_3]\!] = s_3' \rightsquigarrow \beta$$

If the first branch is the preferred one, then the biased strategy requires us to insert control operators in both other branches.

$$\begin{aligned} &\mathcal{D}[\![\langle x \mid \mathbf{case} \{ K_1(\ldots) \Rightarrow s_1,\ K_2(\ldots) \Rightarrow s_2,\ K_3(\ldots) \Rightarrow s_3 \} \rangle]\!] \\ &= x.\mathbf{case} \{ K_1(\ldots) \Rightarrow s_1', \\ &\qquad\qquad K_2(\ldots) \Rightarrow \mathbf{suspend} \{ \alpha \Rightarrow \mathbf{run}(\beta) \{ s_2' \} \}, \\ &\qquad\qquad K_3(\ldots) \Rightarrow \mathbf{suspend} \{ \alpha \Rightarrow \mathbf{run}(\beta) \{ s_3' \} \} \} \rightsquigarrow \alpha \end{aligned}$$

However, this requires more control operators than adapting the preferred branch instead. Moreover, as the choice of the first branch to be preferred is arbitrary, chances are that the

current continuation is the one that the other two branches return to, requiring another insertion of control operators around the whole pattern match. When adapting the preferred branch in the first place, this is not necessary. For pattern matches, it thus seems more reasonable to apply a "majority wins" strategy, which assumes that the current continuation is the one that the majority of branches returns to, adapting the other ones.

$$\begin{aligned}&\mathcal{D}[\![\, \langle x \mid \mathbf{case}\,\{\, K_1(\ldots) \Rightarrow s_1,\ K_2(\ldots) \Rightarrow s_2,\ K_3(\ldots) \Rightarrow s_3 \,\}\rangle \,]\!] \\ &= x.\mathbf{case}\,\{\, K_1(\ldots) \Rightarrow \mathbf{suspend}\,\{\, \beta \Rightarrow \mathbf{run}(\alpha)\,\{\, s_1' \,\}\,\}, \\ &\qquad\quad K_2(\ldots) \Rightarrow s_2', \\ &\qquad\quad K_3(\ldots) \Rightarrow s_3' \,\} \rightsquigarrow \beta\end{aligned}$$

If there is a tie between two sets of branches, then we are back to picking one of them arbitrarily. We have also discussed alternative strategies for conditionals in Section 3.3, such as passing the current continuation down during the translation, which can help to resolve the issues of picking which branches to adapt. The same discussion applies to the more general case of pattern matches.

6.4.2 Codata Types

We have discussed the DS translation of two special cases of codata types in Section 4.3.1, functions and a negation type. Both are codata types with one destructor (Setzer, 2003), but while the `apply` destructor of functions we have seen in Section 6.2 has exactly one consumer parameter in Core, the destructor of the negation type, which we call `not` here, merely has a producer parameter and no consumer parameter at all.

Delimited Function-like Destructors

The DS translation of functions removes the consumer parameter, whose type becomes the return type of the function. This can be generalized to codata types with one destructor that has more than one consumer argument. We can pick one of the consumer arguments to be removed. In the translation from Fun into Core, we have seen that one consumer argument is added to every destructor of a codata type as its last parameter. Hence, it is reasonable to remove the last consumer parameter.

$$\mathcal{D}[\![\, \mathbf{codata}\ T\,\{\, D(\ldots,\ \alpha :^{\mathsf{cns}} \tau)\,\}\,]\!] = \mathbf{codata}\ T\,\{\, D(\ldots) : \tau \,\}$$

The invocation of the destructor is a base case (in general, we again have to check whether the removed covariable occurs free in the translated term).

$$\mathcal{D}[\![\, \langle x \mid D(\ldots,\ \alpha)\rangle \,]\!] = x.D(\ldots) \rightsquigarrow \alpha$$

For copattern matches, we distinguish the same cases as for functions, based on the translation of the body statement. For example, in the pure case, we do not have to insert a control operator.

$$\frac{\mathcal{D}[\![\, s_0 \,]\!] = s_0' \rightsquigarrow \alpha}{\mathcal{D}[\![\, \mathbf{new}\,\{\, D(\ldots,\ \alpha) \Rightarrow s_0 \,]\!] = \mathbf{new}\,\{\, D(\ldots) \Rightarrow s_0' \,\}}\ [\textsc{dcm1p}]$$

If the destructor is bound to a name, we wrap it into a $\tilde{\mu}$-abstraction binding a fresh variable before the translation, just as we did for application frames.

$$\langle \mu\alpha.s \mid D(\ldots,\ \beta)\rangle \longrightarrow \langle \mu\alpha.s \mid \tilde{\mu}x.\langle x \mid D(\ldots,\ \beta)\rangle\rangle \quad (x \text{ fresh})$$

Then we can treat it like a continuation, as we always do for critical pairs, and since the body of the continuation can never return to the fresh parameter x of the continuation, it becomes a **let**-binding for x.

This treatment also scales to codata types with multiple delimited destructors that each have at least one consumer parameter.

$$\mathcal{D}[\![\, \textbf{codata}\ T\,\{\, D_1(\ldots, \alpha_1 :^{\text{cns}} \tau_1),\ D_2(\ldots, \alpha_2 :^{\text{cns}} \tau_2),\ \ldots\,\}\,]\!] \\ = \textbf{codata}\ T\,\{\, D_1(\ldots) : \tau_1,\ D_2(\ldots) : \tau_2,\ \ldots\,\}$$

Destructor invocations can be treated in the same way as above, and in copattern matches we can simply translate every branch independently in the same way as above.

Undelimited Continuation-like Destructors

The negation codata type, whose destructor does not have a consumer parameter, is translated back to the special negation type in direct style, and its copattern matches and destructor invocations are treated as continuations and invocations of them. We can also generalize this to codata types with one destructor having multiple non-consumer parameters. To do so, we can generalize the DS negation type, and **let**-bindings and **process** definitions, present in our DS calculus λ_{D} from Section 3.1, to multiple bindings.

$$\mathcal{D}[\![\, \textbf{codata}\ T\,\{\, D(x : \tau, \ldots)\,\}\,]\!] = \neg(\tau, \ldots)$$

Then we can treat such codata types as continuations (with multiple arguments) for the translation to direct style, just as we did for the negation codata type.

However, if a codata type has multiple destructors with at least one of them having no consumer parameter, then we cannot simply translate it to a negation type in direct style. A possible solution could be to allow for non-returning destructors in DS codata types. In λ_{D} we had a context type # marking undelimited contexts, and we could allow destructors to be marked in this way instead of having a return type.

$$\mathcal{D}[\![\, \textbf{codata}\ T\,\{\, D_1(\ldots, \alpha_1 :^{\text{cns}} \tau_1),\ D_2(\ldots)\,\}\,]\!] \\ = \textbf{codata}\ T\,\{\, D_1(\ldots) : \tau_1,\ D_2(\ldots) : \#\,\}$$

Such an undelimited destructor can then only be called in an undelimited context, and the body of its branch in a copattern match has to be an undelimited statement. The branches in a copattern match for delimited destructors and the invocations of delimited destructors can still be translated as explained before. But for the undelimited destructors, which do not have a consumer parameter, the translation of the corresponding branch in a copattern match rather resembles the cases where a continuation is translated to a definition of a **process**. We expect that the translated body of such a statement does not return, so if this is the case, we do not insert control operators.

$$\frac{\mathcal{D}[\![\, s_1 \,]\!] = \ldots \qquad \mathcal{D}[\![\, s_2 \,]\!] = s_2' \rightsquigarrow \bullet}{\mathcal{D}[\![\, \textbf{new}\,\{\, D_1(\ldots, \alpha_1) \Rightarrow \ldots,\ D_2(\ldots) \Rightarrow s_2\,\}\,]\!] = \textbf{new}\,\{\, D_1(\ldots) \Rightarrow \ldots,\ D_2(\ldots) \Rightarrow s_2'\,\}}\ [\text{DCMNR}]$$

If the body s_2 of the undelimited destructor does return to a covariable α_2, we insert a control operator to reinstall the stack bound to α_2 around the statement, $\textbf{run}(\alpha_2)\,\{\, s_2'\,\}$. Invoking an undelimited destructor is still a base case, but one that does not return anywhere.

$$\mathcal{D}[\![\, \langle x \mid D_2(\ldots)\rangle\,]\!] = x.D_2(\ldots) \rightsquigarrow \bullet$$

6.4.3 Top-Level Function Definitions

For top-level function definitions, considerations similar to those for codata types hold. If the top-level definition has a consumer parameter, we can treat it in the same way as a codata type with one delimited destructor. To translate top-level definitions without a consumer parameter back to direct style, we also have to allow for undelimited top-level definitions in

the DS language, which we can again mark with #. The DS translation can then proceed as for undelimited destructors. Such top-level definitions can be thought of as top-level **process** definitions.

7 Normalizing Core to AxCut

In this chapter, we bring programs in the language Core into a normal form that is suitable for compilation to machine code. We first apply two transformations (Section 7.1), each targeting a smaller fragment of Core. Then we introduce our lower-level intermediate language AxCut (Section 7.2), and translate the fragment resulting from the transformations on Core into it (Section 7.3). AxCut has a slightly different structure than Core and its fragments, which allows us to directly generate code from it (Chapter 8).

7.1 Transformations on Core

In this section, we apply two transformations on Core. The first one is a focusing transformation that names all subterms. On this focused fragment, called Focused Core, we then apply a transformation that shrinks the language by removing redundant constructs. The resulting fragment is called Shrunk Core. Both transformations preserve typeability (Theorems 51 and 53).

7.1.1 Focusing Transformation

The focusing transformation $\mathcal{F}(\,\cdot\,)$ is shown in Figure 7.1. It lifts terms out of argument position, giving names to all these subterms, similar to A-normal form (Flanagan et al., 1993; Binder and Piecha, 2022) or continuation-passing style. This is an extension of the static focusing transformation (Andreoli, 1992; Curien and Herbelin, 2000), which usually only lifts non-value producers of data types and non-covalue consumers of codata types, as discussed in Section 4.2. The transformation targets the focused fragment of Core, presented in Figure 6.3, where arguments of constructors, destructors, and calls to top-level definitions as well as the producer arguments of conditionals, arithmetic operations, and terminating statements are always (co)variables.

To illustrate the idea, consider the example of mapping a function over a list from Example 47.

Example 50 (Focusing of Mapping a Function over a List)**.** There are two parts in the program in Example 47 where non-(co)variable terms occur in argument position. First, in the `Cons` branch in map, both arguments of `Cons` are μ-abstractions. Focusing this statement yields

$$\begin{aligned}&\mathcal{F}(\langle \texttt{Cons}(\mu\beta.\langle f \mid \texttt{apply}(y,\,\beta)\rangle,\,\mu\gamma.\mathsf{map}(ys,\,f,\,\gamma)) \mid \alpha\rangle)\\ &= \langle \mu\beta.\langle f \mid \texttt{apply}(y,\,\beta)\rangle \mid \tilde{\mu}h.\langle \mu\gamma.\mathsf{map}(ys,\,f,\,\gamma) \mid \tilde{\mu}t.\langle \texttt{Cons}(h,\,t) \mid \alpha\rangle\rangle\rangle\end{aligned}$$

The μ-abstractions are lifted out of the `Cons` by generating fresh names h and t for them, which are bound by $\tilde{\mu}$-abstractions, and then cutting the latter with the lifted terms, one after the other. The variables h and t are put in the positions of the lifted terms in the `Cons`.

Second, the call of map in increment has a copattern match as an argument. While this is not a computation, we still lift it and give it a name. Note that we also name the integer literal in the addition inside the copattern match.

$$\begin{aligned}&\mathcal{F}(\mathsf{map}(xs,\,\textbf{new}\,\{\,\texttt{apply}(x,\,\beta) \Rightarrow \langle x\,+\,1 \mid \beta\rangle\,\},\,\alpha))\\ &= \langle \textbf{new}\,\{\,\texttt{apply}(x,\,\beta) \Rightarrow \langle 1 \mid \tilde{\mu}y.\langle x\,+\,y \mid \beta\rangle\rangle\,\} \mid \tilde{\mu}f.\mathsf{map}(xs,\,f,\,\alpha)\rangle\end{aligned}$$

$\mathcal{F}(\cdot) : \mathit{Definition}_{\mathsf{Core}} \to \mathit{Definition}_{\mathsf{Focused\ Core}}$

$$\mathcal{F}(\mathbf{def}\, f(\Gamma)\,\{\, s \,\}) = \mathbf{def}\, f(\Gamma)\,\{\, \mathcal{F}(s) \,\}$$

$\mathcal{F}(\cdot) : \mathit{Statement}_{\mathsf{Core}} \to \mathit{Statement}_{\mathsf{Focused\ Core}}$

$$\begin{array}{lcl}
\mathcal{F}(\langle p_1 \ + \ p_2 \mid c \rangle) & = & \mathcal{B}(p_1)[\lambda a_1.\mathcal{B}(p_2)[\lambda a_2.\langle a_1 \ + \ a_2 \mid \mathcal{F}(c)\rangle]\,] \\
\mathcal{F}(\langle K(\sigma) \mid c \rangle) & = & \mathcal{L}(\sigma)[\lambda as.\langle K(as) \mid \mathcal{F}(c)\rangle] \\
\mathcal{F}(\langle p \mid D(\sigma) \rangle) & = & \mathcal{L}(\sigma)[\lambda as.\langle \mathcal{F}(p) \mid D(as)\rangle] \\
\mathcal{F}(\langle p \mid c \rangle) & = & \langle \mathcal{F}(p) \mid \mathcal{F}(c)\rangle \qquad \text{(for all other cuts)} \\
\mathcal{F}(\mathbf{if}\ p \equiv 0\,\{\, s_1 \,\}\ \mathbf{else}\,\{\, s_2 \,\}) & = & \mathcal{B}(p)[\lambda a.\mathbf{if}\ a \equiv 0\,\{\, \mathcal{F}(s_1) \,\}\ \mathbf{else}\,\{\, \mathcal{F}(s_2) \,\}] \\
\mathcal{F}(f(\sigma)) & = & \mathcal{L}(\sigma)[\lambda as.f(as)] \\
\mathcal{F}(\mathbf{exit}\ p) & = & \mathcal{B}(p)[\lambda a.\mathbf{exit}\ a]
\end{array}$$

$\mathcal{F}(\cdot) : \mathit{Producer}_{\mathsf{Core}} \to \mathit{Producer}_{\mathsf{Focused\ Core}}$

$$\begin{array}{lcl}
\mathcal{F}(x) & = & x \\
\mathcal{F}(\mu\alpha.s) & = & \mu\alpha.\mathcal{F}(s) \\
\mathcal{F}(\mathbf{new}\,\{\, D_1(\Gamma_1) \Rightarrow s_1, \ldots \}) & = & \mathbf{new}\,\{\, D_1(\Gamma_1) \Rightarrow \mathcal{F}(s_1), \ldots \} \\
\mathcal{F}(K(\sigma)) & & \text{does not occur} \\
\mathcal{F}(n) & = & n \\
\mathcal{F}(p_1 \ + \ p_2) & & \text{does not occur}
\end{array}$$

$\mathcal{F}(\cdot) : \mathit{Consumer}_{\mathsf{Core}} \to \mathit{Consumer}_{\mathsf{Focused\ Core}}$

$$\begin{array}{lcl}
\mathcal{F}(\alpha) & = & \alpha \\
\mathcal{F}(\tilde{\mu}x.s) & = & \tilde{\mu}x.\mathcal{F}(s) \\
\mathcal{F}(\mathbf{case}\,\{\, K_1(\Gamma_1) \Rightarrow s_1, \ldots \}) & = & \mathbf{case}\,\{\, K_1(\Gamma_1) \Rightarrow \mathcal{F}(s_1), \ldots \} \\
\mathcal{F}(D(\sigma)) & & \text{does not occur}
\end{array}$$

$\mathcal{B}(\cdot)[\cdot] : \mathit{Producer}_{\mathsf{Core}} \times (\mathit{Variable} \to \mathit{Statement}_{\mathsf{Focused\ Core}}) \to \mathit{Statement}_{\mathsf{Focused\ Core}}$

$$\begin{array}{lcl}
\mathcal{B}(x)[k] & = & k(x) \\
\mathcal{B}(\mu\alpha.s)[k] & = & \langle \mu\alpha.\mathcal{F}(s) \mid \tilde{\mu}x.k(x)\rangle \\
\mathcal{B}(K(\sigma))[k] & = & \mathcal{L}(\sigma)[\lambda as.\langle K(as) \mid \tilde{\mu}x.k(x)\rangle] \\
\mathcal{B}(\mathbf{new}\,\{\, D_1(\Gamma_1) \Rightarrow s_1, \ldots \})[k] & = & \langle \mathbf{new}\,\{\, D_1(\Gamma_1) \Rightarrow \mathcal{F}(s_1), \ldots \} \mid \tilde{\mu}x.k(x)\rangle \\
\mathcal{B}(n)[k] & = & \langle n \mid \tilde{\mu}x.k(x)\rangle \\
\mathcal{B}(p_1 \ + \ p_2)[k] & = & \mathcal{B}(p_1)[\lambda a_1.\mathcal{B}(p_2)[\lambda a_2.\langle a_1 \ + \ a_2 \mid \tilde{\mu}x.k(x)\rangle]\,]
\end{array}$$

$\mathcal{B}(\cdot)[\cdot] : \mathit{Consumer}_{\mathsf{Core}} \times (\mathit{Covariable} \to \mathit{Statement}_{\mathsf{Focused\ Core}}) \to \mathit{Statement}_{\mathsf{Focused\ Core}}$

$$\begin{array}{lcl}
\mathcal{B}(\alpha)[k] & = & k(\alpha) \\
\mathcal{B}(\tilde{\mu}x.s)[k] & = & \langle \mu\alpha.k(\alpha) \mid \tilde{\mu}x.\mathcal{F}(s)\rangle \\
\mathcal{B}(D(\sigma))[k] & = & \mathcal{L}(\sigma)[\lambda as.\langle \mu\alpha.k(\alpha) \mid D(as)\rangle] \\
\mathcal{B}(\mathbf{case}\,\{\, K_1(\Gamma_1) \Rightarrow s_1, \ldots \})[k] & = & \langle \mu\alpha.k(\alpha) \mid \mathbf{case}\,\{\, K_1(\Gamma_1) \Rightarrow \mathcal{F}(s_1), \ldots \}\rangle
\end{array}$$

$\mathcal{L}(\cdot)[\cdot] : \mathit{Arguments}_{\mathsf{Core}} \times (\mathit{Context} \to \mathit{Statement}_{\mathsf{Focused\ Core}}) \to \mathit{Statement}_{\mathsf{Focused\ Core}}$

$$\begin{array}{lcl}
\mathcal{L}(\bullet)[k] & = & k(\bullet) \\
\mathcal{L}(e \ :: \ \sigma)[k] & = & \mathcal{B}(e)[\lambda a.\mathcal{L}(\sigma)[\lambda as.k(a \ :: \ as)]\,]
\end{array}$$

Figure 7.1: The focusing transformation.

To make the transformation one-pass and compositional, and hence avoid administrative redexes, it is split into three mutually recursive functions $\mathcal{F}(\cdot)$, $\mathcal{B}(\cdot)[\cdot]$, and $\mathcal{L}(\cdot)[\cdot]$[6], where the latter two both take a continuation as an additional argument. $\mathcal{B}(\cdot)[\cdot]$ and $\mathcal{L}(\cdot)[\cdot]$ are similar to the functions $\mathcal{B}_v(\cdot)[\cdot]$ and $\mathcal{L}_v(\cdot)[\cdot]$ we have seen in the translation from Fun to Core in 6.3.1, but here, the functions lift all non-(co)variable arguments, not only non-(co)values. Their continuation is a meta-level function that abstracts the name given to a lifted term in the statement out of which the term was lifted. Thus, when such a continuation is applied to a concrete name, i.e., a variable or covariable, it returns a focused statement with this concrete name in the proper argument position. Also similar to the translation from Fun to Core, we leave freshness conditions for newly-bound variables and covariables implicit.

The interesting part for $\mathcal{F}(\cdot)$ is its definition on statements for the cases where producers and consumers occur in argument positions. All other parts of $\mathcal{F}(\cdot)$ are simple congruences. The idea is to lift all non-(co)variable producers and consumers in argument positions by calling $\mathcal{B}(\cdot)[\cdot]$ – or $\mathcal{L}(\cdot)[\cdot]$ in the case where a whole list of arguments is to be lifted – and pass the statement itself in the continuation input. The function $\mathcal{L}(\cdot)[\cdot]$ simply maps $\mathcal{B}(\cdot)[\cdot]$ over a list of arguments, but in continuation-passing style.

To avoid administrative redexes, the cases for a cut with a constituent that has other terms in argument positions, i.e., a constructor or destructor or an arithmetic operation, are special-cased. Otherwise, we would have to introduce spurious μ- or $\tilde{\mu}$-abstractions to turn statements into producers or consumers. For example, for cuts with constructors we bind the arguments σ with

$$\mathcal{L}(\sigma)[\lambda as.\langle K(as) \mid \mathcal{F}(c)\rangle]$$

The continuation abstracts the fresh names as outside of the cut and then puts them into the argument positions. Without special-casing, we would have to abstract the names inside of the cut and then turn the resulting statement into a producer again with a μ-abstraction,

$$\langle \mu\alpha.\mathcal{L}(\sigma)[\lambda as.\langle K(as) \mid \alpha\rangle] \mid \mathcal{F}(c)\rangle$$

This redex could immediately be reduced to arrive at the improved version. Note that typing ensures that the counterpart of the cut in these special cases cannot itself be a special case.

The function $\mathcal{B}(\cdot)[\cdot]$ pattern-matches on a given producer or consumer, transforms it recursively, and in most cases creates a fresh (co)variable and uses a μ- or $\tilde{\mu}$-binding to bind the transformed term in a cut. The body of the μ- or $\tilde{\mu}$-binding consists of the given continuation applied to the freshly generated (co)variable. In the case where the producer or consumer is another constructor or destructor or an arithmetic expression, the functions $\mathcal{B}(\cdot)[\cdot]$ or $\mathcal{L}(\cdot)[\cdot]$ are applied recursively to again lift the corresponding arguments. Otherwise, the function $\mathcal{F}(\cdot)$ is recursively applied to all substatements. The only exception is the case where the producer or consumer is a (co)variable already. Here, the continuation is simply applied to it, which puts the (co)variable into the argument position abstracted by the continuation.

The above transformation preserves typeability.

Theorem 51 (Typeability Preservation)**.**
Let s be a statement in Core, c a consumer, p a producer, Γ a typing context, and τ a type.
If $\Theta \vdash \mathbf{def} f(\Gamma)\{\, s \,\}$ OK, then $\Theta \vdash \mathcal{F}(\mathbf{def} f(\Gamma)\{\, s \,\})$ OK.
If $\Gamma \vdash s$, then $\Gamma \vdash \mathcal{F}(s)$.
If $\Gamma \vdash p :^{prd} \tau$, then $\Gamma \vdash \mathcal{F}(p) :^{prd} \tau$.
If $\Gamma \vdash c :^{cns} \tau$, then $\Gamma \vdash \mathcal{F}(c) :^{cns} \tau$.

Proof Sketch. Straightforward induction. As the types are not affected, the proof amounts to showing that the cuts generated by lifting arguments are well-typed and that the (co)variables to which the lifted arguments are bound have the same types as the latter. But since we can simply choose the types of the fresh (co)variables for the μ- and $\tilde{\mu}$-bindings accordingly, this is trivial. □

[6] standing for: $\mathcal{F}$ocusing, creating $\mathcal{B}$indings, and creating $\mathcal{L}$ists of bindings

7.1.2 Shrinking Transformation

The second transformation $\mathcal{S}(\,\cdot\,)$ removes redundant parts of the language by first inlining the syntax of producers and consumers into statements, and then eliminating combinations that are either impossible due to the typing rules or can be eliminated in a different way. The resulting shrunk fragment of Core, presented in Figure 6.3, thus only consists of statements without separate syntactic categories of producers and consumers. We only consider the cases for which the transformation is interesting, all other cases are simple congruences.

Inlining Producers and Consumers in Cuts

Inlining all six possible producers and four possible consumers in cuts results in 24 different combinations. Six of these combinations are precluded by typing. A constructor and a destructor, as well as a pattern and a copattern match, can never meet in a cut. Moreover, integer literals and arithmetic operations can never be cut with a destructor or a pattern match. This leaves us with the following 18 possibilities.

$$
\begin{array}{rcl}
s & ::= & \langle K(\sigma) \mid \tilde{\mu} x.s \rangle \mid \langle \mu\alpha.s \mid D(\sigma) \rangle \mid \langle K(\sigma) \mid \alpha \rangle \mid \langle x \mid D(\sigma) \rangle \\
& \mid & \langle x \mid \mathbf{case}\,\{\, K(\Gamma) \Rightarrow s,\, \ldots \,\} \rangle \mid \langle \mu\alpha.s \mid \mathbf{case}\,\{\, K(\Gamma) \Rightarrow s,\, \ldots \,\} \rangle \\
& \mid & \langle \mathbf{new}\,\{\, D(\Gamma) \Rightarrow s,\, \ldots \,\} \mid \alpha \rangle \mid \langle \mathbf{new}\,\{\, D(\Gamma) \Rightarrow s,\, \ldots \,\} \mid \tilde{\mu} x.s \rangle \\
& \mid & \langle n \mid \tilde{\mu} x.s \rangle \mid \langle x \mathbin{+} x \mid \tilde{\mu} x.s \rangle \\
& \mid & \langle K(\sigma) \mid \mathbf{case}\,\{\, K(\Gamma) \Rightarrow s,\, \ldots \,\} \rangle \mid \langle \mathbf{new}\,\{\, D(\Gamma) \Rightarrow s,\, \ldots \,\} \mid D(\sigma) \rangle \\
& \mid & \langle \mu\alpha.s \mid \beta \rangle \mid \langle y \mid \tilde{\mu} x.s \rangle \mid \langle \mu\alpha.s \mid \tilde{\mu} x.s \rangle \mid \langle x \mid \alpha \rangle \mid \langle n \mid \alpha \rangle \mid \langle x \mathbin{+} x \mid \alpha \rangle
\end{array}
$$

We will now eliminate the lower eight of these.

Removing Renaming

We start with the cases that can be eliminated by a simple (co)variable renaming. This is the case where a variable or covariable meets a $\tilde{\mu}$- or μ-binding, but also where we pattern-match or copattern-match on a known constructor or destructor, since the arguments σ of constructors and destructors only consist of (co)variables after focusing. These reductions hence never introduce new cuts. We denote the (capture-avoiding) renaming of (co)variable v by v_0 in statement s with $s[v \mapsto v_0]$ and similar for multiple (co)variables.

$$
\begin{array}{lcl}
\mathcal{S}(\langle \mu\alpha.s \mid \beta \rangle) & = & \mathcal{S}(s[\alpha \mapsto \beta]) \\
\mathcal{S}(\langle y \mid \tilde{\mu} x.s \rangle) & = & \mathcal{S}(s[x \mapsto y]) \\
\mathcal{S}(\langle K_j(\sigma) \mid \mathbf{case}\,\{\, K_1(\Gamma_1) \Rightarrow s_1,\, \ldots \,\} \rangle) & = & \mathcal{S}(s_j[\Gamma_j \mapsto \sigma]) \\
\mathcal{S}(\langle \mathbf{new}\,\{\, D_1(\Gamma_1) \Rightarrow s_1,\, \ldots \,\} \mid D_j(\sigma) \rangle) & = & \mathcal{S}(s_j[\Gamma_j \mapsto \sigma])
\end{array}
$$

Removing Critical Pairs and Unknown Cuts

Next, consider the case where a μ- and a $\tilde{\mu}$-binding are cut, the critical pairs. To transform those away, we distinguish between the three different kinds of types. We assume call-by-value evaluation order for data types and call-by-name evaluation order for codata types, which ensures that all η-laws hold for both kinds of types (Binder et al., 2024; Downen and Ariola, 2020). Therefore, we can η-expand the $\tilde{\mu}$-binding in critical pairs of data types and the μ-binding in critical pairs of codata types, based on their set of constructors and destructors, respectively. Here we write $X(\Gamma)$ to mean that the constructor or destructor is applied to the (co)variables bound in Γ, in that order. To avoid exponential blowup, we share the statement of the expanded side of the cut by lifting it into a join point on the top level, similar to how we shared the consumer during the translation from Fun to Core. If there is only one branch, or if the statement is a leaf, we can forgo the lifting.

$$
\begin{array}{rcl}
\mathcal{S}(\langle \mu\alpha.s_1 \mid \tilde{\mu}x.s_2 \rangle_T) & = & \langle \mu\alpha.\mathcal{S}(s_1) \mid \mathbf{case}\,\{\, K_1(\Gamma_1) \Rightarrow \langle K_1(\Gamma_1) \mid \tilde{\mu}x.j(\Gamma)\rangle, \ldots \,\}\rangle \\
\quad \text{where} & & \mathbf{data}\; T\,\{\, K_1(\Gamma_1), \ldots \,\} \in \Theta \qquad (\Gamma_1, \ldots \text{ fresh}) \\
\quad \text{with} & & \mathbf{def}\; j(\Gamma)\,\{\, \mathcal{S}(s_2) \,\} \\
\quad \text{and} & & \Gamma := \mathsf{freeVars}(\mathcal{S}(s_2)) \\
\mathcal{S}(\langle \mu\alpha.s_1 \mid \tilde{\mu}x.s_2 \rangle_T) & = & \langle \mathbf{new}\,\{\, D_1(\Gamma_1) \Rightarrow \langle \mu\alpha.j(\Gamma) \mid D_1(\Gamma_1)\rangle, \ldots \,\} \mid \tilde{\mu}x.\mathcal{S}(s_2)\rangle \\
\quad \text{where} & & \mathbf{codata}\; T\,\{\, D_1(\Gamma_1), \ldots \,\} \in \Theta \qquad (\Gamma_1, \ldots \text{ fresh}) \\
\quad \text{with} & & \mathbf{def}\; j(\Gamma)\,\{\, \mathcal{S}(s_1) \,\} \\
\quad \text{and} & & \Gamma := \mathsf{freeVars}(\mathcal{S}(s_1))
\end{array}
$$

Moreover, completely unknown cuts, that is, cuts of a variable with a covariable, can be transformed away in the same way for data types and codata types. That is, we η-expand the covariable in cuts at data types and we η-expand the variable in cuts at codata types. In this case, however, there is no risk of blowup.

$$
\begin{array}{rcl}
\mathcal{S}(\langle x \mid \alpha \rangle_T) & = & \langle x \mid \mathbf{case}\,\{\, K_1(\Gamma_1) \Rightarrow \langle K_1(\Gamma_1) \mid \alpha\rangle, \ldots \,\}\rangle \\
\quad \text{where} & & \mathbf{data}\; T\,\{\, K_1(\Gamma_1), \ldots \,\} \in \Theta \qquad (\Gamma_1, \ldots \text{ fresh}) \\
\mathcal{S}(\langle x \mid \alpha \rangle_T) & = & \langle \mathbf{new}\,\{\, D_1(\Gamma_1) \Rightarrow \langle x \mid D_1(\Gamma_1)\rangle, \ldots \,\} \mid \alpha\rangle \\
\quad \text{where} & & \mathbf{codata}\; T\,\{\, D_1(\Gamma_1), \ldots \,\} \in \Theta \qquad (\Gamma_1, \ldots \text{ fresh})
\end{array}
$$

As an illustration, consider the following example.

Example 52. The top-level function range creates a list with entries from 0 to some integer n. The entries of this list are doubled by the top-level function double and the result r is used in the remaining term t.

$$\mathbf{let}\; r \;=\; \mathsf{double}(\mathsf{range}(n));\; t$$

After translation and focusing, we obtain the following statement, where s is the result of translating and focusing t.

$$\langle \mu\alpha.\mathsf{range}(n,\, \alpha) \mid \tilde{\mu}x.\langle \mu\beta.\mathsf{double}(x,\, \beta) \mid \tilde{\mu}r.s\rangle\rangle$$

The type of the outer cut is `List`[i64], so to remove the critical pair, we η-expand the right-hand side as follows.

$$
\begin{array}{l}
\langle \mu\alpha.\mathsf{range}(n,\, \alpha) \mid \mathbf{case}\,\{\, \texttt{Nil} \Rightarrow \langle \texttt{Nil} \mid \tilde{\mu}x.\mathsf{jp}(x)\rangle \\
\qquad\qquad\qquad\qquad \texttt{Cons}(y,\, ys) \Rightarrow \langle \texttt{Cons}(y,\, ys) \mid \tilde{\mu}x.\mathsf{jp}(x)\rangle \,\}\rangle \\
\mathbf{def}\; \mathsf{jp}(x :^{\mathsf{prd}} \texttt{List}[\mathsf{i64}])\,\{\, \mathcal{S}(\langle \mu\beta.\mathsf{double}(x,\, \beta) \mid \tilde{\mu}r.s\rangle) \,\}
\end{array}
$$

Here we have introduced the join point jp to share the body of the expanded consumer, which is now the body of jp. This body contains a critical pair of type `List`[i64] again, so its transformation needs η-expansion again, and thus leads to code blowup if not shared. We have assumed that s has no free variables, otherwise these would become additional arguments for jp.

Dealing with Built-In Types

For the built-in type i64, we cannot use η-expansion, so we need another way to resolve critical pairs. The $\tilde{\mu}$-binding can be viewed as a continuation for integers. As we use call-by-value evaluation for integers, we should find a different possibility to model this continuation in such a way that the μ-binding would be reduced first. To do so, we can define a new data type:

$$\mathbf{data}\; \texttt{Cont}\,\{\, \texttt{Ret}(x :^{\mathsf{prd}} \mathsf{i64}) \,\}$$

A pattern match of this data type can indeed be viewed as a continuation for integers, the only difference from the $\tilde{\mu}$-binding being that the integer is wrapped into the constructor `Ret`. This means that in all places where an integer is cut with a covariable, which now stands for a consumer of type `Cont` instead of a $\tilde{\mu}$-binding of type i64, we have to wrap the integer into a `Ret`.

We also have to reflect this change in typing contexts and adapt these accordingly (in the cases above, we have ignored this for ease of presentation). While this type is structurally identical to a data type for boxed integers, with the `Ret` constructor corresponding to a box, we never use it to actually box integers. Its purpose is not to facilitate a uniform representation as used by many languages to implement polymorphism, but only to model first-class continuations for integers.

There are three cases where covariables of type i64 occur in cuts: in a cut of a variable and a covariable of type i64, in the cut of an integer literal with a covariable, and in the cut of an arithmetic expression with a covariable. For the latter two cases, we further insert a $\tilde{\mu}$-binding to give the integer literal or the result of the arithmetic operation a name before wrapping this name into a `Ret`, which is then cut with the covariable.

$$
\begin{array}{lcll}
\mathcal{S}(\langle \mu\alpha.s_1 \mid \tilde{\mu}x.s_2 \rangle_{\mathsf{i64}}) & = & \langle \mu\alpha.\mathcal{S}(s_1) \mid \mathbf{case}\,\{\,\mathtt{Ret}(x) \Rightarrow \mathcal{S}(s_2)\,\}\rangle & \\
\mathcal{S}(\langle x \mid \alpha \rangle_{\mathsf{i64}}) & = & \langle \mathtt{Ret}(x) \mid \alpha \rangle & \\
\mathcal{S}(\langle n \mid \alpha \rangle) & = & \langle n \mid \tilde{\mu}x.\langle \mathtt{Ret}(x) \mid \alpha \rangle\rangle & (x \text{ fresh}) \\
\mathcal{S}(\langle x_1 + x_2 \mid \alpha \rangle) & = & \langle x_1 + x_2 \mid \tilde{\mu}x.\langle \mathtt{Ret}(x) \mid \alpha \rangle\rangle & (x \text{ fresh}) \\
\mathcal{S}(\Gamma,\, \alpha :^{\mathsf{cns}} \mathsf{i64}) & = & \mathcal{S}(\Gamma),\, \alpha :^{\mathsf{cns}} \mathtt{Cont} & \\
\mathcal{S}(\Gamma,\, v :^{\chi} \tau) & = & \mathcal{S}(\Gamma),\, v :^{\chi} \tau &
\end{array}
$$

We can see that no integer wrapped into `Ret`, that is, no producer of type `Cont`, is ever bound to a variable. As will become clear later, this means that we never allocate memory for wrapped integers on the heap. Indeed, they only appear in cuts with covariables in which the role of the constructor is to pick a branch in the pattern match. As there is always just one branch, the one for `Ret`, there is no overhead in modelling continuations for integers this way. We have seen an illustration of this in Example 43 in Section 5.3.

The above transformation again preserves typeability.

Theorem 53 (Typeability Preservation)**.**
Let s be a statement in Focused Core and Γ a typing context.
If $\Theta \vdash \mathbf{def} f(\Gamma)\,\{\,s\,\}$ Ok, then $\Theta \vdash \mathcal{S}(\mathbf{def} f(\Gamma)\,\{\,s\,\})$ Ok.
If $\Gamma \vdash s$, then $\mathcal{S}(\Gamma) \vdash \mathcal{S}(s)$.

Proof Sketch. Straightforward induction.
The only case where types are affected is for covariables of type i64, which become consumers of type `Cont`. As we have ensured that all producers of type i64 that are cut with such a covariable are wrapped into a `Ret` constructor, typeability is preserved for these cases.
Moreover, for critical pairs at type i64, we turn the consumer from a $\tilde{\mu}$-abstraction into a pattern match of type `Cont`, so that the type fits with the covariable bound by the μ-abstraction also in this case. Since η-expansion does not affect types, the corresponding cases for data and codata types are trivial, as well-typedness for the join points introduced is rather immediate.
The only cases left to consider are the ones where we remove renaming. But here we reduce well-typed cuts to substatements, which must hence themselves be well-typed, so this is immediate as well. □

7.2 The Lower-Level Intermediate Language AxCut

In the previous section, we have arrived at the fragment Shrunk Core whose terms only consist of statements, many of which are certain kinds of cuts. Due to the duality between data and codata types, the cuts that involve constructs for these come in pairs exhibiting a perfect symmetry. For example, $\langle K(\sigma) \mid \tilde{\mu}x.s \rangle$ and $\langle \mu\alpha.s \mid D(\sigma) \rangle$ are completely dual to each other. They contain exactly the same kind of information and behave operationally exactly the same (Ostermann et al., 2022), hence we can treat them in exactly the same way during code generation. Therefore, we merge them into a single unified form $\mathbf{let}\ v = X(\sigma);\ s$, where v is either a variable or a covariable, and X is either the name of a constructor or a destructor, as indicated in

Figure 6.3. Similarly, cuts of a constructor or destructor with a (co)variable are represented by **invoke** statements, cuts of (co)pattern matches with (co)variables become **switch** statements, and (co)pattern matches bound by a μ- or $\tilde{\mu}$-abstraction are turned into **create** statements. To avoid the use of cut syntax altogether, we also use a different syntax for binding integer literals and results of arithmetic operations. The resulting statement forms thus are:

$$
\begin{array}{rcl}
s & ::= & \textbf{let}\ v\ =\ X(\sigma);\ s \mid \textbf{invoke}\ v\ X(\sigma) \\
& \mid & \textbf{switch}\ v\ \{\ X(\Gamma) \Rightarrow s,\ \ldots\ \} \mid \textbf{create}\ v\ =\ \Gamma\ \{\ X(\Gamma) \Rightarrow s,\ \ldots\ \};\ s \\
& \mid & \textbf{lit}\ v \leftarrow n;\ s \mid v \leftarrow v\ +\ v;\ s \mid \textbf{if}\ v \equiv 0\ \{\ s\ \}\ \textbf{else}\ \{\ s\ \} \\
& \mid & f(\sigma) \mid \textbf{substitute}\ [\Gamma\ :=\ \sigma];\ s \mid \textbf{exit}\ v
\end{array}
$$

There is one new kind of statement that has not appeared so far, namely explicit substitutions (Abadi et al., 1991) $\textbf{substitute}\ [\Gamma\ :=\ \sigma];\ s$. In fact, this is the final ingredient we need to make our language suitable for a direct translation to machine code. In Core and the fragments considered so far, the structural rules of weakening, contraction, and exchange were implicit, i.e., variables could be used multiple times or not at all and independently of the order in which they were introduced in the context. We will now make these rules explicit using explicit substitution statements, which define a new environment Γ for the remaining statement s based on the variables in scope. This includes reordering variables by using them once in σ, duplicating variables by using them multiple times, and dropping variables by not using them at all. Substitutions happen simultaneously. Logically speaking, they can be thought of as context morphisms (Hofmann, 1997). Instead of the notation $\Gamma\ :=\ \sigma$, we also often write lists of individual assignments as in $v_1'\ :=\ v_1,\ v_2'\ :=\ v_2,\ \ldots$, with the understanding that the left-hand sides constitute Γ and the right-hand sides constitute σ.

Consequently, as we will see in the typing rules, the statements now consume a part of the environment according to linear, or even ordered, logic. To make this consumption explicit in all constructs, we annotate the closure environment Γ in first-class usages of (co)pattern matches, i.e., in the **create** construct. The exceptions from this scheme are the extra-logical constructs, such as arithmetic operators or conditionals.

The resulting calculus is called AxCut. We discuss its typing rules in Section 7.2.1 and give an operational semantics in Section 7.2.2. Finally, we present the translation from Shrunk Core to AxCut in Section 7.3.

7.2.1 Typing Rules

The rule Substitute for explicit substitutions makes the structural rules of weakening, contraction, and exchange explicit. It is the only place where sharing and erasing of variables can occur and also the only way to reorder variables in the environment for the subsequent statement s. The variables present in the old environment Γ can occur in an arbitrary way in the right-hand side σ in an explicit substitution. These variables are then given new names according to the new environment Γ', with types according to the old bindings. The typing judgment for the arguments σ is thus still not ordered.

$$
\frac{\Gamma \vdash \sigma\ :\ \Gamma' \qquad \Gamma' \vdash s}{\Gamma \vdash \textbf{substitute}\ [\Gamma'\ :=\ \sigma];\ s}\ [\textsc{Substitute}]
$$

All other typing rules, except for those that can be thought of as extra-logical, follow the rules of ordered type systems. The extra-logical constructs include integer literals, arithmetic operators, conditionals, as well as the terminating statement **exit** v. For these constructs, the variables of type i64 are merely required to be present in the environment, no matter at what position, and are not consumed.

$$\frac{\Gamma,\ v :^{\mathsf{prd}} \mathsf{i64} \vdash s}{\Gamma \vdash \mathbf{lit}\, v \leftarrow n;\ s}\ [\textsc{Lit}] \qquad \frac{v_1 :^{\mathsf{prd}} \mathsf{i64} \in \Gamma \qquad v_2 :^{\mathsf{prd}} \mathsf{i64} \in \Gamma \qquad \Gamma,\ v :^{\mathsf{prd}} \mathsf{i64} \vdash s}{\Gamma \vdash v \leftarrow v_1\ +\ v_2;\ s}\ [\textsc{Plus}]$$

$$\frac{v :^{\mathsf{prd}} \mathsf{i64} \in \Gamma}{\Gamma \vdash \mathbf{exit}\, v}\ [\textsc{Exit}] \qquad \frac{v :^{\mathsf{prd}} \mathsf{i64} \in \Gamma \qquad \Gamma \vdash s_1 \qquad \Gamma \vdash s_2}{\Gamma \vdash \mathbf{if}\, v \equiv 0\, \{\, s_1 \,\}\, \mathbf{else}\, \{\, s_2 \,\}}\ [\textsc{IfZ}]$$

We could also cast these statements into a common form for *external* statements, which can be instantiated as needed, thus providing a general way to extend the language with external operations. These external statements provide a way to interact with the external environment, e.g., for input and output or termination. They typically only act on external, i.e., non-logical, types like the built-in machine integers.

$$\frac{\Gamma' \vdash m : (\Gamma_1), \ldots \qquad \Gamma \vdash \sigma : \Gamma' \qquad \Gamma, \Gamma_1 \vdash s_1 \quad \ldots}{\Gamma \vdash \mathbf{extern}\, m(\sigma)\{\, (\Gamma_1) \Rightarrow s_1, \ldots \}}\ [\textsc{Extern}]$$

An external operation m can have an arbitrary number of inputs Γ', e.g., zero in the case of literals, one for conditionals, and two for arithmetic expressions. They can also have an arbitrary number of branches, each of which binds a number of variables (Γ_i) in the environment for its body statement s_i. For example, terminal statements have zero branches, arithmetic operations have one, and conditionals have two. Which assumptions and consequences an external operation m has, is defined outside the system. In the rest of the paper, we will stick to the operations we have seen thus far.

Rule Call for calling top-level functions adheres to the requirements of ordered logic. It ensures that the current type environment exactly matches the one assumed by the top-level definition when calling it, i.e., all superfluous variables must have been dropped and the remaining ones must be in the correct order. Here we write $f(\Gamma)$ to mean that the top-level function is applied to the variables bound in Γ, in that order, although we could also leave this information out since it is redundant now.

$$\frac{\mathbf{def}\, f(\Gamma)\, \{\ldots\} \in \Theta}{\Theta \mid \Gamma \vdash f(\Gamma)}\ [\textsc{Call}]$$

The other typing rules are concerned with producers and consumers of logical types, i.e., data and codata types. As they come in dual pairs, and since we use a uniform notation for these constructs anyway, we can formulate their rules uniformly. To do so, we define the following notation to connect the polarity and chirality and use it in the rules.

$$\chi_1(\mathbf{data}) := \mathsf{prd} \qquad \chi_1(\mathbf{codata}) := \mathsf{cns} \qquad \chi_2(\mathbf{data}) := \mathsf{cns} \qquad \chi_2(\mathbf{codata}) := \mathsf{prd}$$

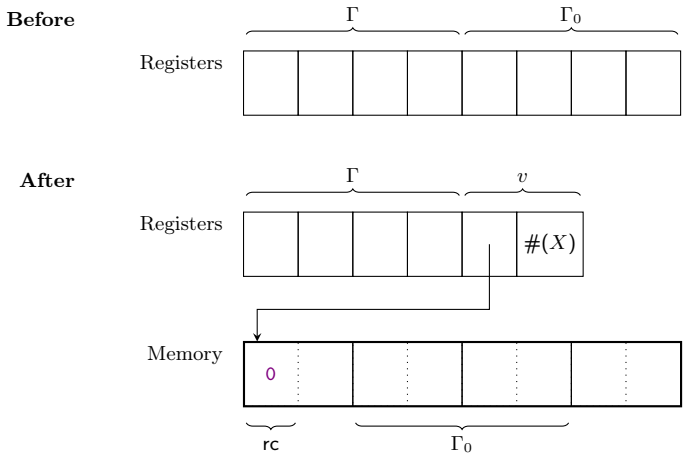
We could also go one step further and turn the system into a fully one-sided sequent calculus (Girard, 1987). However, as this does not affect code generation in an essential way, we refrain from doing so here.

We use the rules Let-π, parametrized by the polarity π, to allocate a constructor or destructor X in memory, and to bind the address to a variable in the remaining statement s. As the arguments of X must all be variables, there are no subderivations for them anymore. Instead, we require the part of the type environment Γ_0 that constitutes the arguments of X to exactly match its signature. This environment is consumed before the new binding is added for the subderivation of s.

$$\frac{\pi\ T\, \{\ldots,\ X(\Gamma_0) \ldots\} \in \Theta \qquad \Gamma,\ v :^{\chi_1(\pi)} T \vdash s}{\Theta \mid \Gamma,\ \Gamma_0 \vdash \mathbf{let}\, v\ =\ X(\Gamma_0);\ s}\ [\textsc{Let-}\pi]$$

While being rather high-level, this typing rule already embodies what happens on the machine level when a **let** statement is executed. It is therefore instructive to briefly consider some low-

level details at this point, which will be extensively discussed in Chapter 8. To provide some intuition, consider the following illustration of the registers and memory.



The typing context corresponds to registers. Before the **let** statement, the registers contain the part Γ of the environment that remains for the subsequent statement and, next to it, the part Γ_0 that is consumed. During the execution of the **let** statement, a new memory block is allocated and the registers corresponding to Γ_0 are stored into it. The first field of the memory block contains its reference count (rc). A pointer to this block is stored into the first register after Γ, which is free now that Γ_0 was consumed. The next register after this pointer is filled with the tag for the constructor or destructor X according to the definition of its type. These two registers constitute the variable v to which X was bound.

Similarly to the Let rules, in the Create rules, the closure environment Γ_0 of the branches is consumed. It must be used in each clause in addition to the parameters Γ_i of the corresponding constructor or destructor.

$$\frac{\pi\ T\,\{\,X_1(\Gamma_1), \dots\,\} \in \Theta \qquad \Gamma,\ v :^{\chi_2(\pi)}\ T \vdash s \qquad \forall i : \Gamma_i, \Gamma_0 \vdash s_i}{\Theta \mid \Gamma,\ \Gamma_0 \vdash \textsf{create}\, v\ =\ \Gamma_0\,\{\,X_1(\Gamma_1) \Rightarrow s_1, \dots\,\};\ s}\ [\textsc{Create}\text{-}\pi]$$

The intuition for what happens on the machine level is quite similar to the **let** statement. The consumed closure environment is stored in a memory block pointed to by the first free register. However, instead of a tag, the second register is filled with a pointer to the code for the (co)pattern match. The above typing rule ensures that the closure environment is in the appropriate registers. Hence, we do not have to explicitly pass the environment and we can keep our types simple.

Dually to the Let and Create rules, in the Switch and Invoke rules no new variable is bound, but the top-most one in the environment is consumed. Thus, on the machine level, after jumping to the correct branch, the corresponding parts of the environment previously stored in a memory block are loaded back into registers. In the Switch rules, the rest of the environment Γ must be used in each clause in addition to the parameters Γ_i of the corresponding constructor or destructor.

$$\frac{\pi\ T\,\{\,X_1(\Gamma_1), \dots\,\} \in \Theta \qquad \forall i : \Gamma, \Gamma_i \vdash s_i}{\Theta \mid \Gamma,\ v :^{\chi_1(\pi)}\ T \vdash \textsf{switch}\, v\,\{\,X_1(\Gamma_1) \Rightarrow s_1, \dots\,\}}\ [\textsc{Switch}\text{-}\pi]$$

In the INVOKE rules, the rest of the environment Γ must exactly match the signature of the constructor or destructor, again because the variable restriction for arguments is baked into the rules. This is similar to the call of top-level functions, and we could again leave the arguments implicit.

$$\frac{\pi\ T\{\ldots, X(\Gamma), \ldots\} \in \Theta}{\Theta \mid \Gamma,\ v :^{\chi_2(\pi)}\ T \vdash \textsf{invoke}\, v\, X(\Gamma)}\ [\textsc{Invoke-}\pi]$$

Example 54 (Mapping a Function over a List in AxCut)**.** To get acquainted with the system, let us look again at the top-level definition `map` from Example 46, which maps a function over a list, this time in AxCut.

```
def map(xs :prd List[i64], f :prd Fun[i64, i64], α :cns List[i64]) {
  substitute [α := α, f := f, xs := xs];
  switch xs {
    Nil ⇒ substitute [α := α]; invoke α Nil,
    Cons(y, ys) ⇒
      substitute [y := y, f0 := f, f := f, ys := ys, α := α];
      create β = (f, ys, α) { Ret(r) ⇒
        substitute [ys := ys, f := f, r := r, α := α];
        create γ = (r, α) {
          Nil ⇒ let rs = Nil; substitute [r := r, rs := rs, α := α]; invoke α Cons(r, rs)
          Cons(z, zs) ⇒
            substitute [α := α, r := r, z := z, zs := zs];
            let rs = Cons(z, zs); substitute [r := r, rs := rs, α := α]; invoke α Cons(r, rs)
        };
        map(ys, f, γ)
      };
      substitute [y := y, β := β, f0 := f0]; invoke f0 apply(y, β)
  }
}
```

It is visible how explicit substitutions are used to drop the function f in the `Nil` case by not mentioning it on any of the right-hand sides, and to duplicate the function in the `Cons` branch by using it in two right-hand sides. Moreover, if necessary, explicit substitutions rearrange the context appropriately before direct jumps to top-level labels or indirect jumps via **invoke**, but also for other non-external constructs, e.g., for the closure environment consumed by the closure γ created by **create** or to bring xs into the right position before the **switch**.

7.2.2 Operational Semantics

We define the semantics of AxCut in terms of an abstract machine, whose syntax is defined in Figure 7.2. The corresponding typing rules are straightforward and can be found in Schuster et al. (2025b). To ease presentation, we use the non-terminal b to denote a list of branches,

$$b ::= \{\, X(\Gamma) \Rightarrow s, \ldots \}$$

The operational semantics is very close to what happens on a real machine. A machine configuration consists of a statement s under execution, a value environment E mapping variables to values, and a program Θ. The latter is required for jumping to top-level functions and does not change during execution. We hence omit it from the definition of the individual steps. Values are either constructor or destructor values $\{\, X;\, \mathsf{E} \,\}$ consisting of a name and a field environment, or closures $\{\, \mathsf{E};\, b \,\}$ consisting of a closure environment and branches, or machine integers.

Syntax of Machine States

Values	V	::=	$\{ X; \mathsf{E} \} \mid \{ \mathsf{E}; b \} \mid 0 \mid 1 \mid \ldots$
Environments	E	::=	$x \mapsto \mathsf{V}, \ldots$
Configurations	M	::=	$\langle s \parallel \mathsf{E} \parallel \Theta \rangle$

Stepping Relation

(let)	$\langle \textbf{let}\ v\ =\ X(\Gamma_0);\ s \parallel \mathsf{E},\ \mathsf{E}_0 \rangle$	$\to$	$\langle s \parallel \mathsf{E},\ v \mapsto \{ X; \mathsf{E}_0 \} \rangle$	where $\mathsf{E}_0 : \Gamma_0$
(create)	$\langle \textbf{create}\ v\ =\ \Gamma_0\ b;\ s \parallel \mathsf{E},\ \mathsf{E}_0 \rangle$	$\to$	$\langle s \parallel \mathsf{E},\ v \mapsto \{ \mathsf{E}_0; b \} \rangle$	where $\mathsf{E}_0 : \Gamma_0$
(switch)	$\langle \textbf{switch}\ v\ b \parallel \mathsf{E},\ v \mapsto \{ X; \mathsf{E}_0 \} \rangle$	$\to$	$\langle b(X) \parallel \mathsf{E},\ \mathsf{E}_0 \rangle$	
(invoke)	$\langle \textbf{invoke}\ v\ X(\sigma) \parallel \mathsf{E},\ v \mapsto \{ \mathsf{E}_0; b \} \rangle$	$\to$	$\langle b(X) \parallel \mathsf{E},\ \mathsf{E}_0 \rangle$	
(jump)	$\langle f(\sigma) \parallel \mathsf{E} \rangle$	$\to$	$\langle \Theta(f) \parallel \mathsf{E} \rangle$	
(subst)	$\langle \textbf{substitute}\ [v_1' := v_1, \ldots];\ s \parallel \mathsf{E} \rangle$	$\to$	$\langle s \parallel v_1' \mapsto \mathsf{E}(v_1), \ldots \rangle$	
(lit)	$\langle \textbf{lit}\ v \leftarrow n;\ s \parallel \mathsf{E} \rangle$	$\to$	$\langle s \parallel \mathsf{E},\ v \mapsto n \rangle$	
(add)	$\langle v \leftarrow v_1\ +\ v_2;\ s \parallel \mathsf{E} \rangle$	$\to$	$\langle s \parallel \mathsf{E},\ v \mapsto \mathsf{E}(v_1)\ +\ \mathsf{E}(v_2) \rangle$	
(ifz)	$\langle \textbf{if}\ v \equiv 0\ \{ s_1 \}\ \textbf{else}\ \{ s_2 \} \parallel \mathsf{E} \rangle$	$\to$	$\langle s_1 \parallel \mathsf{E} \rangle$ $\langle s_2 \parallel \mathsf{E} \rangle$	if $\mathsf{E}(v) = 0$ otherwise

Figure 7.2: Small-step operational machine semantics of AxCut.

Figure 7.2 also defines the evaluation steps of the abstract machine. There is one rule for each kind of statement, which mirrors its typing rule. Rule *(let)* creates a constructor or destructor value by removing the part E_0 corresponding to the variable environment Γ_0 of the constructor or destructor X from the value environment E, and then adding a corresponding binding $\{ X; \mathsf{E}_0 \}$ to E. Dually, rule *(create)* creates a closure. It packages up the part E_0 of the environment corresponding to the closure environment Γ_0 together with the branches b as a new binding. Rule *(switch)* uses a constructor or destructor value by looking up the name X in the value environment, and selecting the corresponding statement $b(X)$ to execute. It then removes the binding for the constructor or destructor value and adds the value environment E_0. Rule *(invoke)* uses a closure by looking up the statement $b(X)$ corresponding to X and extending the value environment with the stored closure environment E_0. The typing rules ensure that the arguments σ of X line up exactly with the bindings in E and could hence be omitted. Rule *(jump)* transfers execution to the statement $\Theta(f)$ of top-level function f defined in program Θ. We could again omit the arguments σ here. Rule *(subst)* creates a new value environment by looking up each variable in the bindings of the old value environment. Rules *(lit)*, *(add)*, and *(ifz)* are as expected. There is no rule for **exit** since we consider a machine state with **exit** as its statement as being terminal.

The only place where parts of the value environment are reordered, duplicated, or dropped is in rule *(subst)* for explicit substitutions. Moreover, besides direct transfer of control in rule *(jump)*, there are two kinds of indirect transfer of control: rule *(switch)* where the constructor or destructor name is unknown but the branches are known, and rule *(invoke)* where the constructor or destructor name is known but the branches are unknown.

Type safety AxCut satisfies the usual theorems of progress (Theorem 55) and preservation (Theorem 56). These theorems have been shown in Idris 2 by an intrinsically typed implementation (Schuster et al., 2025b) and the totality of the step-function on machine states.

Theorem 55 (Progress)**.**
If $\vdash M$*, then either* M *is of the form* $\langle \textbf{exit}\ v \parallel E \parallel \Theta \rangle$ *with* $v \mapsto n$ *in* E *or there exists a unique machine state* M' *such that* $M \to M'$*.*

Theorem 56 (Preservation)**.**
If $\vdash M$ *and* $M \to M'$*, then* $\vdash M'$*.*

7.3 Translating Shrunk Core to AxCut

We now discuss how to translate the fragment Shrunk Core derived in Section 7.1 into AxCut. To do so, we have to insert explicit substitutions and annotate the environment captured by closures. As the placement of explicit substitutions may impact performance, the overall goal is to insert substitutions in such a way that the register pressure is reduced as much as possible, that is, such that there are as few variables in the environment as possible. Therefore, following garbage-free reference counting (Reinking et al., 2021), the substitutions aim to drop unneeded variables as early as possible and duplicate variables as late as possible. To illustrate the idea, consider the following example.

Example 57. The top-level function consTwice receives an integer and a list and returns the same list with the integer prepended twice.

$$\begin{array}{l}\textbf{def}\ \mathsf{consTwice}(v :^{\mathsf{prd}} \mathsf{i64},\ l_0 :^{\mathsf{prd}} \mathtt{List}[\mathsf{i64}],\ \alpha :^{\mathsf{cns}} \mathtt{List}[\mathsf{i64}])\,\{\\ \quad \langle \mathtt{Cons}(v,\ l_0) \mid \tilde{\mu} l_1.\langle \mathtt{Cons}(v,\ l_1) \mid \alpha \rangle \rangle\\ \}\end{array}$$

We insert an explicit substitution before each statement, where for every free variable in the statement we substitute this variable for itself in an order dictated by what variables the statement consumes. This will exchange and weaken variables appropriately. When a variable occurs free more than once, such as v here, we have to contract and rename it. To arrive in AxCut, we also have to use its uniform notation, so the outer cut becomes a **let** statement, while the inner cut becomes an **invoke** statement. The example in AxCut then looks as follows.

$$\begin{array}{l}\textbf{def}\ \mathsf{consTwice}(v :^{\mathsf{prd}} \mathsf{i64},\ l_0 :^{\mathsf{prd}} \mathtt{List}[\mathsf{i64}],\ \alpha :^{\mathsf{cns}} \mathtt{List}[\mathsf{i64}])\,\{\\ \quad \textbf{substitute}\,[v := v,\ \alpha := \alpha,\ v_0 := v,\ l_0 := l_0];\\ \quad \textbf{let}\ l_1 = \mathtt{Cons}(v_0,\ l_0);\\ \quad \textbf{substitute}\,[v := v,\ l_1 := l_1,\ \alpha := \alpha];\\ \quad \textbf{invoke}\ \alpha\ \mathtt{Cons}(v,\ l_1)\\ \}\end{array}$$

The arguments of the `Cons` constructors now exactly match the appropriate part of the context.

The translation is presented in Figure 7.3. To simplify the translation a bit, we assume all variables in the program to be unique, i.e., there is no shadowing. The function for statements $\mathcal{X}[\![\ \cdot\ ]\!]_{\odot}$ receives as an additional input a type environment, which is supposed to be the environment the currently translated statement should be typed in, according to the typing rules of AxCut. For the translation of top-level definitions, the annotated signature Γ is the environment in which its body s is typed, so we translate s with environment Γ. While there may be variables that are not used, i.e., that do not occur free in s, we do not have to insert a substitution here, as we will insert one at the beginning of the translation for each statement. The only exceptions, where no substitution is inserted, are **exit** statements, which end the program anyway, and the case of conditionals. For the latter, we can relinquish a substitution at the beginning since, again, there will be an appropriate substitution in each of the branches and there are no newly bound variables in the branches. Thus, no additional register will become occupied before descending into the substatements.

The other cases where no variable is newly bound are cuts of constructors or destructors with a variable, which become **invoke** statements, and the calls of top-level functions. These are leaves of the AST without any substatements and the typing rules precisely dictate what the environment has to look like. Note, however, that some variables may have to be duplicated. For example, in a call to a top-level definition $f(\Gamma_0)$, some variables from Γ may occur multiple times in Γ_0. To keep the variables in the environment unique, we pick fresh names for the new environment after the substitution, and we write Γ_0^{f} to indicate this (it of course suffices to only pick fresh names for duplicated variables).

$\mathcal{X}[\![\cdot]\!] : \mathit{Definition}_{\mathsf{Shrunk\ Core}} \to \mathit{Definition}_{\mathsf{AxCut}}$

$$\mathcal{X}[\![\mathbf{def}\, f(\Gamma)\,\{\, s \,\}]\!] = \mathbf{def}\, f(\Gamma)\,\{\, \mathcal{X}[\![s]\!]_{\Gamma} \,\}$$

$\mathcal{X}[\![\cdot]\!]_{\odot} : \mathit{Statement}_{\mathsf{Shrunk\ Core}} \times \mathit{Context}_{\mathsf{AxCut}} \to \mathit{Statement}_{\mathsf{AxCut}}$

$$\begin{array}{lll}
\mathcal{X}[\![\langle K(\Gamma_0) \mid \tilde{\mu} x.s \rangle]\!]_{\Gamma} & = & \mathbf{substitute}\,[\Gamma' := \Gamma',\, \Gamma_0^{\mathsf{f}} := \Gamma_0]; \\
 & & \mathbf{let}\, x = K(\Gamma_0^{\mathsf{f}});\, \mathcal{X}[\![s]\!]_{\Gamma',\, x} \\
\text{where} & & \Gamma' = \mathsf{freeVars}(s) \subset \Gamma \\
\mathcal{X}[\![\langle \mu\alpha.s \mid D(\Gamma_0) \rangle]\!]_{\Gamma} & = & \mathbf{substitute}\,[\Gamma' := \Gamma',\, \Gamma_0^{\mathsf{f}} := \Gamma_0]; \\
 & & \mathbf{let}\, \alpha = D(\Gamma_0^{\mathsf{f}});\, \mathcal{X}[\![s]\!]_{\Gamma',\, \alpha} \\
\text{where} & & \Gamma' = \mathsf{freeVars}(s) \subset \Gamma \\
\mathcal{X}[\![\langle K(\Gamma_0) \mid \alpha \rangle]\!]_{\Gamma} & = & \mathbf{substitute}\,[\Gamma_0^{\mathsf{f}} := \Gamma_0,\, \alpha := \alpha];\, \mathbf{invoke}\, \alpha\, K(\Gamma_0^{\mathsf{f}}) \\
\mathcal{X}[\![\langle x \mid D(\Gamma_0) \rangle]\!]_{\Gamma} & = & \mathbf{substitute}\,[\Gamma_0^{\mathsf{f}} := \Gamma_0,\, x := x];\, \mathbf{invoke}\, x\, D(\Gamma_0^{\mathsf{f}}) \\
\mathcal{X}[\![\langle x \mid \mathbf{case}\,\{\, K_1(\Gamma_1) \Rightarrow s_1,\, \ldots \,\} \rangle]\!]_{\Gamma} & = & \mathbf{substitute}\,[\Gamma' := \Gamma',\, x^{\mathsf{f}} := x]; \\
 & & \mathbf{switch}\, x^{\mathsf{f}}\,\{\, K_1(\Gamma_1) \Rightarrow \mathcal{X}[\![s_1]\!]_{\Gamma',\Gamma_1},\, \ldots \,\} \\
\text{where} & & \Gamma' = \bigcup_i \mathsf{freeVars}(s_i) \subset \Gamma \\
\mathcal{X}[\![\langle \mathbf{new}\,\{\, D_1(\Gamma_1) \Rightarrow s_1,\, \ldots \,\} \mid \alpha \rangle]\!]_{\Gamma} & = & \mathbf{substitute}\,[\Gamma' := \Gamma',\, \alpha^{\mathsf{f}} := \alpha]; \\
 & & \mathbf{switch}\, \alpha^{\mathsf{f}}\,\{\, D_1(\Gamma_1) \Rightarrow \mathcal{X}[\![s_1]\!]_{\Gamma',\, \Gamma_1},\, \ldots \,\} \\
\text{where} & & \Gamma' = \bigcup_i \mathsf{freeVars}(s_i) \subset \Gamma \\
\mathcal{X}[\![\langle \mu\alpha.s \mid \mathbf{case}\,\{\, K_1(\Gamma_1) \Rightarrow s_1,\, \ldots \,\} \rangle]\!]_{\Gamma} & = & \mathbf{substitute}\,[\Gamma'^{\mathsf{f}} := \Gamma',\, \Gamma_0 := \Gamma_0]; \\
 & & \mathbf{create}\, \alpha = \Gamma_0\,\{\, K_1(\Gamma_1) \Rightarrow \mathcal{X}[\![s_1]\!]_{\Gamma_1,\, \Gamma_0},\, \ldots \,\}; \\
 & & \mathcal{X}[\![s[\Gamma' \mapsto \Gamma'^{\mathsf{f}}]]\!]_{\Gamma'^{\mathsf{f}},\, \alpha} \\
\text{where} & & \Gamma_0 = \bigcup_i \mathsf{freeVars}(s_i) \subset \Gamma \quad \Gamma' = \mathsf{freeVars}(s) \subset \Gamma \\
\mathcal{X}[\![\langle \mathbf{new}\,\{\, D_1(\Gamma_1) \Rightarrow s_1,\, \ldots \,\} \mid \tilde{\mu} x.s \rangle]\!]_{\Gamma} & = & \mathbf{substitute}\,[\Gamma'^{\mathsf{f}} := \Gamma',\, \Gamma_0 := \Gamma_0] \\
 & & \mathbf{create}\, x = \Gamma_0\,\{\, D_1(\Gamma_1) \Rightarrow \mathcal{X}[\![s_1]\!]_{\Gamma_1,\, \Gamma_0},\, \ldots \,\}; \\
 & & \mathcal{X}[\![s[\Gamma' \mapsto \Gamma'^{\mathsf{f}}]]\!]_{\Gamma'^{\mathsf{f}},\, x} \\
\text{where} & & \Gamma_0 = \bigcup_i \mathsf{freeVars}(s_i) \subset \Gamma \quad \Gamma' = \mathsf{freeVars}(s) \subset \Gamma \\
\mathcal{X}[\![\langle n \mid \tilde{\mu} x.s \rangle]\!]_{\Gamma} & = & \mathbf{substitute}\,[\Gamma' := \Gamma'];\, \mathbf{lit}\, x \leftarrow n;\, \mathcal{X}[\![s]\!]_{\Gamma',\, x} \\
\text{where} & & \Gamma' = \mathsf{freeVars}(s) \subset \Gamma \\
\mathcal{X}[\![\langle x_1 + x_2 \mid \tilde{\mu} x.s \rangle]\!]_{\Gamma} & = & \mathbf{substitute}\,[\Gamma' := \Gamma'];\, x \leftarrow x_1 + x_2;\, \mathcal{X}[\![s]\!]_{\Gamma',\, x} \\
\text{where} & & \Gamma' = (\{\, x_1 \,\} \cup \{\, x_2 \,\} \cup \mathsf{freeVars}(s)) \subset \Gamma \\
\mathcal{X}[\![\mathbf{if}\, x \equiv 0\,\{\, s_1 \,\}\, \mathbf{else}\,\{\, s_2 \,\}]\!]_{\Gamma} & = & \mathbf{if}\, x \equiv 0\,\{\, \mathcal{X}[\![s_1]\!]_{\Gamma} \,\}\, \mathbf{else}\,\{\, \mathcal{X}[\![s_2]\!]_{\Gamma} \,\} \\
\mathcal{X}[\![f(\Gamma_0)]\!]_{\Gamma} & = & \mathbf{substitute}\,[\Gamma_0^{\mathsf{f}} := \Gamma_0];\, f(\Gamma_0^{\mathsf{f}}) \\
\mathcal{X}[\![\mathbf{exit}\, x]\!]_{\Gamma} & = & \mathbf{exit}\, x
\end{array}$$

Figure 7.3: Translation from Shrunk Core to AxCut.

Next, consider the cases where a constructor or destructor meets a $\tilde{\mu}$- or μ-binding in a cut. These are translated into **let** statements. Here, typing dictates that the part Γ_0 of the environment passed to the constructor or destructor must be the right-most part. Since there is no shadowing, the minimal environment in which the translated remaining statement s can be typed consists of the bindings Γ' for the free variables of s, which must be contained in Γ, extended with the newly bound variable. Hence, this is the environment passed into the translation of s. The whole substitution then consists of the bindings for Γ' and Γ_0, in this order. There may be variables in Γ which occur in both Γ' and Γ_0, even multiple times in the latter, and the substitution will clone them accordingly. Again, to keep variables unique, we pick fresh names for Γ_0. There is no need to pick fresh names for the free variables of s, since there can be no duplications within this set. All variables in Γ that occur in neither Γ' nor Γ_0 are dropped. The order of the entries in Γ' can be arbitrary, but to keep the number of exchanges small, it may be a good idea to arrange them in such a way that as few variables as possible change positions.

The cases where a (co)pattern match is cut with a variable are translated into a **switch**. Here, the variable matched on has to be right-most in the environment, and the remaining environment Γ' is shared among the branches, so the latter can be chosen to be the union of the free variables in the statements of all branches. The environment for the recursive translation of the substatement in each branch consists of Γ' extended with the newly bound variables for that branch. We pick a fresh name for the variable we match on, as it may also occur in Γ'.

Similarly, when a (co)pattern match is bound to a variable, which we translate into a **create** statement, the closure environment Γ_0 consists of the union of the free variables in all branches. It has to be the right-most part in the environment. For the recursive translation of the substatement in each branch, we extend Γ_0 with the variables bound for the corresponding branch. The remaining environment Γ' consists again of the free variables of the remaining statement s. For the recursive translation of s, Γ' is again extended with the newly bound variable. Since it is possible that there are variables that occur in both Γ' and Γ_0, we have to pick fresh names for one of them to keep variables unique. But since both environments occur in substatements, we have to reflect this renaming in the corresponding substatement. We choose to rename Γ' as it only occurs in one substatement s and write $s[\Gamma' \mapsto \Gamma'^{\mathsf{f}}]$ to indicate the renaming in s.

The cases for integer literals and arithmetic operations are similar. The only difference for arithmetic operations is that the variables used as arguments cannot be dropped, even if they do not occur free in the remaining statement, as they are not subjected to the ordered typing discipline.

The inserted explicit substitutions often have little or even no computational content if the context is unchanged. In the latter case, we can of course simply leave them out. To avoid calculating the free variables of substatements repeatedly, we can annotate them in one prepass. However, when we substitute renamed variables into substatements as in the **create** cases, we have to update the annotations accordingly.

The above translation again preserves typeability.

Theorem 58 (Typeability Preservation)**.**
Let s be a statement in Shrunk Core and Γ a typing context.
If $\Theta \vdash \mathbf{def} f(\Gamma)\{\, s \,\}$ OK, then $\Theta \vdash \mathcal{X}[\![\, \mathbf{def} f(\Gamma)\{\, s \,\}]\!]$ OK.
If $\Gamma \vdash s$, then $\Gamma \vdash \mathcal{X}[\![\, s \,]\!]_\Gamma$.

Proof Sketch. By induction. As the types are not affected by this translation, the main problem lies in adapting the typing contexts to the ordered typing discipline in the rules of AxCut. This is not too difficult, however, since in the premise $\Gamma \vdash \sigma : \Gamma'$ of the SUBSTITUTE rule, we can still access the context Γ arbitrarily for σ. □

8 Code Generation - From AxCut to Machine Code

In this chapter, we show how to generate machine code from our lower-level intermediate languages AxCut. On the one hand, AxCut is, at its core, a normal form of classical sequent calculus as derived in the previous chapter, on the other hand, it admits a straightforward translation into machine code. Indeed, the language constructs of AxCut already closely correspond to concepts in machine code. For clarity, in this section, we explain our translation to a subset of RISC-V (Waterman et al., 2014). There are other implementations targeting x86-64 and AArch64, which we briefly discuss in Section 9.1.

8.1 RISC-V

Programs in RISC-V are a list of instructions, interspersed with labels l. Each instruction consists of a mnemonic followed by one to three operands. The operands are either registers or offsets, where the latter can be labels or immediate integers. For ease of presentation, we do not take into consideration that the immediate integers of instructions can often only be 12-bit integers[7]. Moreover, we freely use pseudo-instructions, which have to be desugared into actual instructions. For example, the `mv` instruction is defined in terms of `addi`. In RISC-V there are typically 32 registers, which we denote as `x0,...,x31`, each holding a machine integer. Register `x0` is special in that it always contains the value `0`.

Syntax of RISC-V

Instructions I	::=	l :	label
	|	`add` r r r	add registers
	|	`addi` r r i	add immediate
	|	`mv` r r	move register
	|	`li` r i	load immediate
	|	`la` r l	load address
	|	`j` o	jump
	|	`jr` r o	jump register
	|	`beq` r r o	branch if equal
	|	`lw` r i r	load word
	|	`sw` r i r	store word
	|	`ecall`	system call

Registers	r ::=	`x0`, ..., `x31`
Offsets	o ::=	l | i
Labels	l ::=	`l0` | `l1` | ...
Immediates	i ::=	`0` | `1` | ...

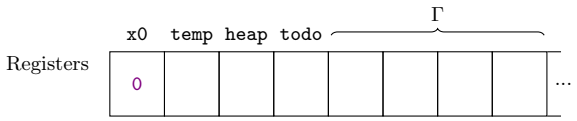
The addition instructions `add` and `addi` add their second and third operand and store the result into the first operand register. The difference is that the third operand is a register for `add` but an immediate for `addi`. Instructions for other arithmetic operations are similar. The `mv` instruction copies the value of the second register into the first register. The load instruction `li` loads an immediate into its register operand, and the load instruction `la` loads an address specified by a label into its register operand. Direct jump instructions `j` jump to an address which is specified by a label or an immediate offset relative to the program counter. Indirect jumps `jr` add the immediate offset operand to the register operand and jump to the resulting address. The conditional instruction `beq` compares its operands and jumps to the given label

[7] This is because instructions all have the same fixed length. Bigger integers actually first need to be loaded into a register, which is then used instead of the immediate operand.

or immediate. Moreover, there are instructions `lw` and `sw` for loading and storing a word from memory. They both add their immediate operand to their second operand register to obtain a memory address. Then they load into or store from their first register operand. Finally, instruction `ecall` makes system calls.

8.2 Overview

Typing judgments in AxCut are of the form $\Theta \mid \Gamma \vdash s$, where Γ is the type environment and s is a statement. The translation to machine code is defined over typing derivations, as it uses the type environment Γ. A statement corresponds to a sequence of machine instructions. At each point in the program, the environment Γ describes the content of the hardware registers (Appel, 1992). Each variable in Γ corresponds to two adjacent registers, corresponding to the two components of values in the operational semantics in Figure 7.2. They are assigned from left to right, starting with the first register not reserved for other purposes. We have three reserved registers, one scratch register `temp` and two registers, `heap` and `todo`, for memory management.



Since register `x0` is always `0`, the first variable in Γ occupies registers `x4` and `x5`, the second one occupies registers `x6` and `x7`, and so on. In the above illustration, there are thus two variables in Γ. Consequently, there can be at most 14 variables in the environment at the same time. This restriction can be lifted, of course, by spilling any further variables to memory.

8.3 Code Generation By Example

In this section, we illustrate code generation on example programs, before giving the full formal definition in Section 8.4.

8.3.1 Compiling Programs and Sequents

We first illustrate the fragment of AxCut that neither requires nor mentions any data or codata types, i.e., that only consists of calls to top-level functions, **substitute** statements, and external operations.

Definitions are Labels, Calls are Direct Jumps

In AxCut, a program is a list of type declarations and top-level function definitions. The signature of a top-level function is the type environment Γ it assumes. To call a top-level function, the type environment Γ must exactly match what is assumed by the definition, which we track in the program Θ. In the following example, we have two definitions, main and loop, and from main, we call loop.

AxCut

```
def main(Γ0) {
  ...
  loop(Γ)
}
def loop(Γ) {
  ...
}
```

$$\frac{\textbf{def}\ \mathsf{loop}(\Gamma) \in \Theta}{\Theta \mid \Gamma \vdash \mathsf{loop}(\Gamma)}\ [\textsc{Call}]$$

Machine Code

```
main:
  ...
  j loop
loop:
  ...
```

We translate each top-level function to the labeled translation of its body statement, and we translate calls to direct jumps to the label. A crucial property, which stands in contrast to most other intermediate representations, is that in our sequent-calculus-based intermediate representations there is no a priori fixed concept of returning functions with a fixed calling convention. Instead, signatures specify a protocol of interaction between a producer and a consumer, which includes synchronous communication, i.e., returning function calls, as a special case. Other examples would be throwing an exception, as in our running example with early return, and coroutining with the caller. Our approach thus subsumes, for example, standard continuation-passing style and double-barreled continuation-passing style, and enables mixing different such calling and returning conventions in a program.

Explicit Substitutions are Parallel Moves

Before a call to a top-level function, the register contents must exactly be what the destination expects. This means, in particular, that the order of the elements of Γ matters. To change the order of variables in AxCut, we use a **substitute** statement, which subsumes the structural rule of exchange. In the following example, the rest of the statement runs in an environment where the order of x and y is swapped, i.e., the old environment is $\Gamma = x :^{\mathsf{prd}} \mathsf{i64}, y :^{\mathsf{prd}} \mathsf{i64}$, the new environment is $\Gamma' = y :^{\mathsf{prd}} \mathsf{i64}, x :^{\mathsf{prd}} \mathsf{i64}$, and the right-hand side in the **substitute** statement is $\sigma = y, x$.

AxCut

$$\frac{x :^{\mathsf{prd}} \mathsf{i64}, y :^{\mathsf{prd}} \mathsf{i64} \vdash y, x : y :^{\mathsf{prd}} \mathsf{i64}, x :^{\mathsf{prd}} \mathsf{i64} \quad y :^{\mathsf{prd}} \mathsf{i64}, x :^{\mathsf{prd}} \mathsf{i64} \vdash \ldots}{x :^{\mathsf{prd}} \mathsf{i64}, y :^{\mathsf{prd}} \mathsf{i64} \vdash \textbf{substitute}\,[y := y, x := x]; \ldots}\ [\textsc{Substitute}]$$

Machine Code

```
mv temp x5
mv x5 x7
mv x7 temp
...
```

We translate explicit substitutions that change the order of variables to parallel moves (Rideau et al., 2008), using the temporary register `temp`. Only the registers of the variables that change their position participate in the moves. Each variable occupies two registers, but integers only use the second one, so we do not have to swap registers `x4` and `x6` here. The structural rules of weakening and contraction will be discussed in Section 8.3.3.

Externs are System Calls and Hardware Instructions

Every program runs in an external environment and eventually needs to terminate, output a result, or interact with the outside world in another way. For this purpose, AxCut includes external statements like **exit**, arithmetic expressions, and conditionals, which generally operate on external types like built-in machine integers. In this example, we halt execution and exit with the code in variable v.

AxCut

$$\frac{v :^{\mathsf{prd}} \mathsf{i64} \in \Gamma}{\Gamma \vdash \textbf{exit}\, v}\ [\textsc{Exit}]$$

Machine Code

```
li x17 93
mv x10 x7
ecall
```

We translate these external statements either to system calls, as in this example, or to hardware instructions, like addition and branching. Here, as required by the Linux ABI, we load the syscall number to `x17`, move its argument to `x10`, and call the environment with `ecall`.

Summary

We have seen that it is possible to write programs in AxCut which just act on external types, without considering logical types. The translation of external statements, like arithmetic operations or conditionals, depends on the target platform and consists of corresponding hardware instructions or system calls. Top-level functions and calls to them are translated to assembly labels and direct jumps. The signature of a top-level function communicates what variables the function expects to be present, that is, which registers must contain what kind of content. As the reordering of variables is accomplished by explicit substitutions, the code for the latter performs the corresponding parallel moves of registers.

This makes AxCut in some sense a low-level language. At the same time, it is a fragment of classical sequent calculus, with logical types, producers, consumers, and cuts between them, which makes it a high-level language. Next, we will look at their translation to machine code.

8.3.2 Compiling Producers and Consumers

User-defined data and codata types are specified by their signatures, which can be arbitrarily mutually recursive. In the following example, we define a data type `Maybe` with two constructors, which contain either some integer or none. Exactly the same considerations are true for codata types as well.

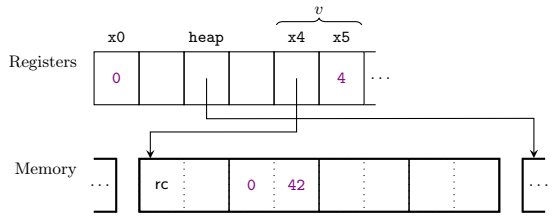
AxCut

data `Maybe` {
  `None`(), `Some`(x :$^{\text{prd}}$ i64)
}

Machine Code

—

Illustration



In machine code, signatures of data and codata types do not appear explicitly. However, they implicitly define mutually dependent invariants about how values are laid out when stored in memory, how they are loaded back into registers, and where in the code this happens. These invariants are established and maintained by all programs by construction.

The picture shows an example where the first variable in a sequent $\Gamma = v :^{\text{prd}} \texttt{Maybe}, \ldots$ is bound to `Some`(42). Its tag 4 resides in register `x5`, and register `x4` contains a pointer to a memory block. The latter consists of four fields, each of which can store two words. The first field serves as a header, containing a reference count, which we will come back to in Section 8.3.3, and an as of yet unused word. The argument 42 is stored in the second field, corresponding to the two registers the argument had occupied before it was stored. More arguments would be stored in the other fields and, if necessary, in more blocks linked together.

In the following examples, we will explain this further. We consider **let** and **switch** statements from the viewpoint of data types and **new** and **invoke** from the viewpoint of codata types. We only do so to help intuition, and everything we say is valid from the dual viewpoint as well.

Constructors are Tags, Fields are Memory Blocks, Pattern Matches are Jump Tables

Let us now consider an example corresponding to the above picture. We construct a producer of signature `Maybe` with **let** and immediately match on it with **switch**. The singleton sequent $v_0 :^{\text{prd}}$ i64 becomes the field environment of the constructor `Some`(v_0). Variable v_0 is no longer available in the rest of the program, as it is moved into the constructor. In the rest of the statement, the only variable in scope is v, a producer of type `Maybe`. When we match on v, there are two subderivations, one for each of the two branches, in accordance with the signature

of `Maybe`. Variable v is removed from the type environment, and, depending on the branch, the constructor fields are added. The above picture corresponds to the situation after the **let** but before the **switch** (also cf. the illustration in Section 7.2.1 again).

AxCut

$$\dfrac{\dfrac{\vdash \ldots \qquad x :^{\mathsf{prd}} \mathsf{i64} \vdash \ldots}{v :^{\mathsf{prd}} \texttt{Maybe} \vdash \textbf{switch}\ v\ \{\ \texttt{None}() \Rightarrow \ldots,\ \texttt{Some}(x) \Rightarrow \ldots\ \}}\ \textsc{Switch}}{v_0 :^{\mathsf{prd}} \mathsf{i64} \vdash \textbf{let}\ v = \texttt{Some}(v_0);\ \textbf{switch}\ v\ \{\ \texttt{None}() \Rightarrow \ldots,\ \texttt{Some}(x) \Rightarrow \ldots\ \}}\ \textsc{Let}$$

Machine Code

```
sw x5 12 heap      jr x5 table      None:      Some:
ACQUIRE x4         table:             ...        RELEASE x4
li x5 4              j None                     lw x5 12 x4
                     j Some                     ...
```

We translate the **let** statement to the machine code in the first column. The variable v_0 is in register `x5`. We store it into a fresh block of memory on the heap at offset `12`. We acquire this block of memory into register `x4` with the macro `ACQUIRE`, the details of which we explain in Section 8.3.3. We then load the constructor tag for `Some` into register `x5`, overwriting the contents of v_0, which is gone now.

We translate the **switch** statement to the indirect jump and jump table `table` in the middle. Constructor tags are offsets into jump tables. The order of the jump-table entries is determined by the signature. The offsets are multiples of `4`, the length of an instruction, hence the tag for `Some` is `4`. After jumping to the branch, we first load the field environment (as illustrated by reading the picture in Section 7.2.1 in the reverse direction), if there is any, and then execute the rest of the statement. In the branch for `Some`, the environment is loaded from the block of memory pointed to by `x4` and the block is released with the macro `RELEASE`, the details of which we also explain in Section 8.3.3. The fields are moved into registers, which overwrites the matched variable.

Objects are Virtual Tables, Closures are Memory Blocks, Destructor Invocations are Indirect Jumps

We now consider the codata type of functions `Fun`[i64, i64] from Example 46. Intuitively, a codata type is defined by what it can do rather than what it is. The only thing we can do with a function is apply it to a value and a continuation to return the result to. Hence, `Fun`[i64, i64] has a single destructor `apply` which accepts a producer and a consumer of an integer as arguments. In the following example, we create a function f with **create**, which defines the closure environment $(v :^{\mathsf{prd}} \mathsf{i64})$ and a statement that implements the application destructor. The statement must use the variable v, as well as the parameters x and α. Variable v is moved into the closure and not available in the remaining statement. The function f is a producer of type `Fun`[i64, i64]. We can use f in the remaining statement with **invoke** on a destructor. In the example, we apply f to arguments r and β that were added to the environment after **create** but before **invoke**. The environment must consist of exactly r, β, and f, in this order.

AxCut

$$\dfrac{\dfrac{}{r :^{\mathsf{prd}} \mathsf{i64},\ \beta :^{\mathsf{cns}} \mathsf{i64},\ f :^{\mathsf{prd}} \texttt{Fun} \vdash \textbf{invoke}\ f\ \texttt{apply}(r, \beta)}\ \textsc{Invoke} \quad \ldots}{f :^{\mathsf{prd}} \texttt{Fun} \vdash \ldots\ \textbf{invoke}\ f\ \texttt{apply}(r, \beta) \qquad\qquad x :^{\mathsf{prd}} \mathsf{i64},\ \alpha :^{\mathsf{cns}} \mathsf{i64},\ v :^{\mathsf{prd}} \mathsf{i64} \vdash \ldots}\ \textsc{Create}$$
$$v :^{\mathsf{prd}} \mathsf{i64} \vdash \textbf{create}\ f = (v :^{\mathsf{prd}} \mathsf{i64})\{\ \texttt{apply}(x, \alpha) \Rightarrow \ldots\ \};\ \ldots\ \textbf{invoke}\ f\ \texttt{apply}(r, \beta)$$

Machine Code

```
sw x5 12 heap      table:         apply:
ACQUIRE x4           j apply        RELEASE x8
la x5 table                         lw x9 12 x8
...                                 ...
jr x9 0
```

We translate the **create** statement to first store register `x5`, which corresponds to v, in a block of memory on the heap, which we then acquire into register `x4`. This is analogous to how the field environment of a constructor is stored. We then generate the virtual table `table` with one entry in the middle and load its address into register `x5`. This is dual to how we load constructor tags. While in this example there is only a single entry in the virtual table, in general, there can be zero or more. The implementation for each branch starts by loading the closure environment from the corresponding block of memory and then releasing the block. In the example, this block is pointed to by `x8`, occupied by the third variable in the environment. In general, it must come after all destructor arguments, which are determined by the signature of `Fun`[i64, i64].

We translate the **invoke** statement to an indirect jump to the virtual table, which must be in register `x9`. This means that the instructions between **create** and **invoke** must have rearranged the registers, putting the arguments for the destructor `apply` into registers `x4`, `x5` and `x6`, `x7`. The offset into the virtual table is given by the tag of destructor `apply`, which in this case is `0`. If there was more than one destructor, it could be different. This implements dynamic dispatch.

Exploiting the Symmetry between Data and Codata

Finally, we demonstrate some additional flexibility we gain from the symmetry of data and codata types. Consider the signature of a data type `Result` that is either `Ok` or an `Err`. Types like this are common return types in functions that want to signal an error condition to their caller. Moreover, these functions are usually chained together and some languages provide syntactic sugar for this.

AxCut

```
data Result {
  Ok(x :prd i64), Err(i :prd i64)
}
```

Machine Code

—

When compiling a chain of matches naively, each function in such a chain will store the result in memory upon return, which the caller then immediately investigates. In AxCut, however, we can represent pattern-matching contexts directly. While producers of type `Result` represent expressions, consumers of the type represent their contexts. We create such contexts with **create**, as in the following example.

AxCut

$$\frac{k :^{\mathsf{cns}} \texttt{Result} \vdash f(k) \qquad x :^{\mathsf{prd}} \mathsf{i64} \vdash \ldots \qquad i :^{\mathsf{prd}} \mathsf{i64} \vdash \ldots}{\textbf{create}\ k\ =\ ()\{\,\texttt{Ok}(x) \Rightarrow \ldots, \texttt{Err}(i) \Rightarrow \ldots\,\};\ f(k)}\ [\textsc{Create}]$$

Machine Code

```
li x4 null       table:       Ok:        Err:
la x5 table        j Ok         ...        ...
j f                j Err
```

We avoid storing intermediate results in memory, as we directly pass this context k to the top-level function f. Intuitively, the context is a stack frame with two return addresses, known as vectored returns [Peyton Jones (1992); Marlow et al. (2007)]. The statement in f then invokes k with the corresponding destructor to select the control path, for example, with **invoke** $k\,\texttt{Err}(e)$. The values are returned in registers. This is another example of a definable protocol of interaction in AxCut.

Summary

We have seen how we translate the logical fragment of AxCut to RISC-V machine code. The translation is straightforward, since language constructs in AxCut closely correspond to concepts in machine code. The code for creating and using producers and consumers is dual, in congruence with the abstract machine steps. Both **let** and **create** store values onto the heap. But for **let**, the block of memory is paired with a tag and contains the corresponding arguments, and for **create**, it is paired with a jump table and contains the closure environment. Similarly, both **switch** and **invoke** perform indirect jumps with a tag being used as an offset into a jump table. But for **switch**, the offset is unknown as it was bound by **let**, and for **invoke**, the jump table is unknown as it was bound by **create**. In both cases, at the beginning of each branch, the values stored in the memory block are loaded back into registers, as determined by the type signatures.

So far, we have assumed that all variables, except in external statements, are used linearly. Next, we describe how we translate the non-linear use of variables.

8.3.3 Compiling Weakening and Contraction

We first consider memory management for the fragment of AxCut that we have seen so far, without the structural rules of weakening and contraction. Since values are used linearly, memory management is trivial.

The Linear Fragment

We divide memory into blocks of the same size, which we store in a free list in register `heap`. It is precisely the statements **let** and **create**, corresponding to cuts with an activation rule (i.e., μ or $\tilde{\mu}$), that acquire memory, and precisely the statements **switch** and **invoke**, corresponding to cuts with an axiom rule, also known as deactivation, that release it.

Acquire and release When we store values, we acquire a fresh block of memory. We move register `heap` into the destination, and load the next element of the free list, stored at offset `0`, into `heap`. If the free list is empty, we fall back to bump allocation, as will be discussed in Section 8.4. Alternatively, we could prepare the heap to be a linked list of free blocks at program startup to make just this code work.

When we load values, we release the block of memory. We store the current head of the free list at offset `0` in the block in the destination register, and move this block into register `heap`.

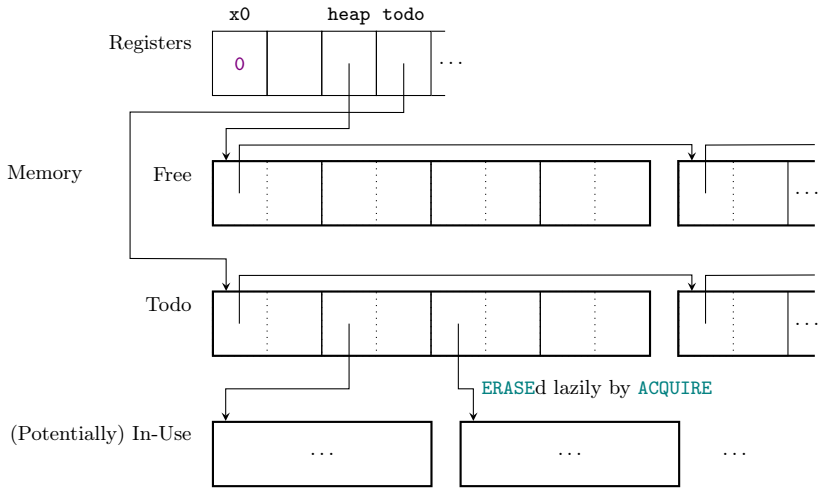
```
ACQUIRE x4   =   mv x4 heap          RELEASE x4   =   sw heap 0 x4
                 lw heap 0 heap                       mv heap x4
```

The Non-Linear Rest

This simple implementation works for a linear language. However, AxCut also allows for non-linear use of variables with explicit weakening and contraction as part of substitutions. Therefore, as already illustrated in Section 8.3.2, we add a reference count to the start of each block of

memory. When inspecting this reference count, we obviously have a reference to it. It therefore counts the number of other references, starting at `0`. Reference counting is an exceptionally good fit for AxCut, because of a trivial but profound observation: As long as values are used linearly, their reference count never changes. In other words, it is only the structural rules of weakening and contraction that require us to inspect and update reference counts, and consequently to track them at all. Even though we do track them, after a quick uniqueness check, we enter the fast path described above. Since continuation frames are typically used linearly, we hence get fast code for stack-like usage, even though we do not maintain a stack. Moreover, the typing rules of AxCut naturally enforce that there are no cycles between blocks of memory.

More specifically, we use lazy reference counting (Weizenbaum, 1963, 1969). In combination with the allocation of equal-sized blocks that we use, we have an implementation of constant-time reference counting (Lam and Parreaux, 2024). With this strategy, the only waste of memory is internal fragmentation due to the fact that each memory block might be larger than needed. The main difference from the system of Lam and Parreaux (2024) is that we retain the fast path for linear use of values. We do so by maintaining two free lists, one whose blocks can be used immediately, which we also call the linear free list, pointed to by register `heap`, and one where the children of blocks must still be traversed and erased, which we call the lazy free list, pointed to by register `todo`.



We acquire blocks from the linear free list as explained above, only falling back to the lazy free list if the linear free list is empty. Releasing a block whose reference count is `0` puts it onto the linear free list. There is no need to process the children since they are moved into registers. Blocks are only put onto the lazy free list if the last reference to them is erased by weakening. The above definitions of `ACQUIRE` and `RELEASE` have to be adapted accordingly, as will be discussed in Section 8.4.

Share The structural rule of contraction allows for the duplication of formulas in a sequent. We express the use of the contraction rule by mentioning the same variable more than once in the right-hand side of an explicit substitution. Here, we share the variable f into variables f_1 and f_2. It is precisely in these cases that we have to increment the reference count of the block of memory of f. The substitutions tell us exactly when, where, and how much these increments happen. The block of memory of f is in register `x4`. We use the temporary register `temp` to do the increment.

AxCut

$$\frac{f :^{\mathsf{cns}} \texttt{Fun} \vdash f, f : f_1 :^{\mathsf{cns}} \texttt{Fun}, f_2 :^{\mathsf{cns}} \texttt{Fun} \quad f_1 :^{\mathsf{cns}} \texttt{Fun}, f_2 :^{\mathsf{cns}} \texttt{Fun} \vdash \ldots}{f :^{\mathsf{cns}} \texttt{Fun} \vdash \textbf{substitute}\,[f_1 := f, f_2 := f]; \ldots}\ [\textsc{Substitute}]$$

Machine Code

```
lw temp 0 x4
addi temp temp 1
sw temp 0 x4
...
```

Erase The structural rule of weakening allows for ignoring irrelevant formulas in a sequent. Since explicit substitutions define precisely the type environment by their left-hand sides, we express weakening in AxCut by not mentioning a variable in any right-hand side of a substitution. Here we erase the variable y.

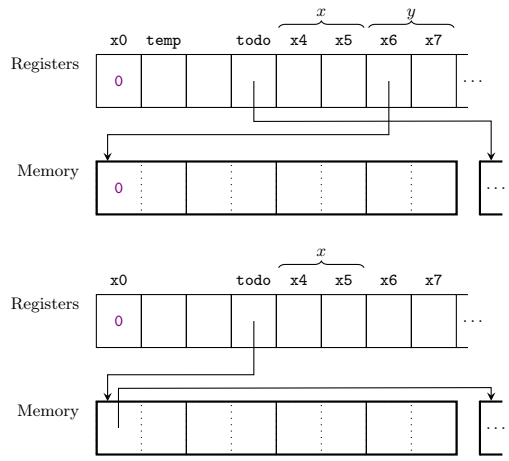
AxCut

$$\frac{x :^{\mathsf{prd}} \texttt{Result}, y :^{\mathsf{prd}} \texttt{Maybe} \vdash x : x :^{\mathsf{prd}} \texttt{Result} \qquad x :^{\mathsf{prd}} \texttt{Result} \vdash \ldots}{x :^{\mathsf{prd}} \texttt{Result}, y :^{\mathsf{prd}} \texttt{Maybe} \vdash \textbf{substitute}\,[x := x]; \ldots}\ [\textsc{Substitute}]$$

Machine Code

```
lw temp 0 x6
beq temp x0 then
  addi temp temp -1
  sw temp 0 x6
  j done
then:
  sw todo 0 x6
  mv todo x6
done:
...
```

Illustration



It is precisely in these cases that we have to decrement the reference count. If it already was 0, meaning that this was the only reference to the block of memory, we put it onto our lazy free list in register `todo`, as in the above illustration. Otherwise, we decrement the reference count.

Summary

Our strategy for memory management is guided by and optimized for linearity. We know precisely the positions where memory management happens, dictated by explicit structural rules embodied by the **substitute** statements. Explicit substitutions are the only way to drop and duplicate variables, so the code for them appropriately decrements and increments the reference counts of memory blocks associated with the dropped and duplicated variables. All operations take constant time, independent of the number of live or dead blocks.

$\boxed{\mathcal{M}[\![\ \cdot\ ]\!] : \mathit{Definition}_{\mathsf{AxCut}} \to I^*}$

$$\mathcal{M}[\![\ \textsf{def}\ f(\Gamma)\,\{\ s\ \}\ ]\!] \quad = \quad f : \mathcal{M}[\![\ s\ ]\!]$$

$\boxed{\mathcal{M}[\![\ \cdot\ ]\!] : \mathit{Statement}_{\mathsf{AxCut}} \to I^*}$

$$\begin{array}{lcll}
\mathcal{M}[\![\ f(\Gamma)\ ]\!] & = & \texttt{j}\ f & \\
\mathcal{M}[\![\ \textsf{substitute}\ [\Gamma' := \sigma];\ s\ ]\!] & = & \textsc{Share}\,[\Gamma' := \sigma] & \\
 & & \textsc{Erase}\,[\Gamma' := \sigma] & \\
 & & \textsc{Move}\,[\Gamma' := \sigma] & \\
 & & \mathcal{M}[\![\ s\ ]\!] & \\
\mathcal{M}[\![\ \textsf{exit}\ v\ ]\!] & = & \texttt{li x17 93} & \\
 & & \texttt{mv x10}\ (\textsc{Reg}_2\ v) & \\
 & & \texttt{ecall} & \\
\mathcal{M}[\![\ \textsf{lit}\ v \leftarrow n;\ s\ ]\!] & = & \texttt{li}\ (\textsc{Reg}_2\ v)\ n & \\
 & & \mathcal{M}[\![\ s\ ]\!] & \\
\mathcal{M}[\![\ v \leftarrow v_1\ +\ v_2;\ s\ ]\!] & = & \texttt{add}\ (\textsc{Reg}_2\ v)\ (\textsc{Reg}_2\ v_1)\ (\textsc{Reg}_2\ v_2) & \\
 & & \mathcal{M}[\![\ s\ ]\!] & \\
\mathcal{M}[\![\ \textsf{if}\ v \equiv 0\,\{\ s_1\ \}\ \textsf{else}\,\{\ s_2\ \}\ ]\!] & = & \texttt{beq}\ (\textsc{Reg}_2\ v)\ \texttt{x0}\ l & \\
 & & \quad \mathcal{M}[\![\ s_2\ ]\!] & \\
 & & l :\ \mathcal{M}[\![\ s_1\ ]\!] & (l \text{ fresh}) \\
\mathcal{M}[\![\ \textsf{let}\ v\ =\ X(\Gamma_0);\ s\ ]\!] & = & \textsc{Store}\,(\textsc{Reg}_1\ v)\,\Gamma_0 & \\
 & & \texttt{li}\ (\textsc{Reg}_2\ v)\ (\textsc{Index}\ X) & \\
 & & \mathcal{M}[\![\ s\ ]\!] & \\
\mathcal{M}[\![\ \textsf{create}\ v\ =\ \Gamma_0\ b;\ s\ ]\!] & = & \textsc{Store}\,(\textsc{Reg}_1\ v)\,\Gamma_0 & \\
 & & \texttt{la}\ (\textsc{Reg}_2\ v)\ l & \\
 & & \mathcal{M}[\![\ s\ ]\!] & \\
 & & l :\ \textsc{VTable}\ b\,\Gamma_0 & (l \text{ fresh}) \\
\mathcal{M}[\![\ \textsf{switch}\ v\ b\ ]\!] & = & \texttt{jr}\ (\textsc{Reg}_2\ v)\ l & \\
 & & l :\ \textsc{JTable}\ b\,\Gamma & (l \text{ fresh}) \\
\mathcal{M}[\![\ \textsf{invoke}\ v\ X(\Gamma)\ ]\!] & = & \texttt{jr}\ (\textsc{Reg}_2\ v)\ (\textsc{Index}\ X) &
\end{array}$$

Figure 8.1: Translation to RISC-V.

8.4 Translation from AxCut to RISC-V

The translation from AxCut into RISC-V is defined in Figure 8.1. It uses auxiliary definitions from Figure 8.2 and Figure 8.3. Note that while we have used syntax highlighting for code macros such as `RELEASE` in the previous sections, we use small caps (e.g., Release) in the formal definitions. We use functions $\textsc{Reg}_1$ and $\textsc{Reg}_2$ to map variables to the registers they occupy. Similarly, we use functions $\textsc{Offset}_1$ and $\textsc{Offset}_2$ to calculate the offsets for variables into an environment stored in a memory block.

We translate each top-level definition to a label in RISC-V followed by the translation of the corresponding statement, and we translate calls to top-level definitions to direct jumps. We translate explicit substitutions to *erase* and *share* the variables appropriately, and to perform parallel moves before the translation of the remaining statement. The parallel-moves algorithm reassigns the registers according to the substitution and is described by Rideau et al. (2008), formalizing a folklore result that the assignment can be performed in linear time with at most one temporary register. In the definition of Share, we first check for each variable from the old environment how many times it is assigned to variables in the new environment. If this number is greater than 1, the variable has been duplicated and the reference count of the corresponding

Auxiliary Definitions

$\textsc{Share}\,[\Gamma' := \sigma]\,(\Gamma,\, v :^{\chi} \tau)$ =
  if $\textsc{NumRefs}\,v\,[\Gamma' := \sigma] > 1$
    $\textsc{ShareBlock}\,(\textsc{Reg}_1\,v)\,(\textsc{NumRefs}\,v\,[\Gamma' := \sigma] - 1)$
  $\textsc{Share}\,[\Gamma' := \sigma]\,\Gamma$

$\textsc{Erase}\,[\Gamma' := \sigma]\,(\Gamma,\, v :^{\chi} \tau)$ =
  if $\textsc{NumRefs}\,v\,[\Gamma' := \sigma] = 0$
    $\textsc{EraseBlock}\,(\textsc{Reg}_1\,v)$
  $\textsc{Erase}\,[\Gamma' := \sigma]\,\Gamma$

$\textsc{ShareBlock}\,r\,n$ =
```
beq r x0 l                    (l fresh)
  lw temp 0 r
  addi temp temp n
  sw temp 0 r
l :
```

$\textsc{EraseBlock}\,r$ =
```
beq r x0 l1                   (l1, l2 fresh)
  lw temp 0 r
  beq temp x0 l2
    addi temp temp -1
    sw temp 0 r
    j l1
  l2 : sw todo 0 r
    mv todo r
l1 :
```

$\textsc{Reg}_1\,v$	first register of variable v
$\textsc{Reg}_2\,v$	second register of variable v
$\textsc{Index}\,m$	four times index of constructor or destructor X in its signature
$\textsc{Offset}_1\,v$	first memory offset for v in environment
$\textsc{Offset}_2\,v$	second memory offset for v in environment
$\textsc{Move}\,[\Gamma' := \sigma]$	parallel moves for assignments $[\Gamma' := \sigma]$
$\textsc{NumRefs}\,v\,[\Gamma' := \sigma]$	number of variables v is assigned to in $[\Gamma' := \sigma]$
$\textsc{ShareFields}\,r$	$\textsc{ShareBlock}\,f\,1$ for all fields f in block r
$\textsc{EraseFields}\,r$	$\textsc{EraseBlock}\,f$ for all fields f in block r
`temp`	reserved register for temporaries
`todo`	reserved register for lazy free list
`heap`	reserved register for linear free list

Figure 8.2: Auxiliary definitions for the translation.

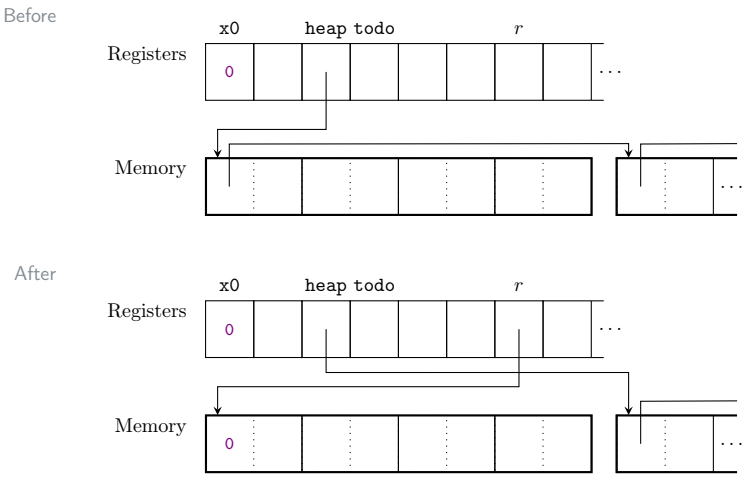
memory block is incremented accordingly by ShareBlock. The variable could, however, also contain an object that has no associated memory block in its first register. Therefore, ShareBlock first checks whether the register indeed points to a memory block. Similarly, in Erase we check whether a variable does not occur in the new environment at all. If this is the case, we erase its memory block with EraseBlock. The latter again first checks whether there actually is a memory block before loading its reference count. It then distinguishes two cases. If there is another reference to this block, the reference count is non-zero and is simply decremented. If the reference count is zero, the block is prepended to the lazy free list pointed to by `todo`.

We translate external statements depending on the target platform. For example, a terminal **exit** statement is translated to an appropriate syscall by first loading the syscall number and the exit code into registers and then executing an `ecall` instruction. Integer literals are loaded with an `li` instruction, and arithmetic operators are translated to a corresponding instruction with the appropriate registers. Conditionals become conditional branching instructions such as `beq`. This requires only one fresh label, since, by construction, the code for the other branch cannot fall through. Either the program ends there, or there will be a direct or indirect jump to another location. In particular, control flow can be joined again by jumping to the same location in both branches.

The remaining statements are those concerned with producers and consumers. The duality of producers and consumers leads to dual translations.

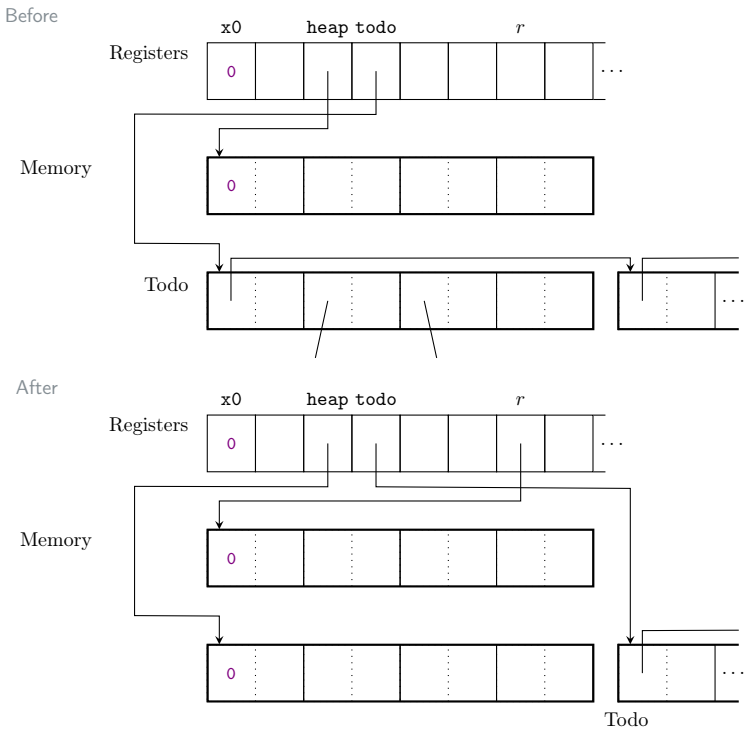
We translate the creation of a constructor or destructor with **let** by first storing the registers corresponding to the environment Γ_0 into a free block of memory. We assume that `heap` always points to a free block that is immediately usable, so we use this block in StoreV (Figure 8.3) to perform the stores. Acquire then moves the address in `heap` into the first register for the constructor or destructor. Since we use equal-size memory blocks to obtain constant-time memory management, it may be necessary to link multiple blocks together if the environment to be stored does not fit into one block. The necessary plumbing for doing so and also for zeroing unused fields in a block is not shown here. The second register for the constructor or destructor is loaded with the index of the tag in the signature of its type, multiplied by `4` to get an offset in code. Finally, we generate the code for the translation of the remaining statement.

When the environment is stored, we have to restore the invariant that `heap` always points to a free block of memory. To do so, Acquire distinguishes three cases. If the linear free list pointed to by `heap` contains a further block, then the first field of the previously free block is non-zero as it contains the address of the next block. The latter is loaded into `heap` and the reference count of the previously free block is initialized to `0`.

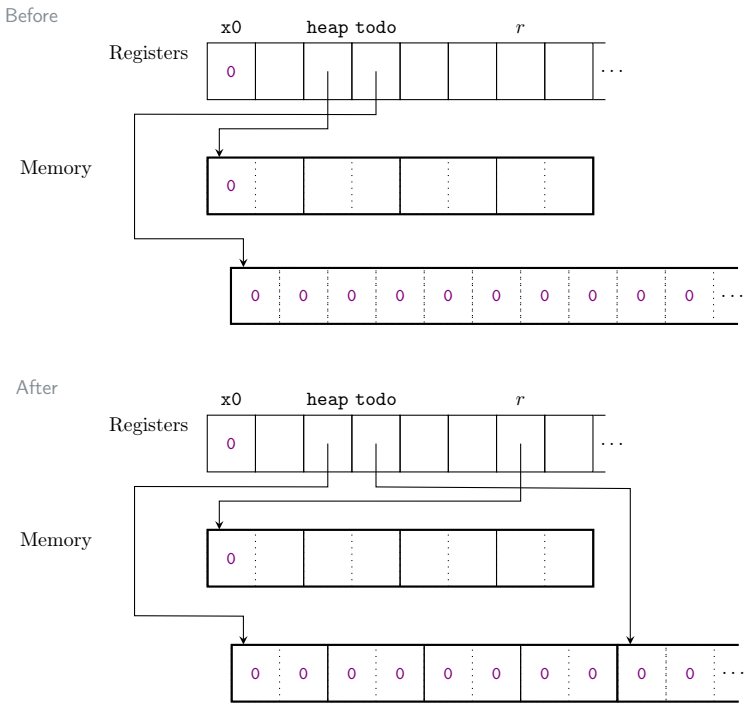


Otherwise, the lazy free list pointed to by `todo` is checked. If it does contain a block, this is the new free block `heap` now points to. Its first field is set to `0`, effectively unlinking it from

the lazy free list, and, since the fields of this block were not erased recursively when it was put on the lazy free list, this is done now.



If the lazy free list does not contain a block either, which is indicated by a 0 at the memory location `todo` points to, we fall back to bump-allocation from the remaining big chunk of memory.



Auxiliary Definitions (continued)

$\textsc{JTable}\,\{\,\overline{X_i(\Gamma_i) \Rightarrow s_i}\,\}\,\Gamma = \overline{\texttt{j}\ l_i}$ ($\overline{l_i}$ fresh)
$\quad \textsc{JTableB}\,\{\,\overline{X_i(\Gamma_i) \Rightarrow s_i}\,\}\,\Gamma\,\overline{l_i}$

$\textsc{JTableB}\,\{\,X_1(\Gamma_1) \Rightarrow s_1,\ b\,\}\,\Gamma\,(l_1,\ \overline{l})$ (x fresh after Γ)
$\quad l_1 : \textsc{Load}\,(\textsc{Reg}_1\ x)\,\Gamma_1$
$\quad\quad \mathcal{M}[\![\, s_1 \,]\!]$
$\quad \textsc{JTableB}\ b\,\Gamma\,\overline{l}$

$\textsc{VTable}\,\{\,\overline{X_i(\Gamma_i) \Rightarrow s_i}\,\}\,\Gamma_0 = \overline{\texttt{j}\ l_i}$ ($\overline{l_i}$ fresh)
$\quad \textsc{VTableB}\,\{\,\overline{X_i(\Gamma_i) \Rightarrow s_i}\,\}\,\Gamma_0\,\overline{l_i}$

$\textsc{VTableB}\,\{\,X_1(\Gamma_1) \Rightarrow s_1,\ b\,\}\,\Gamma_0\,(l_1,\ \overline{l})$ (x fresh after Γ_1)
$\quad l_1 : \textsc{Load}\,(\textsc{Reg}_1\ x)\,\Gamma_0$
$\quad\quad \mathcal{M}[\![\, s_1 \,]\!]$
$\quad \textsc{VTableB}\ b\,\Gamma_0\,\overline{l}$

$\textsc{Load}\,r\,\Gamma = \textsc{Release}\,r$
$\quad \textsc{LoadV}\,r\,\Gamma$

$\textsc{LoadV}\,r\,(\Gamma,\ v :^{\chi} \tau) = \texttt{lw}\ (\textsc{Reg}_2\ v)\ (\textsc{Offset}_2\ v)\ r$
$\quad \texttt{lw}\ (\textsc{Reg}_1\ v)\ (\textsc{Offset}_1\ v)\ r$
$\quad \textsc{LoadV}\,r\,\Gamma$

$\textsc{Store}\,r\,\Gamma = \textsc{StoreV}\,\Gamma$
$\quad \textsc{Acquire}\,r$

$\textsc{StoreV}\,(\Gamma,\ v :^{\chi} \tau) = \texttt{sw}\ (\textsc{Reg}_2\ v)\ (\textsc{Offset}_2\ v)\ \texttt{heap}$
$\quad \texttt{sw}\ (\textsc{Reg}_1\ v)\ (\textsc{Offset}_1\ v)\ \texttt{heap}$
$\quad \textsc{StoreV}\,\Gamma$

$\textsc{Acquire}\,r =$ (l_1, l_2, l_3 fresh)

```
mv r heap
lw heap 0 heap
beq heap x0 l1
    sw x0 0 r
    j l2
l1 : mv heap todo
    lw todo 0 todo
    beq todo x0 l3
        sw x0 0 heap
        EraseFields heap
        j l2
    l3 : addi todo heap 32
l2 :
```

$\textsc{Release}\,r =$ (l_1, l_2 fresh)

```
lw temp 0 r
beq temp x0 l1
    addi temp temp -1
    sw temp 0 r
    ShareFields r
    j l2
l1 : sw heap 0 r
    mv heap r
l2 :
```

Figure 8.3: Auxiliary definitions for the translation (continued).

We translate the usage of a constructor or destructor bound to variable v (**switch**) to a jump instruction that adds the second register of v to the address of a jump table l and then jumps to the resulting address. Since that value is exactly the index in the signature multiplied by 4, the jump ends up at the entry of the jump table for that tag. The jump table is labeled with a fresh label and consists of one entry for each branch, and each entry is a jump to a fresh label for the corresponding branch. We translate each branch first to the corresponding label, then to loading the environment Γ_i, and finally the translation of the branch statement, as can be seen in the definition of JTableB (Figure 8.3). When loading the environment, the corresponding memory block is released. To do so, Release first checks whether its reference count is zero. If this is the case, the memory block is prepended to the linear free list pointed to by `heap`. Otherwise, there is a reference from somewhere else. Then, the reference count of the block is decremented and its fields are shared since there is now an additional reference to them in the registers they are loaded into. Sharing the fields requires loading them into registers, so to avoid loading the fields twice, we could also share each field only after it has been loaded anyway, and we do so in our implementation (making Load and Store a little less dual than in Figure 8.3).

Dually to the creation of a constructor or destructor, we translate the creation of a closure with **create** to a virtual table l, which is placed right after the code for the remaining statement. Before the code of the remaining statement, we store the closure environment Γ_0 to memory, which moves the address of the corresponding memory block into the first register of the closure. We then load the address of the virtual table into the second register. For the virtual table, we translate each branch to first load the closure environment Γ_0 and then execute the translated statement.

The usage of a closure v with **invoke** is translated to an indirect jump to the address in the second register of v, adding the index of the constructor or destructor. Restoring the closure environment is taken care of at the destination. The arguments of the constructor or destructor are already in the corresponding registers, as ensured by our typing rules.

8.4.1 Section Summary

AxCut serves as a bridge between classical sequent calculus and machine code. In this section, we have presented a direct translation from AxCut to RISC-V. AxCut supports, in addition to algebraic data types, also control effects and codata types. Yet, the generated code does not require any runtime system. In the next section, we describe our implementation and present benchmarks.

9 Evaluation

In the previous chapters, we have described the full compilation pipeline from the surface language Fun, via the sequent-calculus-based intermediate representations Core and AxCut, down to machine code. In this section, we briefly discuss our Rust implementation of the compilation pipeline (Section 9.1), and report benchmarking results (Section 9.2). The Rust implementation of the compiler described in this thesis is available at Müller et al. (2025a). The benchmarks are available at Müller et al. (2025b). We conclude with the discussion of related work in Section 9.3.

9.1 Implementation

We have implemented the full compilation pipeline in Rust[8]. We parse a textual representation of the surface language Fun and perform simple typechecking during which we instantiate all type parameters in type declarations with monomorphic types, as described in Section 6.1. The implementation of the compilation pipeline mostly follows the descriptions in the previous sections. We first translate the typechecked programs into the intermediate language Core, then perform the focusing and shrinking transformation, before translating into AxCut. The main difference from the presentation in Chapter 7 is that we translate into the unified syntax of AxCut before performing the linearization pass on that non-linearized version. Moreover, the implementation uses a fully one-sided version of AxCut. Both of these differences are minor.

Our implementation not only generates code for RISC-V, but also supports compilation to x86-64 and AArch64. Both of these work in a very similar way. In fact, most of the logic of code generation is performed in an abstraction layer, and the different backends mainly implement the details in which they differ. The implementation currently neither performs register allocation nor does it perform any optimization passes. Moreover, register and memory layout could be improved significantly.

RISC-V

The implementation of the translation to RISC-V closely follows the presentation in Chapter 8. The main difference to the code presented there is that we use the slightly more performant version for loading and sharing fields of memory blocks we have described. For the RISC-V backend we have currently not implemented any platform-dependent plumbing to turn the generated assembly code into a complete routine that can be executed.

x86-64

The code we generate for x86-64 is again very similar to the translation in Chapter 8. However, since we want to create an executable to run on a Linux system, we emit code for the Yasm Assembler[9], a more performant rewrite of the Netwide Assembler (NASM)[10]. The emitted code also contains the plumbing which turns it into a callable assembler routine. We then use the object code generated for this assembler routine by Yasm and link it with a tiny driver program written in C that gathers command-line options, allocates memory that can be used by the

[8] https://www.rust-lang.org
[9] https://github.com/yasm/yasm
[10] https://nasm.us

routine, and calls the routine with these arguments. We also link it with a tiny runtime that currently only implements simple C functions for printing integers, but can be extended in the future. In general, the instructions for x86-64 are different from those for RISC-V, of course. However, we only use a small subset of basic instructions (like, `mov`, `add`, `jmp`, `cmp`, etc.), so that the differences for the translation are minor. Still, one practically important difference is that in x86-64 there are only 16 registers compared to 32 registers in RISC-V. To allow for more than six variables to be live at the same time, we spill variables onto the stack, which we do not use for any other purpose.

AArch64

We can moreover generate code for AArch64, again in a very similar way. We emit code that can be processed by a standard assembler such as the GNU assembler[11]. Using the same C driver and runtime as for x86-64, the resulting executables can be run on a Linux system and have also been tested on an Apple M1. Even though there are 32 registers on AArch64, we have also implemented spilling for variables in this backend, since we have found that in some larger programs it can still be too restrictive to allow for at most 14 variables to be live at the same time.

9.2 Benchmarks

To evaluate the performance of the code produced by our compilation pipeline, we have implemented a benchmark suite consisting of a considerable number of programs of varying size. We compare the programs written in our surface language Fun with a selection of industrial and research languages. All benchmarks were conducted on a 9th Gen Intel(R) Core(TM) i5-9500 running Arch Linux. The running times were measured using the command-line benchmarking tool `hyperfine`[12], which we configured to perform 3 warmup runs before taking the measurements. Since many of our benchmark programs are highly recursive and we run them on large inputs, we have increased the default stack size to avoid overflowing the C call stack. While our approach does not make use of the call stack, we had to allocate a sufficiently large amount of heap space at the beginning of the programs in our compiler, as our implementation currently does not dynamically demand more memory from the operating system.

Systems

We compare the running time of a range of different systems with the implementation of our compilation pipeline: SML/NJ, MLton, OCaml, Koka, Effekt, and Rust. These systems have been chosen for their different implementation strategies of data and functions, usage of the runtime stack, optimizations, and memory management. For each of them, we manually translated programs into the respective language.

SML/NJ[13] (110.99.8) is a compiler for Standard ML (Milner et al., 1997) that uses an intermediate representation based on continuation-passing style (Appel, 1992) and directly generates machine code from it after applying several transformations and optimizations. Being based on CPS, it does not generally make use of the runtime stack. It uses copying generational garbage collection.

MLton[14] (20210117) is an optimizing whole-program compiler for Standard ML. It uses a sequence of intermediate representations and directly generates machine code from the last one. It uses generational garbage collection with a mix of copying and mark-compact.

[11] https://www.gnu.org/software/binutils/
[12] https://github.com/sharkdp/hyperfine
[13] https://www.smlnj.org
[14] http://mlton.org

OCaml[15] (5.3.0) directly generates machine code (with `ocamlopt`) from its intermediate representations, but does not perform too many optimizations. It uses generational garbage collection, with a mix of copying and mark-and-sweep.

Koka[16] (Leijen, 2014) (3.2.2) generates C code (Xie and Leijen, 2021) and invokes a C compiler to generate machine code. It uses garbage-free reference counting (Reinking et al., 2021) and optimizes programs to use in-place mutation and tail-recursion modulo cons (Lorenzen et al., 2023). We have invoked it with full optimizations (i.e., `-O2`).

Effekt[17] (Brachthäuser et al., 2020) (0.45.0) generates LLVM[18] code (Muhcu et al., 2025) and invokes the LLVM toolchain to optimize programs and generate machine code. Similar to Koka, it also uses garbage-free reference counting, and LLVM performs significant optimizations.

Rust (1.89.0) also generates LLVM code and invokes the LLVM toolchain to optimize programs and generate machine code. It uses a mix of linearity and borrowing (Weiss et al., 2021) for garbage collection. It also features opt-in reference-counted references, which we use in some benchmarks to get shareable data structures for better comparison. Similar to Koka, we have invoked it with full optimizations (i.e., `--opt-level=3`).

All of these compilers perform optimizations to varying degrees, while our compilation pipeline for Fun does not perform any of them. Still, as some of the compilers might rely on optimizations to ensure good code quality, we have decided to run them with optimizations enabled. The goal of these benchmarks is thus not to show raw performance, but to demonstrate that the implementation of our approach actually works and is comparable to existing compilers when it comes to performance.

Programs

Table 9.1 lists the benchmark programs for which we have measured the running times, along with short descriptions and the arguments used to run them. For all programs but `Fish`, the first argument is the number of iterations, i.e., how often the main part of the program was executed in one run. Most of the benchmark programs are taken from the Manticore benchmark suite[19] (marked with $*$) and a few are taken and adapted from Haskell's NoFib suite (Partain, 1993; Chen and Parreaux, 2024) (marked with †). We have chosen those programs from these suites that can be implemented with the features we currently support in Fun, leaving out programs that need features like floating point numbers, strings, and more general IO. The limited amount of features is not fundamental, however, and we plan to add support for more features in the future. The benchmark programs are complemented by a few micro benchmarks we wrote ourselves (Schuster et al., 2025b).

The programs vary in size, from just a few lines to several hundred lines of code. They allow us to measure several different aspects of (functional) programming languages, such as pushing and popping stack frames via recursion, first-class functions and indirect calls, allocation and deallocation of data types, linear and non-linear usage of data types, returns to pattern matching contexts, and also loops and operations on integer values. We believe that together these programs provide a good spectrum for assessing the performance of various language features. None of these programs uses advanced control flow features like exceptions or control operators, which we measure and report separately in Section 9.2.

Results

Figure 9.2 and Figure 9.3 show the benchmarking results of the various programs in the different systems. The figures show the speedup factors of the other systems relative to the implemen-

[15] https://ocaml.org
[16] https://koka-lang.github.io/koka/doc/index.html
[17] https://effekt-lang.org
[18] https://llvm.org
[19] https://github.com/ManticoreProject/benchmark

Program	Arguments	Description
`Ack`*	4, 3, 10	The Ackerman function
`Boyer`†	2, 9	The Boyer constraint solver
`Constraints`†	1, 6	Tolmach and Nordin's constraint solver
`Cryptarithm1`†	1, 1	Cryptarithm solver
`Deriv`*	1000000, 5, 7	Symbolic differentiation
`Divrec`*	1000, 20000	Recursively calculates division by 2
`EraseUnused`	1, 10000	Calculates N by needlessly allocating linked lists
`Evenodd`*	10, 20000000	Recursively calculates if the input is even or odd
`FactorialAccumulator`	1, 10000000	Calculates the factorial with a magic constant
`Fib`*	1, 39	Calculates Fibonacci numbers
`TailFib`*	10000000, 44	Calculates Fibonacci numbers with tail recursion
`Fish`†	150	Generates lines for a fractal image
`Gcd`†	10, 400	Calculates the greatest common divisor
`Integer`†	1, 4500001	Integer operations $(*, +, \ldots)$ on ranges
`IterateIncrement`	1, 100000000	Computes N using higher-order functions
`Lcss`†	400	Hirschberg's LCSS algorithm
`Life`*	1, 800	Simulates the Game of Life
`LookupTree`	1, 10000000	Calculates N by creating a binary tree
`Motzkin`*	1, 20	Calculates Motzkin numbers
`MatchOptions`	1, 10000000	Computes N using a chain of pattern matches
`Merge`*	500, 50000	Creates two lists and merges them
`Minimax`*	1	Solves Tic-Tac-Toe with the Minimax algorithm
`Nqueens`*	1, 11	Solves the Nqueens problem
`Perm`*	1, 3, 9	Calculates permutations of lists
`Primes`*	10, 20000	Implements the Sieve of Eratosthenes
`Sudan`*	1000000, 2, 2, 2	Calculates the Sudan function
`SumRange`	1, 10000000	Sums a range of numbers
`Tak`*	1000, 21, 20, 11	Calculates the Takeuchi function
`Takl`*	100, 21, 20, 11	Tak using lists to encode natural numbers
`Cpstak`*	100, 21, 20, 11	CPS-transformed Tak

Table 9.1: Used arguments and descriptions of benchmark programs. Those marked with $*$ are from Manticore, those marked with † are from NoFib, those without a mark are our own micro benchmarks.

tation of our compiler, which serves as the baseline. Thus, values greater than 1 mean shorter execution times, i.e., the other system performs better. We have chosen a logarithmic scale, since for some programs the speedups (or slowdowns) are rather large. In particular, this is the case for `IterateIncrement` where Effekt, Rust, and Koka optimize the program to be constantly under the measuring threshold, independent of the used argument. The same is true for the Rust version of `MatchOptions`.

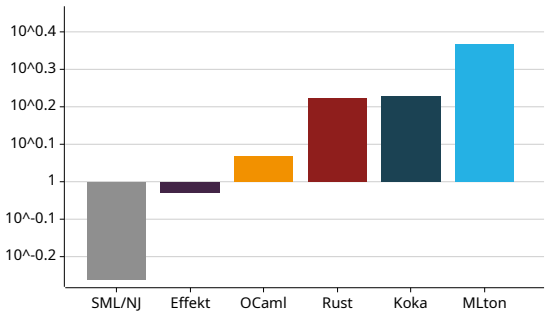


Figure 9.1: Geometric means of relative performance results on a logarithmic scale.

In general, we can see that even though the implementation of code generation for our compilation pipeline itself is fairly simple and we do not perform any optimizations, running times are comparable with state-of-the-art compilers. However, we can also see that compile-time optimizations make a significant difference. This is also visible when looking at the geometric means of the relative speedups for the different languages over all benchmark programs shown in Figure 9.1.

MLton, which is specifically designed to optimize functional language features aggressively, exhibits the highest mean speedup (roughly factor 2.32) and is also faster than our compiler for almost all programs, though to a somewhat varying degree.

Koka, which also performs several optimizations specifically for functional programs but in addition relies on a state-of-the-art C compiler for lower-level optimizations, exhibits a significant mean speedup (roughly factor 1.69) as well. Its speedups across the individual programs vary a bit more though, with some programs also being slower than the code produced by our compiler.

The same is true for *Rust* (roughly factor 1.66), which, however, puts significantly less focus on optimizing functional features like first-class functions or linked lists used by many programs, and we do not go to great lengths to avoid cloning, so a smaller speedup is to be expected. We note though, that for `Deriv` we used vectors from Rust's standard library instead of linked lists, since otherwise the performance is unreasonably slow. It should also be noted that Rust drops the whole memory allocated before the program ends, while the memory management of the other languages in principle allows leaving the cleanup of allocated memory to the OS after the end of the program. This can make a significant difference in some programs, such as `EraseUnused`.

OCaml optimizes much less aggressively, and consequently exhibits a significantly smaller mean speedup (roughly factor 1.17). This is, however, exacerbated a bit by three negative outliers in our micro benchmarks (`LookupTree`, `MatchOptions`, `SumRange`), in which OCaml performs considerably worse. These outliers are likely due to the garbage collector not being properly tuned for such use cases.

They are even worse for *SML/NJ*, which in the mean generates considerably slower code compared to our compiler (roughly factor 0.55), but across the programs has a more varied performance. Its compilation approach using a CPS-based intermediate representation is the

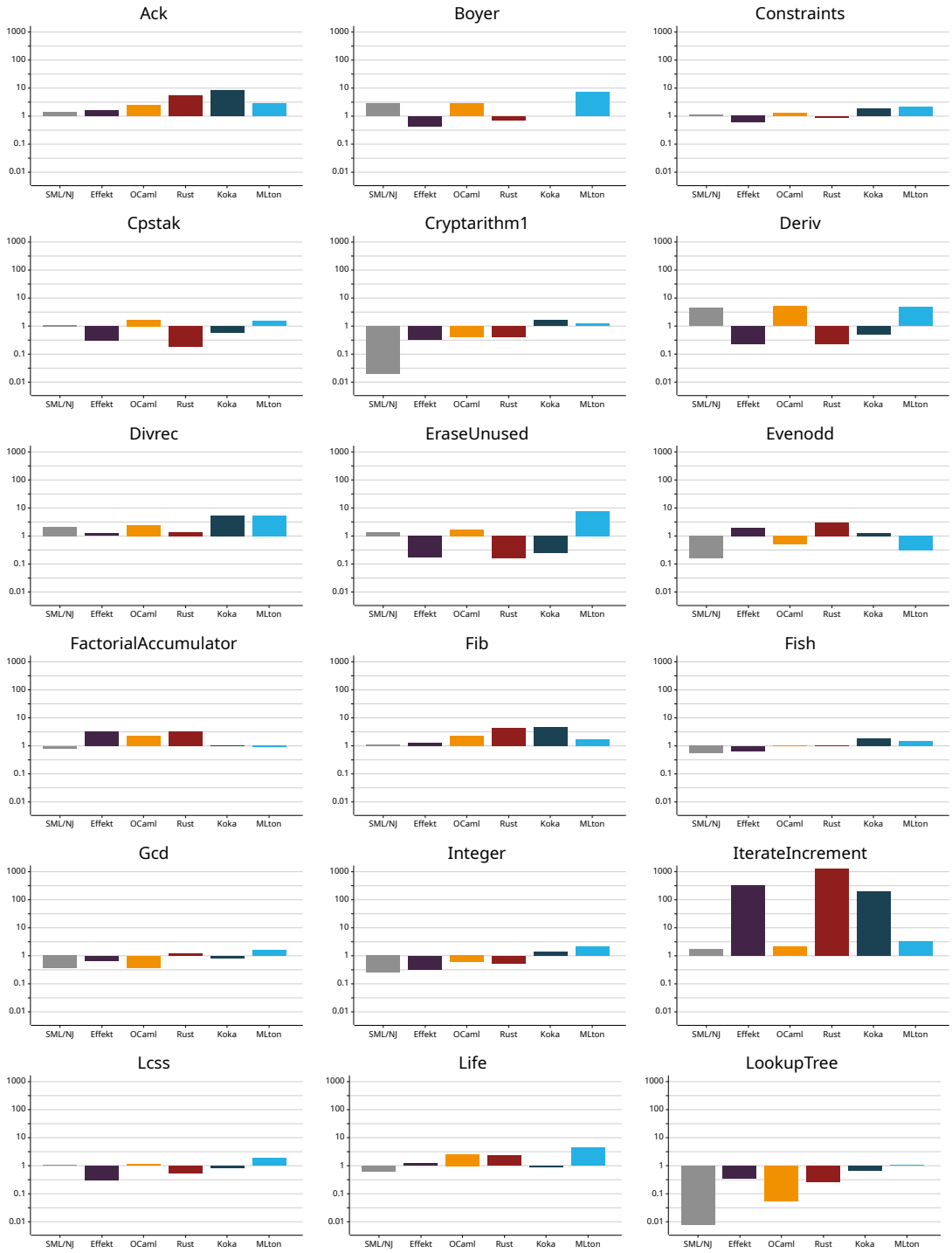


Figure 9.2: Relative performance results on a logarithmic scale (A-L).

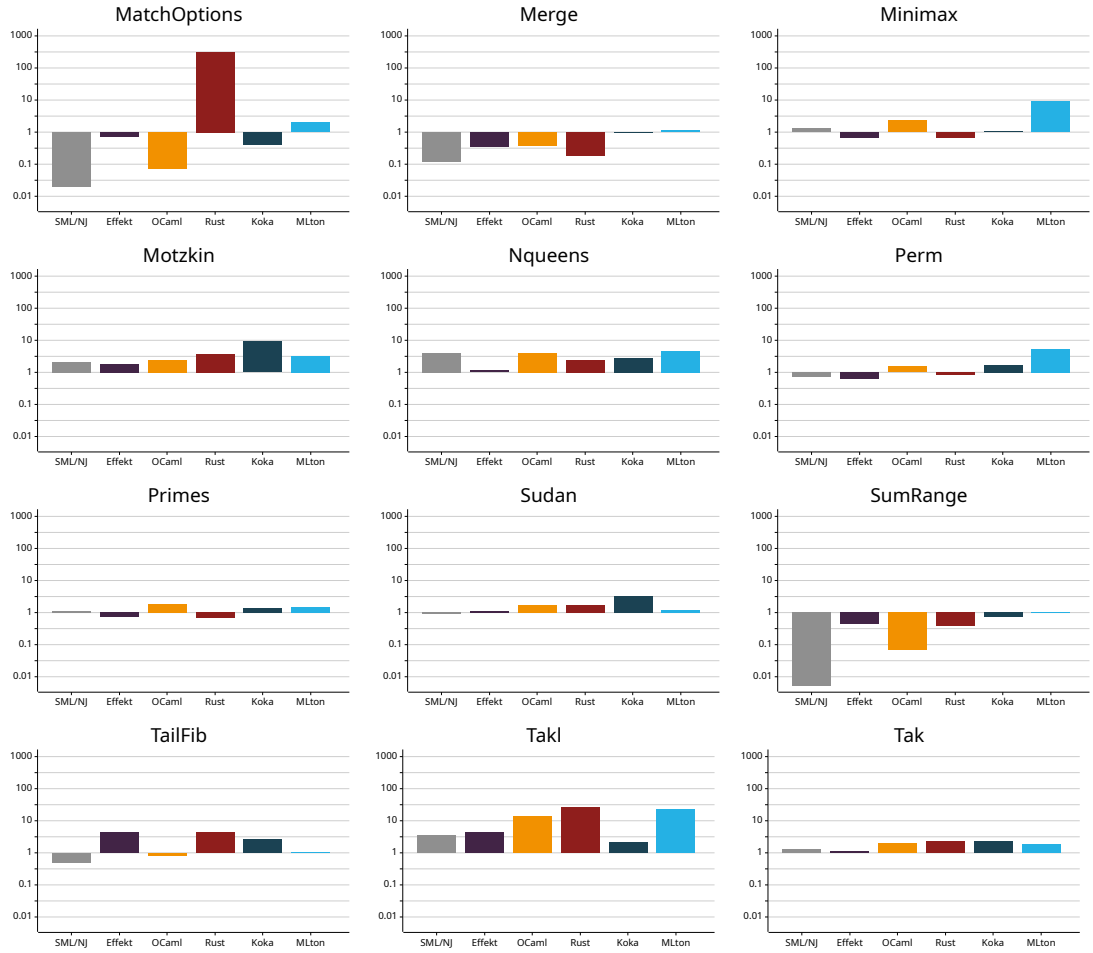


Figure 9.3: Relative performance results on a logarithmic scale (M-Z).

one that is probably closest to our approach among the compilers we compare against, so we deem the results to be particularly interesting. This also shows that while the mean gives a rough indication of performance, it should be interpreted with some care.

The mean performance of *Effekt* is quite close to that of our compiler (roughly factor 0.93). To support tail calls and control effects, like our compiler, Effekt does not make significant use of the C call stack. However, in contrast to our implementation, Effekt supports a more general form of delimited continuations, as do Koka and OCaml. Effekt's calling convention and memory management currently seem to come with an overhead in some benchmarks.

The programs in all languages are written in such a way that all implementations are as close to each other as possible. For some languages, in particular Rust and Effekt, more idiomatic versions would likely be considerably more performant. That said, we expect that the results for our compiler will significantly improve once we implement optimizations, increasing the competitive ability of our approach further. We leave exploring optimizations to future work.

Programs with Control Operators

To assess the performance of control operators, we have further implemented variations for a few of the above benchmark programs, also taken from the Manticore suite. All of them are recursive, often highly so. They only differ from the original versions in that they unnecessarily

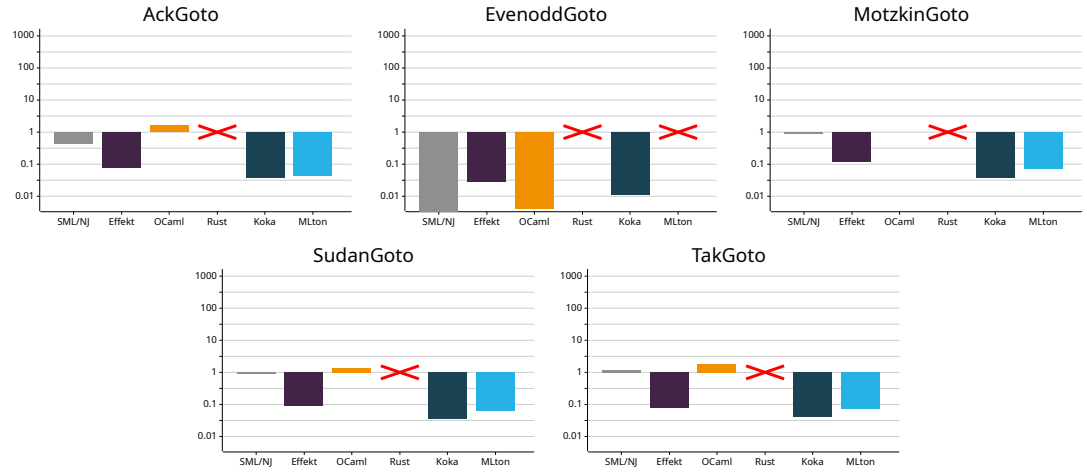


Figure 9.4: Relative performance results on a logarithmic scale for `Goto`-benchmarks.

use control operators in a few places with the goal of measuring the overhead they cause. We measure them separately and do not factor them into the geometric means, because the languages we benchmark against wildly vary in what control operators they support, making them less comparable.

In our surface language Fun, we use the **label** and **goto** constructs, which is why we have added the suffix `Goto` to the names of the original versions. In the `Goto`-versions, we insert a **label** around the initial call of the function computing the result and then pass the captured consumer to the call. The functions thus all receive an additional consumer argument, which they invoke instead of returning normally. Except for recursive tail calls, we insert another **label** around each function call, capturing the current consumer and passing it to the call, that is, all functions still return to the same context as in the original version and there is no short-cutting. The only exception to this scheme is `EvenoddGoto`, where only one of the two mutually recursive functions receives an additional consumer argument, while the other keeps returning normally.

In the other languages, we have used different control operators. In SML, i.e., for SML/NJ and MLton, we have used `callcc` and `throw`, since they have essentially the same semantics as **label** and **goto** and hence are mostly comparable. In OCaml we have used exception handlers. In place of **label**, we insert an exception handler handling raised exceptions in a trivial way, and instead of invoking a consumer, we raise an exception to model returns. The functions thus of course do not receive an additional argument. In Koka and Effekt we use effect handlers to model exception handlers. Rust has no direct support for control operators. One possibility would be to write the program in monadic style with the `Result` type, which results in programs with roughly the same performance as the original ones. This is quite different from using control operators, however, so we have chosen not to include them in the comparison. Another option would be to raise unwinding panics and catch them. This mechanism is not intended to be used as a general exception mechanism, however, and it is quite slow, so we have not included these versions either.

Figure 9.4 shows the results for the `Goto`-benchmarks. We can see that the relative performance with respect to our compiler has degraded for all other systems when comparing with the relative performance for the original versions, but to varying degrees. For SML/NJ and OCaml, the relative performance has not degraded too much, with the exception of the `EvenoddGoto`-benchmark, where it has degraded dramatically. OCaml's exception handlers are rather fast and SML/NJ's approach using continuation-passing style is well-suited for implementing `callcc`. The reason why `EvenoddGoto` is such an outlier is likely because the nesting

depth of control operators in it is particularly high (several millions) compared to the other programs (several thousands for `AckGoto` and a few tens for the others). MLton, in contrast, is much less well-tuned towards using `callcc`, and its relative performance has dropped rather strongly. For `EvenoddGoto`, MLton's performance was even too slow to measure it. The relative performance of Koka and Effekt has also degraded significantly, with Koka having the strongest decline. This shows that the considerably more powerful effect handlers are more difficult to implement efficiently, and potentially also block further optimizations. Notably, the relative performance of Effekt and Koka for `EvenoddGoto` also dropped more than for the other programs, but not as dramatically more as for OCaml and SML/NJ. This indicates a good performance scalability of effect handlers in Koka and Effekt. In general, while the comparison here is not quite fair because the control operators in the languages differ in their expressiveness and the examples are somewhat artificial, the partly orders-of-magnitude advantage of our compiler over the other approaches is an encouraging indication that our technique is particularly well-suited for implementing control operators.

9.3 Related Work

We discuss term assignments for classical sequent calculus, which our intermediate representations are explicitly based on. After a brief discussion of codata types and control operators, we compare our systems with other kinds of intermediate representations that are most closely related.

9.3.1 Term Assignments for Sequent Calculus

Curien and Herbelin (2000) present the $\lambda\mu\tilde{\mu}$-calculus as a term assignment for Gentzen (1935a,b)'s classical sequent calculus LK. They introduce the three syntactic categories of terms, contexts, and commands (which we call statements) and lay the foundation for other term assignments for classical sequent calculus. They only consider the logical connective of implication and globally fix the evaluation order to be either call-by-value or call-by-name.

Wadler (2003) presents a similar term assignment for LK, the dual calculus, where contexts are called coterms and commands are called statements. The dual calculus is based on conjunction, disjunction, and negation instead of implication. It also fixes a global evaluation order, but is designed to make the duality between call-by-value and call-by-name obvious. This duality had already been observed before by Filinski (1989).

Zeilberger (2008) explains a symmetric logical system of proofs and refutations, introduces a term assignment for it, and relates the logical concepts of polarity and focusing to the evaluation order within programs. He uses daimons (Girard, 2001) for external effects and explicit substitutions for structural rules. His system does not a priori allow for variables that stand for pairs, but he shows that this is admissible. His system features logical connectives, including logical shifts that mediate between call-by-value and call-by-name, but does not allow for user-defined data and codata types. The approach of relating the polarity of types to evaluation orders and using shift connectives to mediate between them is discussed in more detail by Zeilberger (2009) and Munch-Maccagnoni (2009, 2013).

Downen (2017) has devoted an entire thesis to bringing the duality inherent in sequent calculi into the world of programming languages by providing various term assignments for symmetric logical systems. Our work is based on many of his insights. Many of these have been published independently, for example, symmetric user-defined data and codata types (Downen and Ariola, 2014), a tutorial explaining a computational interpretation of classical logic via focusing (Downen and Ariola, 2018), and how to mix call-by-value and call-by-name for arbitrary data and codata types within the same program (Downen and Ariola, 2020).

Ostermann et al. (2022) present a fully symmetric calculus, which has not only left and right introduction rules like sequent calculi, but also left and right elimination rules. They list

these for a large number of positive and negative logical connectives and their duals. They explain how elimination forms can be recovered from opposite introduction rules using their core language constructs of axiom, cut, and activations, i.e., μ- and $\tilde{\mu}$-abstractions.

9.3.2 Codata Types

Codata types were originally invented by Hagino (1989). They had the most success in proof assistants such as Agda, where they help circumvent certain technical problems that arise when trying to model coinductive types. Copattern matching as a way to create producers of codata types was popularized by Abel et al. (2013), although the basic idea of the concept had been around before that, see, e.g., Zeilberger (2008). But probably the best starting point to learn more about codata types is an article written by Downen et al. (2019).

9.3.3 Control Operators and Classical Logic

The **label**/**goto** construct that we are using in Fun is an example of control operators for undelimited continuations, of which Landin's operator J (Thielecke, 1998; Felleisen, 1987; Landin, 1965a) likely is the oldest. Their translation into Core uses μ-abstractions, which are also a form of control operator that was originally introduced by Parigot (1992) before it became a part of the $\lambda\mu\tilde{\mu}$-calculus of Curien and Herbelin (2000). Control operators have an important relationship to classical logic via the Curry-Howard isomorphism. This relationship was discovered by Griffin (1989); a more thorough introduction can be found in Sørensen and Urzyczyn (2006).

9.3.4 Intermediate Representations

Compiling programming languages via continuation-passing style is a long and ongoing tradition (Steele, 1978; Appel, 1992; Kennedy, 2007). In continuation-passing style, functions explicitly call their continuation instead of returning to their implicitly available calling context. Our approach is more general in that our intermediate languages allow for multiple arbitrarily structured explicit contexts. This subsumes, for example, double-barreled continuation-passing style for implementing exceptions. We can mix different such calling and returning conventions within the same program. Finally, our contexts are fully first-class, following classical sequent calculus.

Appel (1992) describes the full compilation pipeline of the SML/NJ compiler, which uses CPS as intermediate representation. While SML does support control operators for undelimited continuations, just as we do, it does not exhibit general codata types nor different evaluation strategies. The structure of Appel (1992)'s pipeline is somewhat similar to ours. After translating from the typechecked surface language into CPS, it includes transformations for closure conversion, which have later been improved upon (Shao and Appel, 2000), and for register spilling. This ensures that every variable can directly correspond to a register. The transformed representation is then compiled to an abstract machine that is fairly close to machine code and from which several different platforms can be targeted, although there also is an alternative LLVM backend by now (Farvardin and Reppy, 2021). Our treatment of closures is quite a bit simpler since we use flat closures with two registers to represent them and we do not support recursive closures directly, although we can easily express the latter using top-level functions. We have no transformation for register spilling, since we simply spill variables that do not fit into registers to the stack, which we do not use for other purposes (except calling runtime functions) anyway. The runtime of Appel (1992)'s system includes a tracing garbage collector, while we use a lazy constant-time reference counting scheme, facilitated by our use of explicit substitutions. Moreover, (Appel, 1992) performs register allocation and extensive optimizations, which we also plan to do in the future.

An alternative to continuation-passing style is A-normal form (Sabry and Felleisen, 1992; Flanagan et al., 1993). The result of every non-trivial computation is bound to a variable, which facilitates code generation. For the same reason, we bind every non-variable expression and context to a (co)variable. While continuation-passing style and A-normal form syntactically fix the evaluation order, this is not the case for our focused language, where the evaluation order is determined by how critical pairs are resolved. We only do so during our shrinking pass, based on the polarity of types. Thus, after shrinking, and in particular in the AxCut language, the evaluation order is syntactically explicit.

Leißa et al. (2015) present an intermediate representation where, unlike in lambda-lifted representations, there are no scopes and top-level definitions do not receive parameters. Rather, the nesting of scopes follows from the use of variables. With our explicit substitutions in AxCut, we can similarly view our top-level functions as not taking parameters, but instead being annotated with the free variables they assume to be present.

Maurer et al. (2017) and Cong et al. (2019) each present an intermediate representation with some form of control operator that reifies the current context and gives a name to it. In both of these works, the use of this name is restricted and it is not first-class. In contrast, we can give names to arbitrary contexts and use them in unrestricted ways. Moreover, our notion of context is more general and goes beyond returning to it with an argument.

Downen et al. (2016) present an intermediate representation that is explicitly based on sequent calculus. However, being in the context of the GHC Haskell compiler forced some design decisions in order to maintain a direct correspondence to the existing intermediate representation of Haskell. In particular, consumers are restricted in order to keep the system pure. The authors write:

> However, it still remains to be seen how an unrestrained, and thus more cohesive and simpler, classic sequent calculus would fare as the intermediate language of a compiler.

Our work can be seen as the first part of answering this question.

10 Conclusion

In this thesis, we have presented two different ways how explicit consumers can be used in the compilation of functional programming languages. Explicit consumers enable complex non-local control flow without the need for runtime support. They make the control flow of programs explicit, and thus enable optimizations by simple inlining and reduction that are difficult to achieve otherwise, in particular in the presence of non-trivial control flow.

Back to Direct Style

Continuation-passing style is a well-studied example of a system with explicit consumers, namely continuations. Its use as a compiler intermediate language has been long established. However, CPS also comes with downsides. In particular, it does not make use of the call stack provided by many platforms and instead allocates stack frames on the heap, which can lead to worse performance. A translation back to direct style opens up the possibility for a compiler to get the best of both worlds by translating programs into CPS, transforming them there, and then translating back to DS. We have taken an important step in this direction with the design of a DS translation satisfying many properties that are important to make it usable in a compiler. In particular, it never inserts unnecessary control operators or duplicates code, and it works on the full language. Still, it is possible to make different design choices for the insertion of control operators. We have designed our translation to fit particularly nicely with our abstract machine. From a compiler perspective, it would be useful to also consider reduction theories for our calculi, in particular, to be able to reason better about optimizations in the CPS language.

There are several interesting directions for future work. While our basic calculi are quite minimal, we have also discussed potential solutions for extensions with features like conditionals and data and codata types, which should be investigated further. Another important future direction is to extend our approach to languages featuring more powerful delimited continuations, facilitating, e.g., the application of the translation for the compilation of effect handlers. A promising idea is to adapt our calculi and translations in such a way that they can be iterated, thus supporting the whole CPS hierarchy. It would also be interesting to consider partial translations back to direct style that leave the parts of the program exhibiting non-trivial control flow in CPS. This would avoid the need for the support of control operators, while still having some of the benefits of executing programs in direct style. Yet another interesting consideration is the interaction of continuations with local mutable state. A CPS language might use some sort of state passing and a translation back to direct style might then try to restore the use of native local mutable state where possible.

Compiling with the Sequent Calculus

CPS is not the only possibility for an intermediate language with explicit consumers. Languages based on classical sequent calculus take the idea of explicit consumers even further by making them symmetric to producers, thus exposing the structure of consumers in the same way as for producers. This makes such languages a promising alternative to CPS for intermediate representations. We have presented a full compilation pipeline with intermediate representations based on classical sequent calculus. We have started with a functional surface language Fun, exhibiting general data and codata types, as well as control operators for undelimited continuations, and ended up with machine code for several different platforms. The intermediate representation Core, used as target for Fun, is directly based on fully symmetric classical

sequent calculus. To facilitate code generation, we have identified AxCut as a fragment of this term assignment for sequent calculus proofs together with a normalization into this fragment. It serves as a bridge between the logical system and bare-metal machine code. The symmetric nature of classical sequent calculus naturally facilitates expressing control effects without the need for a runtime system, and enables a uniform representation of data and codata types. The high-level structure of the logical system is reflected as invariants imposed on the generated machine code and its use of registers and memory.

Our benchmarking results are promising, showing that our approach is comparable to sophisticated compilers for other languages that produce optimized code, even though our implementation is, as of yet, rather simple and does not perform any optimizations. Similar to compilers using CPS, we do not make use of the call stack. This is to some degree ameliorated by having a separate linear free list in our memory management system. It will be interesting to further investigate the impact on performance once we have implemented optimizations. A potential avenue for improvements in this regard could be to selectively apply the techniques of our translation back to direct style for parts of programs that exhibit pure stack-like usage. Furthermore, we believe that the uniform and principled representation of programs, expressions, and contexts in our sequent-calculus-based intermediate representations not only admits many standard optimizations but also opens up many new optimizations across various language features. It will be particularly interesting to investigate the compilation of high-level features for structuring interactive programs, such as async/await, coroutines, or effect handlers, via this compilation pipeline in the future. Finally, we hope that the simplicity of code generation from AxCut makes an equally simple mechanized proof of its correctness possible.

Bibliography

Martin Abadi, Luca Cardelli, P-L Curien, and J-J Lévy. 1991. Explicit substitutions. *Journal of Functional Programming* 1, 4 (1991), 375–416. doi:10.1017/S0956796800000186

Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. 2013. Copatterns: Programming Infinite Structures by Observations. In *Proceedings of the Symposium on Principles of Programming Languages*. ACM, New York, NY, USA, 27–38. doi:10.1145/2429069.2429075

Jean-Marc Andreoli. 1992. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation* 2, 3 (1992), 297–347. doi:10.1093/logcom/2.3.297

Andrew W. Appel. 1992. *Compiling with Continuations*. Cambridge University Press, New York, NY, USA. doi:10.1017/CBO9780511609619

Olivier Savary Belanger, Matthew Z. Weaver, and Andrew W. Appel. 2019. *Certified Code Generation from CPS to C*. Technical Report. Department of Computer Science, Princeton University. https://www.cs.princeton.edu/~appel/papers/CPStoC.pdf

Malgorzata Biernacka, Dariusz Biernacki, and Sergueï Lenglet. 2011. Typing Control Operators in the CPS Hierarchy. In *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming (PPDP '11)*. Association for Computing Machinery, New York, NY, USA, 149–160. doi:10.1145/2003476.2003498

Małgorzata Biernacka and Olivier Danvy. 2007. A concrete framework for environment machines. *ACM Trans. Comput. Logic* 9, 1 (Dec. 2007), 6–es. doi:10.1145/1297658.1297664

Dariusz Biernacki, Mateusz Pyzik, and Filip Sieczkowski. 2020. A Reflection on Continuation-Composing Style. In *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 167)*, Zena M. Ariola (Ed.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 18:1–18:17. doi:10.4230/LIPIcs.FSCD.2020.18

Dariusz Biernacki, Mateusz Pyzik, and Filip Sieczkowski. 2021. Reflecting Stacked Continuations in a Fine-Grained Direct-Style Reduction Theory. In *23rd International Symposium on Principles and Practice of Declarative Programming (PPDP 2021)*. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3479394.3479399

David Binder and Thomas Piecha. 2022. Administrative Normal Forms and Focusing for Lambda Calculi. In *Logically Speaking. A Festschrift for Marie Duží*, Pavel Materna and Bjørn Jespersen (Eds.). Tributes, Vol. 49. College Publications.

David Binder, Marco Tzschentke, Marius Müller, and Klaus Ostermann. 2024. Grokking the Sequent Calculus (Functional Pearl). *Proc. ACM Program. Lang.* 8, ICFP (aug 2024). doi:10.1145/3674639

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as Capabilities: Effect Handlers and Lightweight Effect Polymorphism. *Proc. ACM Program. Lang.* 4, OOPSLA (Nov. 2020). doi:10.1145/3428194

Edwin Brady. 2020. *Idris 2: Quantitative Type Theory in Action*. Technical Report. University of St Andrews, Scotland, UK. https://www.type-driven.org.uk/edwinb/papers/idris2.pdf

Yijia Chen and Lionel Parreaux. 2024. *The Long Way to Deforestation: A Type Inference and Elaboration Technique for Removing Intermediate Data Structures (Artifact)*. https://doi.org/10.5281/zenodo.11500626

Youyou Cong, Leo Osvald, Grégory M. Essertel, and Tiark Rompf. 2019. Compiling with Continuations, or Without? Whatever. *Proc. ACM Program. Lang.* 3, ICFP (July 2019), 79:1–79:28. doi:10.1145/3341643

Pierre-Louis Curien and Hugo Herbelin. 2000. The duality of computation. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 233–243. doi:10.1145/351240.351262

Pierre-Louis Curien and Guillaume Munch-Maccagnoni. 2010. The duality of computation under focus. In *IFIP international conference on theoretical computer science*. Springer, 165–181. doi:10.1007/978-3-642-15240-5_13

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (Oct. 1991), 451–490. doi:10.1145/115372.115320

Olivier Danvy. 1992. Back to Direct Style. In *Symposium Proceedings on 4th European Symposium on Programming (ESOP'92)*. Springer-Verlag, Berlin, Heidelberg, 130–150. doi:10.1007/3-540-55253-7_8

Olivier Danvy and Julia L. Lawall. 1992. Back to Direct Style II: First-Class Continuations. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming (LFP '92)*. Association for Computing Machinery, New York, NY, USA, 299–310. doi:10.1145/141471.141564

Olivier Danvy and Lasse R. Nielsen. 2003. A First-Order One-Pass CPS Transformation. *Theor. Comput. Sci.* 308, 1–3 (Nov. 2003), 239–257. doi:10.1016/S0304-3975(02)00733-8

Olivier Danvy and Frank Pfenning. 1995. *The Occurrence of Continuation Parameters in CPS Terms*. Technical Report CMU-CS-95-121. Department of Computer Science, Carnegie Mellon University.

Paul Downen. 2017. *Sequent Calculus: A Logic and a Language for Computation and Duality*. Ph. D. Dissertation. University of Oregon. doi:10.1017/S0956796818000023

Paul Downen and Zena M Ariola. 2014. The duality of construction. In *Programming Languages and Systems: 23rd European Symposium on Programming, ESOP 2014*. Springer, 249–269. doi:10.1007/978-3-642-54833-8_14

Paul Downen and Zena M Ariola. 2018. A tutorial on computational classical logic and the sequent calculus. *Journal of Functional Programming* 28 (2018), e3. doi:10.1017/S0956796818000023

Paul Downen and Zena M Ariola. 2020. Compiling with classical connectives. *Logical Methods in Computer Science* 16 (2020). doi:10.23638/LMCS-16(3:13)2020

Paul Downen and Zena M. Ariola. 2021. Duality in Action. In *6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 195)*, Naoki Kobayashi (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 1:1–1:32. doi:10.4230/LIPIcs.FSCD.2021.1

Paul Downen, Luke Maurer, Zena M. Ariola, and Simon Peyton Jones. 2016. Sequent calculus as a compiler intermediate language. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. Association for Computing Machinery, New York, NY, USA, 74–88. doi:10.1145/2951913.2951931

Paul Downen, Zachary Sullivan, Zena M. Ariola, and Simon Peyton Jones. 2019. Codata in Action. In *Programming Languages and Systems*, Luís Caires (Ed.). Springer International Publishing, Cham, 119–146. doi:10.1007/978-3-030-17184-1_5

Kavon Farvardin and John Reppy. 2020. From Folklore to Fact: Comparing Implementations of Stacks and Continuations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 75–90. doi:10.1145/3385412.3385994

Kavon Farvardin and John Reppy. 2021. A New Backend for Standard ML of New Jersey. In *Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages (IFL '20)*. Association for Computing Machinery, New York, NY, USA, 55–66. doi:10.1145/3462172.3462191

Matthias Felleisen. 1987. Reflections on Landin's J-operator: A Partly Historical Note. *Computer Languages* 12, 3 (1987), 197–207. doi:10.1016/0096-0551(87)90022-1

Matthias Felleisen, Daniel P. Friedman, Eugene Kohlbecker, and Bruce Duba. 1987. A syntactic theory of sequential control. *Theoretical Computer Science* 52, 3 (1987), 205–237. doi:10.1016/0304-3975(87)90109-5

Andrzej Filinski. 1989. Declarative Continuations: an Investigation of Duality in Programming Language Semantics. In *Category Theory and Computer Science*. Springer-Verlag, Berlin, Heidelberg, 224–249.

Cormac Flanagan, 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)*. Association for Computing Machinery, New York, NY, USA, 237–247. doi:10.1145/155090.155113

Gerhard Gentzen. 1935a. Untersuchungen über das logische Schließen. I. *Mathematische Zeitschrift* 39 (1935). doi:10.1007/BF01201353

Gerhard Gentzen. 1935b. Untersuchungen über das logische Schließen. II. *Mathematische Zeitschrift* 39 (1935). doi:10.1007/BF01201363

Jean-Yves Girard. 1987. Linear Logic. *Theoretical Computer Science* 50 (1987), 1–101. doi:10.1016/0304-3975(87)90045-4

Jean-Yves Girard. 1991. A new constructive logic: classic logic. *Mathematical Structures in Computer Science* 1, 3 (1991), 255–296. doi:10.1017/S0960129500001328

Jean-Yves Girard. 1993. On the unity of logic. *Annals of Pure and Applied Logic* 59, 3 (1993), 201–217. doi:10.1016/0168-0072(93)90093-S

Jean-Yves Girard. 2001. Locus Solum: From the rules of logic to the logic of rules. *Mathematical Structures in Computer Science* 11, 3 (2001), 301–506. doi:10.1017/S096012950100336X

Timothy G. Griffin. 1989. A Formulae-as-Type Notion of Control. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '90)*. Association for Computing Machinery, New York, NY, USA, 47–58. doi:10.1145/96709.96714

Tatsuya Hagino. 1989. Codatatypes in ML. *Journal of Symbolic Computation* 8, 6 (1989), 629–650. doi:10.1016/S0747-7171(89)80065-3

John Hatcliff and Olivier Danvy. 1994. A Generic Account of Continuation-Passing Styles. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '94)*. Association for Computing Machinery, New York, NY, USA, 458–471. doi:10.1145/174675.178053

Martin Hofmann. 1997. *Syntax and Semantics of Dependent Types*. Cambridge University Press, 79–130. doi:10.1017/CBO9780511526619.004

William A Howard. 1980. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism* 44 (1980), 479–490.

Yukiyoshi Kameyama. 2007. Axioms for Control Operators in the CPS Hierarchy. *Higher Order Symbol. Comput.* 20, 4 (Dec. 2007), 339–369. doi:10.1007/s10990-007-9009-x

Yukiyoshi Kameyama and Masahito Hasegawa. 2003. A Sound and Complete Axiomatization of Delimited Continuations. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP '03)*. Association for Computing Machinery, New York, NY, USA, 177–188. doi:10.1145/944705.944722

Andrew Kennedy. 2007. Compiling with Continuations, Continued. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP '07)*. Association for Computing Machinery, New York, NY, USA, 177–190. doi:10.1145/1291151.1291179

Oleg Kiselyov and Chung-chieh Shan. 2007. A Substructural Type System for Delimited Continuations. In *Typed Lambda Calculi and Applications*, Simona Ronchi Della Rocca (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 223–239. doi:10.1007/978-3-540-73228-0_17

Chun Kit Lam and Lionel Parreaux. 2024. Being Lazy When It Counts: Practical Constant-Time Memory Management for Functional Programming. In *Functional and Logic Programming: 17th International Symposium, FLOPS 2024, Kumamoto, Japan, May 15–17, 2024, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 188–216. doi:10.1007/978-981-97-2300-3_11

Peter John Landin. 1965a. Correspondence between ALGOL 60 and Church's Lambda-notation: part I. *Commun. ACM* 8, 2 (feb 1965), 89–101. doi:10.1145/363744.363749

Peter J Landin. 1965b. A generalization of jumps and labels. In *Report, UNIVAC Systems Programming Research*.

Peter J. Landin. 1998. A Generalization of Jumps and Labels. *Higher-Order and Symbolic Computation* 11, 2 (1998), 125–143. Reprint from Landin (1965b).

Julia L. Lawall and Olivier Danvy. 1993. Separating Stages in the Continuation-Passing Style Transformation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '93)*. Association for Computing Machinery, New York, NY, USA, 124–136. doi:10.1145/158511.158613

Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types, In Proceedings of the Workshop on Mathematically Structured Functional Programming. *Electronic Proceedings in Theoretical Computer Science*. doi:10.4204/eptcs.153.8

Roland Leißa, Marcel Köster, and Sebastian Hack. 2015. A graph-based higher-order intermediate representation. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '15)*. IEEE Computer Society, USA, 202–212. doi:10.5555/2738600.2738626

Paul Blain Levy, John Power, and Hayo Thielecke. 2003. Modelling environments in call-by-value programming languages. *Information and Computation* 185, 2 (2003), 182–210. doi:10.1016/S0890-5401(03)00088-9

Anton Lorenzen, Daan Leijen, and Wouter Swierstra. 2023. FP^2: Fully in-Place Functional Programming. *Proc. ACM Program. Lang.* 7, ICFP (aug 2023). doi:10.1145/3607840

Simon Marlow, Alexey Rodriguez Yakushev, and Simon Peyton Jones. 2007. Faster laziness using dynamic pointer tagging. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP '07)*. Association for Computing Machinery, New York, NY, USA, 277–288. doi:10.1145/1291151.1291194

Marek Materzok and Dariusz Biernacki. 2012. A Dynamic Interpretation of the CPS Hierarchy. In *Programming Languages and Systems*, Ranjit Jhala and Atsushi Igarashi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 296–311. doi:10.1007/978-3-642-35182-2_21

Luke Maurer, Paul Downen, Zena M. Ariola, and Simon L. Peyton Jones. 2017. Compiling Without Continuations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 482–494. doi:10.1145/3062341.3062380

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. 1997. *The definition of standard ML: revised*. MIT press. doi:10.7551/mitpress/2319.001.0001

Eugenio Moggi. 1989. Computational lambda-calculus and monads. In *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*. 14–23. doi:10.1109/LICS.1989.39155

Eugenio Moggi. 1991. Notions of computation and monads. *Information and Computation* 93, 1 (1991), 55–92. doi:10.1016/0890-5401(91)90052-4 Selections from 1989 IEEE Symposium on Logic in Computer Science.

Serkan Muhcu, Philipp Schuster, Michel Steuwer, and Jonathan Immanuel Brachthäuser. 2025. Multiple Resumptions and Local Mutable State, Directly. *Proc. ACM Program. Lang.* 9, ICFP (Aug. 2025). doi:10.1145/3747529

Marius Müller, Philipp Schuster, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. 2023. Back to Direct Style: Typed and Tight. *Proc. ACM Program. Lang.* 7, OOPSLA1 (apr 2023). doi:10.1145/3586056

Guillaume Munch-Maccagnoni. 2009. Focalisation and Classical Realisability. In *Computer Science Logic*, Erich Grädel and Reinhard Kahle (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 409–423. doi:10.1007/978-3-642-04027-6_30

Guillaume Munch-Maccagnoni. 2013. *Syntax and Models of a non-Associative Composition of Programs and Proofs*. Ph. D. Dissertation. Univ. Paris Diderot.

Marius Müller, Philipp Schuster, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. 2023. *Back to Direct Style: Typed and Tight*. Extended Technical Report. University of Tübingen, Germany. https://se.informatik.uni-tuebingen.de/publications/mueller23continuation.

Marius Müller, Marco Tzschentke, Philipp Schuster, Jonathan Immanuel Brachthäuser, Klaus Ostermann, and David Binder. 2025a. *SCC - Sequent Calculus Compiler*. https://github.com/SequentCalculus/sequent-calculus-compiler

Marius Müller, Marco Tzschentke, Philipp Schuster, Jonathan Immanuel Brachthäuser, Klaus Ostermann, and David Binder. 2025b. *SCC - Sequent Calculus Compiler - Benchmark Suite*. https://github.com/SequentCalculus/sequent-calculus-compiler-bench

Lasse R. Nielsen. 2002. A Simple Correctness Proof of the Direct-Style Transformation. *BRICS Report Series* 9, 2 (Jan. 2002). doi:10.7146/brics.v9i2.21719

Klaus Ostermann, David Binder, Ingo Skupin, Tim Süberkrüb, and Paul Downen. 2022. Introduction and elimination, left and right. *Proceedings of the ACM on Programming Languages* 6, ICFP (2022), 438–465. doi:10.1145/3547637

Zoe Paraskevopoulou and Anvay Grover. 2021. Compiling with continuations, correctly. *Proc. ACM Program. Lang.* 5, OOPSLA (Oct. 2021). doi:10.1145/3485491

Michel Parigot. 1992. $\lambda\mu$-Calculus: An algorithmic interpretation of classical natural deduction. In *Logic Programming and Automated Reasoning*, Andrei Voronkov (Ed.). Springer, Berlin, Heidelberg, 190–201.

Will Partain. 1993. The nofib benchmark suite of Haskell programs. In *Functional Programming, Glasgow 1992: Proceedings of the 1992 Glasgow Workshop on Functional Programming, Ayr, Scotland, 6–8 July 1992*. Springer, 195–202.

Simon L. Peyton Jones. 1992. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of functional programming* 2, 2 (1992), 127–202. doi:10.1017/S0956796800000319

Alex Reinking, Ningning Xie, Leonardo de Moura, and Daan Leijen. 2021. Perceus: Garbage free reference counting with reuse. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, USA, 96–111. doi:10.1145/3453483.3454032

John C. Reynolds. 1972. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM annual conference*. ACM, New York, NY, USA, 717–740.

Laurence Rideau, Bernard Paul Serpette, and Xavier Leroy. 2008. Tilting at windmills with Coq: Formal verification of a compilation algorithm for parallel moves. *Journal of Automated Reasoning* 40 (2008), 307–326. doi:10.1007/s10817-007-9096-8

Amr Sabry 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)*. Association for Computing Machinery, New York, NY, USA, 288–298. doi:10.1145/141471.141563

Amr Sabry and Matthias Felleisen. 1993. Reasoning about Programs in Continuation-Passing Style. *LISP and Symbolic Computation* 6 (1993), 289–360. doi:10.1007/BF01019462

Amr Sabry and Philip Wadler. 1997. A Reflection on Call-by-Value. *ACM Trans. Program. Lang. Syst.* 19, 6 (Nov. 1997), 916–941. doi:10.1145/267959.269968

Philipp Schuster, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. 2020. Compiling Effect Handlers in Capability-Passing Style. *Proc. ACM Program. Lang.* 4, ICFP (Aug. 2020). doi:10.1145/3408975

Philipp Schuster, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. 2022. Region-based Resource Management and Lexical Exception Handlers in Continuation-Passing Style. In *Programming Languages and Systems: 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 492–519. doi:10.1007/978-3-030-99336-8_18

Philipp Schuster, Marius Müller, Klaus Ostermann, and Jonathan Immanuel Brachthäuser. 2025a. Compiling Classical Sequent Calculus to Stock Hardware: The Duality of Compilation. *Proc. ACM Program. Lang.* 9, OOPSLA1 (apr 2025). doi:10.1145/3720507

Philipp Schuster, Marius Müller, Klaus Ostermann, and Jonathan Immanuel Brachthäuser. 2025b. *Artifact of the paper 'Compiling Classical Sequent Calculus to Stock Hardware: The Duality of Compilation'*. doi:10.5281/zenodo.14917573

Anton Setzer. 2003. Java as a Functional Programming Language. In *Types for Proofs and Programs*, Herman Geuvers and Freek Wiedijk (Eds.). Springer, Berlin, Heidelberg, 279–298. doi:10.1007/3-540-39185-1_16

Zhong Shao and Andrew W. Appel. 2000. Efficient and safe-for-space closure conversion. *ACM Trans. Program. Lang. Syst.* 22, 1 (Jan. 2000), 129–161. doi:10.1145/345099.345125

KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting Effect Handlers onto OCaml. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 206–221. doi:10.1145/3453483.3454039

Guy L. Steele. 1978. *Rabbit: A Compiler for Scheme*. Technical Report. USA. https://hdl.handle.net/1721.1/6913

Morten Heine Sørensen and Paweł Urzyczyn. 2006. *Lectures on the Curry-Howard Isomorphism*. Studies in Logic and the Foundations of Mathematics, Vol. 149. Elsevier.

Hayo Thielecke. 1998. An Introduction to Landin's "A Generalization of Jumps and Labels". *Higher Order Symbol. Comput.* 11, 2 (sep 1998), 117–123. doi:10.1023/A:1010060315625

Paulo Torrens, Dominic Orchard, and Cristiano Vasconcellos. 2024. On the Operational Theory of the CPS-Calculus: Towards a Theoretical Foundation for IRs. *Proc. ACM Program. Lang.* 8, ICFP (Aug. 2024). doi:10.1145/3674630

Philip Wadler. 2003. Call-by-value is dual to call-by-name. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP '03)*. ACM, New York, NY, USA, 189–201. doi:10.1145/944705.944723

Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. 2014. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0*. Technical Report UCB/EECS-2014-54. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html

Aaron Weiss, Olek Gierczak, Daniel Patterson, and Amal Ahmed. 2021. Oxide: The Essence of Rust. (2021). doi:10.48550/arXiv.1903.00982

J. Weizenbaum. 1963. Symmetric list processor. *Commun. ACM* 6, 9 (sep 1963), 524–536. doi:10.1145/367593.367617

Joseph Weizenbaum. 1969. Recovery of reentrant list structures in SLIP. *Commun. ACM* 12, 7 (jul 1969), 370–372. doi:10.1145/363156.363159

Ningning Xie and Daan Leijen. 2021. Generalized Evidence Passing for Effect Handlers: Efficient Compilation of Effect Handlers to C. *Proc. ACM Program. Lang.* 5, ICFP (aug 2021). doi:10.1145/3473576

Noam Zeilberger. 2008. On the unity of duality. *Annals of Pure and Applied Logic* 153, 1 (2008), 66–96. doi:10.1016/j.apal.2008.01.001

Noam Zeilberger. 2009. *The Logical Basis of Evaluation Order and Pattern-Matching.* Ph. D. Dissertation. Carnegie Mellon University, USA.