Compiling with the Sequent Calculus

MARIUS MÜLLER, University of Tübingen, Germany
DAVID BINDER, University of Kent, United Kingdom
MARCO TZSCHENTKE, University of Tübingen, Germany
PHILIPP SCHUSTER, University of Tübingen, Germany
KLAUS OSTERMANN, University of Tübingen, Germany
JONATHAN IMMANUEL BRACHTHÄUSER, University of Tübingen, Germany

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 paper, 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 our work as a continuation, and generalization, of Andrew Appel's landmark work on "Compiling with Continuations".

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Theory of computation** → *Type theory*; *Linear logic*; *Control primitives*; Operational semantics.

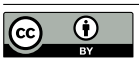
Additional Key Words and Phrases: intermediate representations, sequent calculus, continuations, codata types, control effects, code generation

ACM Reference Format:
Marius Müller, David Binder, Marco Tzschentke, Philipp Schuster, Klaus Ostermann, and Jonathan Immanuel Brachthäuser. 2026. Compiling with the Sequent Calculus. *ACM Trans. Program. Lang. Syst.* 1, 1 (September 2026), 87 pages. https://doi.org/10.1145/3845991

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

Authors' Contact Information: Marius Müller, Department of Computer Science, University of Tübingen, Tübingen, Germany, mari.mueller@uni-tuebingen.de; David Binder, School of Computing, University of Kent, Canterbury, United Kingdom, D.Binder@kent.ac.uk; Marco Tzschentke, Department of Computer Science, University of Tübingen, Tübingen, Germany, marco.tzschentke@uni-tuebingen.de; Philipp Schuster, Department of Computer Science, University of Tübingen, Tübingen, Germany, philipp.schuster@uni-tuebingen.de; Klaus Ostermann, Department of Computer Science, University of Tübingen, Tübingen, Germany, klaus.ostermann@uni-tuebingen.de; Jonathan Immanuel Brachthäuser, Department of Computer Science, University of Tübingen, Tübingen, Germany, jonathan.brachthaeuser@uni-tuebingen.de.



ACM 1558-4593/2026/9-ART
https://doi.org/10.1145/3845991

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. What should an IR look like that serves both purposes?

1.1 Classical Sequent Calculus

We – as well as other researchers – have argued that we should employ Gentzen [1935a]'s classical sequent calculus as a basis for an attractive IR that bridges the gap between high- and low-level languages [Binder et al. 2024; Downen et al. 2016; Schuster et al. 2025a]. Sequent-calculus-based intermediate representations have specific advantages – that we are going to discuss shortly – which distinguish them from IRs based on natural deduction, such as direct style or continuation-passing style (CPS). These benefits can be explained by the characteristic property of term assignment systems for the classical sequent calculus [Curien and Herbelin 2000; Downen 2017; Wadler 2003; Zeilberger 2008]: There are three syntactic categories of producers p (variously called programs, terms, proofs, etc.), consumers c (variously called contexts, coterms, refutations, etc.), and statements s (variously called commands, contradictions, etc.) that replace the single syntactic category of expressions found in most functional languages. Let us look at them in more detail.

Producers p construct an element of some type τ and correspond to expressions in most programming languages. But instead of the familiar typing judgment $\Gamma \vdash e : \tau$ for expressions, we write $\Gamma \vdash p :^{\text{prd}} \tau$ to make it explicit in the judgment that we are typing a producer. *Consumers* c, in contrast, destruct an element of some type τ, and we write $\Gamma \vdash c :^{\text{cns}} \tau$ for the corresponding typing judgment. Intuitively, they represent evaluation contexts or continuations and are not typically found as first-class entities in programming languages. *Cuts* $\langle p \mid c \rangle$ are one form of statement – the most important, but not necessarily the only one – which combine a producer and a consumer of the same type. Statements do not have a type, and the following typing rule for cuts therefore omits a type annotation in the conclusion.

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

Operationally, cuts correspond to redexes, and computation or reduction corresponds to cut elimination. To make this clearer, let us consider an example. Producers of the list type are terms built from its constructors `Nil` and `Cons`, such as the two-element list $\texttt{Cons}(1, \texttt{Cons}(2, \texttt{Nil}))$. Consumers of that type are built by pattern matching on `Nil` and `Cons` and returning a statement in each branch, as in the consumer $\textbf{case}\ \{\texttt{Nil} \Rightarrow s_1, \texttt{Cons}(x, xs) \Rightarrow s_2\}$. Combining the example producer with the example consumer, we can therefore instantiate the typing judgment for cuts as follows:

$$\frac{\begin{array}{c}\Gamma \vdash \texttt{Cons}(1, \texttt{Cons}(2, \texttt{Nil})) :^{\text{prd}} \texttt{List}[\textbf{i64}] \\ \Gamma \vdash \textbf{case}\ \{\texttt{Nil} \Rightarrow s_1, \texttt{Cons}(x, xs) \Rightarrow s_2\} :^{\text{cns}} \texttt{List}[\textbf{i64}]\end{array}}{\Gamma \vdash \langle \texttt{Cons}(1, \texttt{Cons}(2, \texttt{Nil})) \mid \textbf{case}\ \{\texttt{Nil} \Rightarrow s_1, \texttt{Cons}(x, xs) \Rightarrow s_2\} \rangle}\ \textsc{Cut}$$

This cut can be eliminated by reducing it to the statement $s_2[x \mapsto 1, xs \mapsto \texttt{Cons}(2, \texttt{Nil})]$. The interesting thing to observe is that instead of a familiar case expression like $p.\textbf{case}\ \{\ldots\}$ that

combines the scrutinee and the branches in a single expression, the sequent calculus *factors* these case expressions into three parts: The producer p, the consumer **case** $\{\dots\}$, and the cut $\langle p \mid c \rangle$.

1.1.1 Expressing Advanced Control-Flow. Modern programs are interactive and continually communicate with external hardware and software components. This communication must be asynchronous in order to utilize hardware effectively, and programming languages provide first-class support in the form of features like resumable exceptions, generators, async IO, and effect handlers. One possibility to compile these language features that rely on non-local control flow 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, i.e. without the need for a low-level encoding, such as state machines, or support from a runtime system.

Term assignment systems for the classical sequent calculus are one particularly well-suited such IR due to the symmetry between producers and consumers [Downen and Ariola 2014; Ostermann et al. 2022]. Consumers are first-class: They can be named, and hence, they can also be stored, duplicated, and dropped. The resulting logical system is classical and thereby more expressive than intuitionistic logic. In particular, a consumer can be given a name with the right activation rule,

$$\frac{\Gamma, \alpha :^{\text{cns}} \tau \vdash s}{\Gamma \vdash \mu\alpha.s :^{\text{prd}} \tau}\ \textsc{Act-R}$$

Intuitively, the producer $\mu\alpha.s$ captures and reifies its surrounding context as a first-class consumer in the *covariable* α. This rule can be used to derive classical reasoning principles such as the law of excluded middle or Peirce's law [Downen and Ariola 2018; Wadler 2003], which correspond to particular control operators [Griffin 1989], such as `call/cc`. In fact, the μ-abstraction itself has originally been conceived as a control operator [Parigot 1992].

Using this rule also allows us to retain much of the nesting structure of original programs in the high-level source language translated into an IR based on classical sequent calculus. For example, nested calls are translated as follows

$$[\![f(g(42))]\!]_\alpha = f(\mu\beta.g(42, \beta), \alpha)$$

Both f and g take a producer argument and an additional consumer argument after the translation. Here, α stands for the outer context of the whole expression. In order to call g, we capture its surrounding context and name it β. Despite giving us access to first-class control, we require neither an encoding nor a runtime system, as we will see. In comparison, a translation to continuation-passing style (CPS), while also giving access to first-class control, changes the nesting structure of programs to make the evaluation order explicit.

1.1.2 Controlling Evaluation Order. The two evaluation orders call-by-value (CBV) and call-by-name (CBN) 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. For our IR, we would like to express both evaluation orders naturally and uniformly, preserving the original structure of source programs as much as possible. A key to this is the left activation rule.

$$\frac{\Gamma, x :^{\text{prd}} \tau \vdash s}{\Gamma \vdash \tilde{\mu}x.s :^{\text{cns}} \tau}\ \textsc{Act-L}$$

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

The consumer $\tilde{\mu}x.s$, dually to $\mu\alpha.s$, reifies a surrounding producer and names it with variable x, just as typical let-bindings found in many programming languages. Consider again the nested call $f(g(42))$. Under CBN, control first enters f, which can choose to demand the result of g. Under CBV, control first enters g, which can choose to return its result to f. We continue to translate the nested call, in an additional step, as follows:

$$[\![f(g(42))]\!]_\alpha = \langle \mu\beta.g(42,\beta) \mid \tilde{\mu}x.f(x,\alpha) \rangle$$

This program representation appears to be more akin to a typical CPS translation under CBV, but in contrast to CPS, it still does not prescribe the evaluation order, both are possible. Under CBV, we give priority to the producer and substitute the consumer into it, and under CBN, we give priority to the consumer and substitute the producer into it, resulting in two different statements.

$$\langle \mu\beta.g(42,\beta) \mid \tilde{\mu}x.f(x,\alpha) \rangle \begin{array}{ll} \rightarrow_{\text{CBV}} & g(42, \tilde{\mu}x.f(x,\alpha)) \\ \rightarrow_{\text{CBN}} & f(\mu\beta.g(42,\beta), \alpha) \end{array}$$

While we can 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; Downen 2017; Downen and Ariola 2020, 2021; Girard 1991, 1993; Munch-Maccagnoni 2009], 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. This symmetry between data and codata types and between producers and consumers gives us a richer representation of program contexts than in CPS, as we discuss next.

1.1.3 Representing First-Class Contexts. 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 therefore typically add a bottom type $\bot$ and model continuations as functions with one input,

$$\tau \to \bot$$

This implicitly marks continuations as consumers for their input. As functions do not return either in CPS, their return type is also $\bot$. 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

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

However, functions and continuations are treated quite differently in practice, for example, when performing optimizations. Therefore, continuations are often given a built-in type different from functions, e.g., an explicit negation type $\neg\tau$ [Kennedy 2007].

Our notion of consumers can model arbitrary program contexts directly, generalizing continuations. Indeed, a consumer can, for example, be a first-class pattern-matching context.

$$\frac{\textbf{data}\ T\,\{\,K_1(\Gamma_1), \ldots\,\} \in \Theta \qquad \forall i\ :\ \Gamma, \Gamma_i \vdash s_i}{\Theta \mid \Gamma \vdash \textbf{case}\,\{\,K_1(\Gamma_1) \Rightarrow s_1, \ldots\,\} :^{\text{cns}} T}\ \textsc{Case}$$

Continuations for primitive integers are then pattern-matching contexts for a particular data type with a single constructor.

$$\frac{\mathbf{data}\,\mathtt{Cont}\,\{\mathtt{Ret}(x :^{\mathrm{prd}} \mathbf{i64})\} \in \Theta \qquad \Gamma, x :^{\mathrm{prd}} \mathbf{i64} \vdash s}{\Theta \mid \Gamma \vdash \mathbf{case}\,\{\mathtt{Ret}(x :^{\mathrm{prd}} \mathbf{i64}) \Rightarrow s\} :^{\mathrm{cns}} \mathtt{Cont}}\ \textsc{Case}$$

While it may seem that this requires us to box integers when returning to such a context, we will see that our compilation technique makes sure that the integer stays in a register and is not boxed. Even multiple integers would be returned in registers.

It is then also natural to return to a pattern-matching context that discriminates between an `Ok`- and an `Error`-case.

$$\frac{\mathbf{data}\,\mathtt{Res}\,\{\mathtt{Ok}(x :^{\mathrm{prd}} \mathbf{i64}), \mathtt{Err}(i :^{\mathrm{prd}} \mathbf{i64})\} \in \Theta \qquad \Gamma, x :^{\mathrm{prd}} \mathbf{i64} \vdash s_1, \quad \Gamma, i :^{\mathrm{prd}} \mathbf{i64} \vdash s_2}{\Theta \mid \Gamma \vdash \mathbf{case}\,\{\mathtt{Ok}(x :^{\mathrm{prd}} \mathbf{i64}) \Rightarrow s_1, \mathtt{Err}(i :^{\mathrm{prd}} \mathbf{i64}) \Rightarrow s_2\} :^{\mathrm{cns}} \mathtt{Res}}\ \textsc{Case}$$

Here too, our compilation technique makes sure that neither the result nor the error code are boxed: We return either of them in a register. Similarly to pattern-matching contexts, consumers can also model function-application frames, making them first-class as well.

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.

1.2 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 [Binder et al. 2024; Downen et al. 2016], in this paper 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 provide an implementation of the complete compilation pipeline. While in this article 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 article 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.

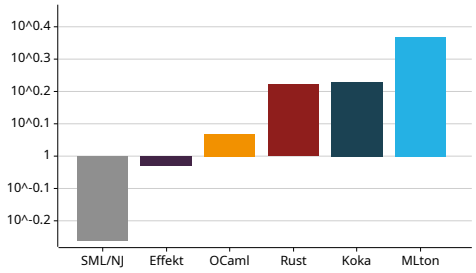


Fig. 1. Geometric mean of relative speedups over all benchmark programs compared to our implementation, higher is better.

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.

The rest of this article is structured as follows:

- In Section 2 we give an example-based overview of our compilation pipeline.
- In Section 3 we introduce the surface language **Fun**.
- In Section 4 we introduce the intermediate language **Core**.
- In Section 5 we show how to translate programs from the surface language **Fun** to the intermediate language **Core**.
- In Section 6 and Section 7 we show how to transform **Core** to a normal form **AxCut** which serves as a low-level intermediate language suitable for generating machine code. More concretely, in Section 6 we show a transformation on programs in **Core** that names all intermediate computations, similar to ANF or focusing transformations. Moreover, we present a transformation to shrink the language by removing redundant constructs. Then, in Section 7, we give a translation that makes the structural rules of weakening, contraction, and exchange explicit.
- In Section 8 we show how to generate machine code from **AxCut**.
- In Section 9 we briefly discuss our implementation of the compilation pipeline and report on our benchmark results.
- We discuss related work in Section 10 and conclude in Section 12.

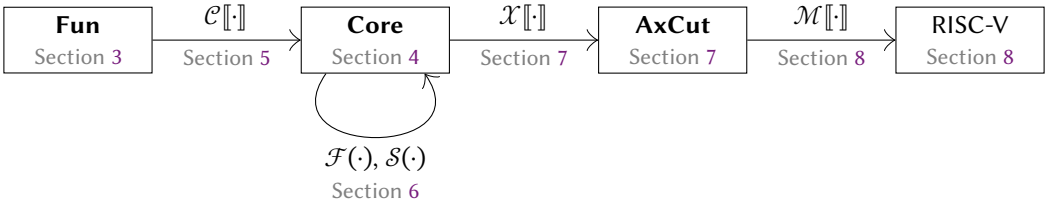
Prior Work. This article is based on two published papers:

- In Binder et al. [2024] we have given an introduction to languages based on sequent calculus by presenting a translation from a high-level functional surface language to an IR based on sequent calculus. The surface language and sequent-calculus IR in this work are based on those in Binder et al. [2024], but we choose call-by-name evaluation order for codata, showing how different evaluation orders can be mixed when using an IR based on classical sequent calculus. Moreover, the translation given in Binder et al. [2024] generates a lot of administrative redexes. In this work we show how such a translation can be done in an efficient way, using techniques found in one-pass, first-order, compositional CPS translations.

- In Schuster et al. [2025a] we have picked up where we had left off in Binder et al. [2024]. We have shown how an IR based on classical sequent calculus can be transformed into a normal form that is suitable for direct code generation, and then how to actually generate code. The normalization transformation consists of several steps: Focusing gives names to all parts of a program, shrinking removes redundant constructs from the IR, and linearization makes the structural rules of weakening, contraction, and exchange explicit. In Schuster et al. [2025a], these steps are described with a varying degree of precision. In particular, all of them are only presented on the part of the language corresponding to the logical core, and the linearization step is only sketched by means of an example. In this work we make all of these steps precise for the full language and moreover show how these transformations can be performed efficiently, avoiding problems of a naive implementation. Furthermore, we include some of the steps that were left out in Schuster et al. [2025a]'s formal description of code generation.

2 Overview

In this section, we give an overview of our complete compilation pipeline, which can be illustrated as follows:



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.

2.1 The Surface Language Fun

The surface language **Fun** is in many regards an ordinary functional programming language. 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 2.1 (Functions as Codata).

$$\begin{aligned}\mathbf{i64} \to \mathbf{i64} &\mathrel{\hat{=}} \mathbf{codata}\,\mathtt{Fun}\,\{\,\mathtt{apply}(x : \mathbf{i64}) : \mathbf{i64}\,\}\\ \lambda x.t &\mathrel{\hat{=}} \mathbf{new}\,\{\,\mathtt{apply}(x) \Rightarrow t\,\}\\ t_1\ t_2 &\mathrel{\hat{=}} t_1.\mathtt{apply}(t_2)\end{aligned}$$

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

Example 2.2 (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 2.3 (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 the 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**. **label** behaves like `let/cc` [Reynolds 1972]: It captures the current program context and makes it available in its body, but the invocation of such a continuation is performed with the explicit **goto** construct. 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 2.4 (Multiplication with Early Return).

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

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

In the rest of this section, we use the code from Example 2.4 to illustrate all intermediate stages of the compiler.

2.2 The High-Level Intermediate Language Core

In the first step (detailed in Section 5), we translate the surface language **Fun** into the high-level intermediate language **Core**. The foundation of **Core** is the $\lambda\mu\tilde{\mu}$-calculus [Curien and Herbelin 2000], 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. 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 2.4 of multiplication of a list with early return, this time in **Core**.

*Example 2.5 (Multiplication with Early Return in **Core**).*

```
def rmult(xs :prd List[i64], β :cns i64) { ⟨ μα.rmult_rec(xs, α, α) | β ⟩ }
def rmult_rec(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.rmult_rec(ys, α, γ) | β ⟩ } } ⟩
}
```

Similar to how functions in CPS receive an additional continuation argument, both top-level functions have an additional covariable β that stands for the current program context in place of a return type. This covariable is made available in the body of rmult, where it is bound to covariable α by a μ-abstraction and then used twice when calling rmult_rec. This implements the control operator **label**. As in the original program, the first covariable α abstracted by rmult_rec 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 rmult_rec. 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 rmult_rec, 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 rmult_rec 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 4.

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, rmult_rec has two, one for normal return and one for early return, which is more akin to double-barreled CPS [Kennedy 2007; Thielecke 2002]. Similar to top-level functions, destructors of codata types in **Core** do not have return types either, but instead receive an additional consumer argument. For example, the codata type of first-class functions from Example 2.1 becomes

$$\textbf{codata}\ \mathsf{Fun}\,\{\,\mathsf{apply}(x :^{\mathsf{prd}} \textbf{i64}, \alpha :^{\mathsf{cns}} \textbf{i64})\,\}$$

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[2].

Example 2.6 (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 done and pull.

$$\textbf{codata}\ \mathtt{Receiver}\,\{\,\mathtt{stop},\ \mathtt{push}(n :^{\mathsf{prd}} \textbf{i64}, \mathit{rest} :^{\mathsf{prd}} \mathtt{Sender})\,\}$$
$$\textbf{codata}\ \mathtt{Sender}\,\{\,\mathtt{done},\ \mathtt{pull}(\mathit{next} :^{\mathsf{prd}} \mathtt{Receiver})\,\}$$

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 the integer 0, and a receiver discard, which discards all integers pushed into it.

$$\begin{aligned}
&\textbf{def}\ \mathtt{zeroes}(r :^{\mathsf{prd}} \mathtt{Receiver})\,\{\\
&\quad\langle r \mid \mathtt{push}(0, \textbf{new}\,\{\,\mathtt{done} \Rightarrow \textbf{exit}\,0,\ \mathtt{pull}(\mathit{next}) \Rightarrow \mathtt{zeroes}(\mathit{next})\,\})\rangle\\
&\}\\
&\textbf{def}\ \mathtt{discard}(s :^{\mathsf{prd}} \mathtt{Sender})\,\{\\
&\quad\langle s \mid \mathtt{pull}(\textbf{new}\,\{\,\mathtt{stop} \Rightarrow \textbf{exit}\,0,\ \mathtt{push}(n, \mathit{rest}) \Rightarrow \mathtt{discard}(\mathit{rest})\,\})\rangle\\
&\}\\
&\textbf{def}\ \mathtt{connect}()\,\{\,\mathtt{discard}(\textbf{new}\,\{\,\mathtt{done} \Rightarrow \textbf{exit}\,0,\ \mathtt{pull}(\mathit{next}) \Rightarrow \mathtt{zeroes}(\mathit{next})\,\})\,\}
\end{aligned}$$

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

[2]To facilitate such definitions, we could add some kind of type expressing that a top-level function or destructor does not return. This is similar to how the never type "!" is used in the programming language Rust.

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.

2.3 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**.

2.3.1 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 2.5, the details are discussed in Section 6.1 (also cf. Section 4.2).

Example 2.7 (Focusing of Multiplication with Early Return). The only part of the program in Example 2.5 that has a non-(co)variable term in argument position is the **else** branch in rmult_rec. Focusing this statement yields (here $\mathcal{F}$ is the focusing function)

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

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.

2.3.2 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 2.7 above. These critical pairs play an important role in the reduction theory of our sequent-calculus-based languages, determining the evaluation order, as we will discuss in detail in Section 4.2. 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 6.2.

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 2.8 (Shrinking of Multiplication with Early Return).

```
def rmult(xs :prd List[i64], β :cns Cont) { rmult_rec(xs, β, β) }
def rmult_rec(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.rmult_rec(ys, α, γ) | case { Ret(r) ⇒ ⟨y * r | μ̃z.⟨Ret(z) | β⟩⟩ }⟩
        }
    }⟩
}
```

First note that compared to the version in Example 2.5, 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 rmult_rec. 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 rmult_rec that was generated by focusing, as discussed in Example 2.7. 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.

2.4 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 Section 7.1.

With the unified syntax of **AxCut** and after inferring explicit substitutions, our running example of multiplication of a list with early return from Example 2.8 looks as follows (ignore the colors for now, they only become relevant in Section 2.5).

*Example 2.9 (Multiplication with Early Return in **AxCut**).*

```
def rmult(xs :prd List[i64], β :cns Cont) {
    substitute[xs := xs, β := β, γ := β]; rmult_rec(xs, β, γ)
}
def rmult_rec(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)
          };
          rmult_rec(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, because the information they provide on where variables are duplicated and dropped can be used 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.

2.5 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 Section 8.

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

Example 2.10 (Code Generation for Multiplication with Early Return). We generate the following machine code for the highlighted parts in Example 2.9.

```
rmult :
  ShareBlock x6
  mv x8 x6
  mv x9 x7
  j rmult_rec
```

```
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 (with the source on the right and the target on the left), `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 Section 8.3 - Section 8.6.

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 article.

3 The Surface Language Fun

In this section, we introduce the language **Fun** that we want to compile to machine code. This language is in many respects an ordinary, expression-oriented, functional programming language. 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 2.1. These features are not only interesting from the point of view of programmers using them, but also from a compilation perspective. We start with the syntax, then give the typing rules, and finally define the operational semantics.

3.1 Syntax

*Definition 3.1 (Syntax of **Fun**).* For variables and names we assume that $x, y, \ldots \in$ Variables, $\alpha, \beta, \ldots \in$ Covariables, $T \in$ TypeNames, $K, D, X \in$ Tags, and $f \in$ Labels.

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

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 $p + p$ and conditional expressions like **if** $p_1 \equiv 0 \{ p_2 \}$ **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 $p.\mathbf{case}\,\{ K(\Gamma) \Rightarrow p, \ldots \}$. Data type declarations are specified on the top-level with the syntactic form $\mathbf{data}\, T\, \{ K(\Gamma), \ldots \}$. 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 [Downen et al. 2019; Hagino 1989] 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 $\mathbf{new}\, \{ D(\Gamma) \Rightarrow p, \ldots \}$. Just as for data types, codata type declarations are specified on the top-level with the syntactic form $\mathbf{codata}\, T\, \{ D(\Gamma) : \tau, \ldots \}$. 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 2.2 in Section 2.1, and we have also illustrated the call-by-name evaluation order in Example 2.3. We have also mentioned that first-class functions are just another codata type with one `apply` destructor [Setzer 2003] 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 3.2 (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 which respectively capture and invoke a program context, or continuation, α, essentially like `let/cc` [Reynolds 1972]. An additional control operator is the construct **exit** p which simply terminates the program. 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 2.4 in Section 2.1, where we have implemented multiplication of a list of integers with early return.

3.2 Typing Rules

Typing in **Fun** is split into several different judgment forms defined in Figure 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 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 $\Theta \mid \Gamma \vdash c :^{\text{cns}} \tau$ 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 its argument, i.e., of the exit code, must be **i64**. The typing rule for covariables, the only kind of consumer, simply checks whether the covariable is in the context.

Well-Formed Declarations $\Theta \vdash \delta$ Ok

$$\frac{\Theta \mid \Gamma \vdash p : \tau}{\Theta \vdash \mathbf{def}\ f(\Gamma) : \tau\,\{\,p\,\}\ \text{Ok}}\ \textsc{Wf-Def} \qquad \cdots$$

Argument Typing $\Theta \mid \Gamma \vdash \sigma : \Gamma'$

$$\frac{}{\Theta \mid \Gamma \vdash \bullet : \bullet}\ \textsc{Arg}_1 \qquad \frac{\Theta \mid \Gamma \vdash \sigma : \Gamma' \quad \Theta \mid \Gamma \vdash p : \tau}{\Theta \mid \Gamma \vdash \sigma, p : \Gamma', x : \tau}\ \textsc{Arg}_2 \qquad \frac{\Theta \mid \Gamma \vdash \sigma : \Gamma' \quad \Theta \mid \Gamma \vdash c :^{\text{cns}} \tau}{\Theta \mid \Gamma \vdash \sigma, c : \Gamma', \alpha :^{\text{cns}} \tau}\ \textsc{Arg}_3$$

Producer Typing $\Theta \mid \Gamma \vdash p : \tau$

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

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

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

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

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

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

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

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

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

$$\frac{\alpha :^{\text{cns}} \tau \in \Gamma}{\Gamma \vdash \alpha :^{\text{cns}} \tau}\ \textsc{Covar}$$

Fig. 2. Typing rules for **Fun**.

3.3 Operational Semantics

While **Fun** is in many regards a standard functional programming language, the definition of its operational semantics is somewhat complicated by the control operators **label** and **goto**, and by incorporating both call-by-value and call-by-name evaluation orders. To account for both, we define the operational semantics in terms of a substitution-based abstract machine with an explicit stack. Definition 3.3 defines its syntax.

*Definition 3.3 (Syntax of Operational Semantics of **Fun**).*

$$
\begin{array}{rcll}
A & ::= & \mathbf{let}\ x = \Box;\ p \mid f(v, \Box, \sigma) \mid \Box + p \mid n + \Box \mid \mathbf{if}\ \Box \equiv 0\ \{\, p \,\}\ \mathbf{else}\ \{\, p \,\} & \textit{By-Value Frames} \\
 & \mid & K(v, \Box, \sigma) \mid p.D(v, \Box, \sigma) \mid \Box.\mathbf{case}\ \{\ldots\} & \\
F & ::= & A \mid \Box.D(v) & \textit{Frames} \\
S & ::= & \mathbf{exit} \mid F :: S & \textit{Stacks} \\
M & ::= & \langle p \parallel S \parallel \Theta \rangle & \textit{States} \\
c & ::= & \alpha \mid S & \textit{Consumers} \\
v & ::= & n \mid K(v) \mid \mathbf{new}\ \{\ldots\} \mid S & \textit{Values} \\
a & ::= & v \mid p \quad \text{where } p : \mathbf{codata}\, T\, \{\ldots\} & \textit{Argument Values} \\
v & ::= & \bullet \mid v, a &
\end{array}
$$

A stack is a list of evaluation frames F, ending with an **exit** frame. Evaluation frames mirror each of the language constructs but with one hole. They are divided into by-value frames A and by-name frames $\Box.D(v)$. While by-value frames force most computation, only by-name frames force computation at codata types. A machine state M consists of a producer p in focus, a stack S, and the whole program Θ. Final states are of the form $\langle n \parallel \mathbf{exit} \parallel \Theta \rangle$ with the integer n being the exit code of the evaluated program. Program execution is started in the state $\langle p \parallel \mathbf{exit} \parallel \Theta \rangle$, where p is the body of the main function. Since stacks are first-class due to the control operator **label**, we add them as an additional kind of consumer. Note, however, that they only exist at runtime and cannot occur in a program before execution. We also slightly generalize the **goto** c (p) construct and its typing rule Goto to allow for S to occur as its consumer. Moreover, we define values v and argument values a. While values are ready to interact, argument values will not be forced in argument position. Values are integers, constructors whose arguments are all argument values, copattern matches, and stacks. Argument values are exactly those terms that can be substituted for a variable or covariable. Hence, they contain values, but also all producers of codata types, which follow call-by-name. We write v for a list of argument values. Note that frames implement left-to-right evaluation, because all arguments to the left of the hole must be argument values.

The hole in a frame always stands for a producer, because consumers never need further evaluation before substituting them. Therefore, stacks are typed as consumers, with their type being determined by the type of their hole. **exit** frames (rule Done) always consume an integer. An evaluation frame F (rule Frame) is a well-typed consumer of type τ if it is a well-typed producer of some type τ_0 when filling its hole with a fresh variable of type τ. The rest of the stack S must then be a consumer of type τ_0.

$$\frac{}{\bullet \vdash \mathbf{exit} :^{\mathrm{cns}} \mathbf{i64}}\ \textsc{Done} \qquad \frac{x : \tau \vdash F[x] : \tau_0 \quad (x \text{ fresh}) \qquad \bullet \vdash S :^{\mathrm{cns}} \tau_0}{\bullet \vdash F :: S :^{\mathrm{cns}} \tau}\ \textsc{Frame}$$

The only rule for the typing judgment $\vdash M$ of machine states is Execution, which requires that the producer in focus is closed and well-typed, and the stack is a well-typed consumer of the same type.

$$\frac{\Theta \mid \bullet \vdash p : \tau \qquad \Theta \vdash S :^{\mathrm{cns}} \tau \qquad \vdash \Theta\ \textsc{Ok}}{\vdash \langle p \parallel S \parallel \Theta \rangle}\ \textsc{Execution}$$

The evaluation steps are presented in Definition 3.4. As the program never changes in any way, we leave it implicit in the rules for better readability.

*Definition 3.4 (Stepping Relation for **Fun**).*

(arg)	$\langle A[p] \parallel S \rangle$	$\to$	$\langle p \parallel A :: S \rangle$	if $p \notin a$
(force)	$\langle p.D(v) \parallel S \rangle$	$\to$	$\langle p \parallel \square.D(v) :: S \rangle$	if $p \notin v$
(ret)	$\langle v \parallel F :: S \rangle$	$\to$	$\langle F[v] \parallel S \rangle$	
(let)	$\langle \textbf{let } x = a;\ p \parallel S \rangle$	$\to$	$\langle p[x \mapsto a] \parallel S \rangle$	
(call)	$\langle f(v) \parallel S \rangle$	$\to$	$\langle p[\Gamma \mapsto v] \parallel S \rangle$	
	where $\Theta(f) = \textbf{def } f(\Gamma) : \tau \{ p \}$			
(add)	$\langle n_1 + n_2 \parallel S \rangle$	$\to$	$\langle n \parallel S \rangle$	
	where $n \equiv n_1 + n_2$			
(ifz)	$\langle \textbf{if } n \equiv 0 \{ p_1 \} \textbf{ else } \{ p_2 \} \parallel S \rangle$	$\to$	$\langle p_1 \parallel S \rangle$	if $n = 0$
			$\langle p_2 \parallel S \rangle$	otherwise
(case)	$\langle K(v).\textbf{case } \{ \dots, K(\Gamma) \Rightarrow p, \dots \} \parallel S \rangle$	$\to$	$\langle p[\Gamma \mapsto v] \parallel S \rangle$	
(dtor)	$\langle \textbf{new } \{ \dots, D(\Gamma) \Rightarrow p, \dots \}.D(v) \parallel S \rangle$	$\to$	$\langle p[\Gamma \mapsto v] \parallel S \rangle$	
(label)	$\langle \textbf{label } \alpha \{ p \} \parallel S \rangle$	$\to$	$\langle p[\alpha \mapsto S] \parallel S \rangle$	
(goto)	$\langle \textbf{goto } S\ (p) \parallel S' \rangle$	$\to$	$\langle p \parallel S \rangle$	
(exit)	$\langle \textbf{exit } p \parallel S \rangle$	$\to$	$\langle p \parallel \textbf{exit} \rangle$	

Rule *arg* pushes a by-value frame A onto the stack if the producer p in its hole is not an argument value. Rule *force* pushes a frame $\square.D(v)$ onto the stack if the producer p in its hole is not a value. Only rule *force* forces producers of codata type, while rule *arg* ignores them, implementing the call-by-name strategy for them. Note that a destructor only becomes a forcing frame once all of its arguments are argument values. Rule *ret* pops an evaluation frame F off the stack if the producer v in focus is a value. In rule *let*, we substitute the bound producer a into the remaining one p if the bound producer is an argument value. Similarly, if all arguments v in a call of a top-level function f are argument values (rule *call*), we substitute these into the body p of the called function. We write $[x \mapsto a]$ for standard capture-avoiding substitution of term a for variable x, and similarly for multiple (co)variables and terms. Rules *add* and *ifz* are as expected. In rule *case* for pattern matches, we substitute the argument values v of the constructor K into the body p of the matching branch. Rule *dtor* for copattern matches is exactly dual. The behavior of the control operators **label** and **goto** is defined by rules *label* and *goto*, respectively. In rule *label*, we capture the current stack and substitute it for the covariable α in the body p of the **label** expression. In rule *goto*, we reinstall a captured stack S, replacing the current stack S'. Finally, rule *exit* replaces the current stack with an **exit** frame for the evaluation of the argument p, because an **exit** expression never returns to the current stack.

Type safety. **Fun** satisfies the usual theorems of progress (Theorem 3.5) and preservation (Theorem 3.6). The proof is a straightforward case analysis in both cases.

Theorem 3.5 (Progress).
If $\vdash M$, then either M is of the form $\langle n \parallel \textbf{exit} \parallel \Theta \rangle$ or there exists a unique machine state M' such that $M \to M'$.

Theorem 3.6 (Preservation).
If $\vdash M$ and $M \to M'$, then $\vdash M'$.

4 The High-Level Intermediate Language Core

The intermediate language **Core** introduced in this section is directly based on the sequent calculus. The skeleton of **Core** is formed by the $\lambda\mu\tilde{\mu}$-calculus of Curien and Herbelin [2000], but extended with arbitrary data and codata types as in System $\mathcal{CD}$ of Downen and Ariola [2020]. As before, we present the syntax, the typing rules, and the operational semantics.

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 Definition 4.1 in Figure 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 producer bindings with prd, dually to how consumer bindings are annotated with cns. We 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. 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 said to be positive and codata types are negative [Andreoli 1992; Downen 2017; Downen and Ariola 2020, 2021; Girard 1991, 1993].

The only non-terminals not defined in Definition 4.1 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 Definition 4.2 in Figure 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 $\mathbf{let}\, v = X(\sigma);\, s$. Moreover, there is an additional construct **substitute** to be explained in Section 7.

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 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

Definition 4.1 (Syntax of Types, Contexts, Declarations and Programs).

$$
\begin{array}{lcll}
\tau & ::= & \mathbf{i64} \mid T & \textit{Types} \\
\chi & ::= & \mathrm{prd} \mid \mathrm{cns} & \textit{Chirality} \\
\Gamma & ::= & \bullet \mid \Gamma, v :^{\chi} \tau & \textit{Typing Contexts} \\
\pi & ::= & \mathbf{data} \mid \mathbf{codata} & \textit{Polarity} \\
\delta & ::= & \pi\, T\, \{\, X(\Gamma), \ldots \,\} \mid \mathbf{def}\ f(\Gamma)\, \{\, s \,\} & \textit{Declarations} \\
\Theta & ::= & \bullet \mid \Theta, \delta & \textit{Programs}
\end{array}
$$

Definition 4.2 (Syntax of Terms).

Core

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

Focused Core (*Section 6.1*)

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

Shrunk Core (*Section 6.2*)

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

AxCut (*Section 7*)

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

Fig. 3. Syntax of intermediate languages.

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 3.2.

*Example 4.3 (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.

$$
\begin{array}{l}
\textbf{data}\ \mathtt{List}[A]\,\{\,\mathtt{Nil},\ \mathtt{Cons}(x :^{\mathrm{prd}} A, xs :^{\mathrm{prd}} \mathtt{List}[A])\,\} \\
\textbf{codata}\ \mathtt{Fun}[A, B]\,\{\,\mathtt{apply}(x :^{\mathrm{prd}} A, \beta :^{\mathrm{cns}} B)\,\} \\
\textbf{def}\ \mathtt{map}(xs :^{\mathrm{prd}} \mathtt{List}[\mathbf{i64}], f :^{\mathrm{prd}} \mathtt{Fun}[\mathbf{i64}, \mathbf{i64}], \alpha :^{\mathrm{cns}} \mathtt{List}[\mathbf{i64}])\,\{ \\
\quad \langle xs \mid \textbf{case}\,\{\,\mathtt{Nil} \Rightarrow \langle \mathtt{Nil} \mid \alpha \rangle, \\
\qquad\qquad \mathtt{Cons}(y, ys) \Rightarrow \langle \mathtt{Cons}(\mu\beta.\langle f \mid \mathtt{apply}(y, \beta)\rangle, \mu\gamma.\mathtt{map}(ys, f, \gamma)) \mid \alpha \rangle\,\} \rangle \\
\} \\
\textbf{def}\ \mathtt{increment}(xs :^{\mathrm{prd}} \mathtt{List}[\mathbf{i64}], \alpha :^{\mathrm{cns}} \mathtt{List}[\mathbf{i64}])\,\{ \\
\quad \mathtt{map}(xs, \textbf{new}\,\{\,\mathtt{apply}(x, \beta) \Rightarrow \langle x + 1 \mid \beta \rangle\,\}, \alpha) \\
\}
\end{array}
$$

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 5. The lifting of computations out of argument position is a separate step, which we will discuss in Section 6.1. 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.

4.1 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.

As before, there are a number of judgment forms used to type expressions in **Core**, whose rules are presented in Figure 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.

As the rules for arguments are virtually identical to the ones for **Fun** in Figure 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

Well-Formed Declarations $\Theta \vdash \delta$ Ok

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

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

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

The rules Var, Ctor, Lit and Plus are identical to the ones in Figure 2.

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

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

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

The rule Covar is identical to the one in Figure 2.

Statement Typing $\Theta \mid \Gamma \vdash s$

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

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

Fig. 4. Typing rules for **Core**.

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.

4.2 Operational Semantics

We define the operational semantics of **Core** again in terms of an abstract machine. However, in contrast to **Fun**, we do not need stacks, because their role is now played by the first-class consumers already present in **Core**, and the reduction steps are defined on statements. In particular, we have $\tilde{\mu}$-bindings as consumers that allow us to bind arbitrary producers to a variable, and dually, we have μ-bindings as producers that allow us to bind arbitrary consumers to a covariable. This also raises the question of how to reduce cuts of these two constructs, the famous *critical pairs* [Curien and Herbelin 2000]:

$$\begin{array}{rcl} \langle \mu\alpha.s_1 \mid \tilde{\mu}x.s_2 \rangle & \to & s_1[\alpha \mapsto \tilde{\mu}x.s_2] \\ & \to & s_2[x \mapsto \mu\alpha.s_1] \end{array}$$

Since s_1 and s_2 could be completely unrelated, having both rules generally renders a system non-confluent. This is the *fundamental dilemma of computation*.

A related question is how to deal with computations in argument position, such as in $f(\mu\alpha.s)$, where f is a top-level function. If $\mu\alpha.s$ is of a codata type, which use call-by-name, we do not want to evaluate it. Otherwise, we use call-by-value, so we do want to evaluate it. To facilitate this, we bind the result of the argument to a fresh variable with a $\tilde{\mu}$-binding:

$$f(\mu\alpha.s) \to \langle \mu\alpha.s \mid \tilde{\mu}x.f(x) \rangle$$

This is *focusing*, which originates in proof search [Andreoli 1992]. A sequence of focusing steps like the one illustrated above brings terms into a focused form[3]. Focusing can be performed statically before evaluation [Curien and Herbelin 2000]. For reasoning about programs, however, having dynamic focusing steps [Wadler 2003] during evaluation can be beneficial, as they enrich the equational theory. Both approaches are semantically equivalent [Downen and Ariola 2018], the only difference being whether the focusing steps are performed before or during evaluation. We will come back to static focusing in Section 6.1.

Either way, the focusing steps often result in critical pairs, as seen in the above example. But when in this form, it is obvious which side of the cut we want to reduce: Reducing the $\tilde{\mu}$-binding would just revert the focusing step, so we have to reduce the μ-binding to make real progress. Hence, for call-by-value evaluation, we reduce the μ-binding in a critical pair. For call-by-name evaluation, however, we would not want to reduce the producer, as it would be a value already, so we reduce the $\tilde{\mu}$-binding in this case. The evaluation of critical pairs determines and is determined by the evaluation strategy. Therefore, since we use call-by-name evaluation for codata types but call-by-value evaluation otherwise, evaluation depends on typing.

To make this precise, we have to define which producers are values that can be substituted for a variable, and which consumers are covalues that can be substituted for a covariable. Moreover, we have to define where focusing steps must be applied. Definition 4.4 presents the syntax for the operational semantics.

[3]The proof search described by Andreoli [1992] has further requirements, such as cut-freeness and η-expansion, but the eponymous component of focusing proof search is about keeping focus on the subformulas of a decomposed formula. The focusing steps in the programming language literature usually only bring connectives into such a focused form.

*Definition 4.4 (Syntax of Operational Semantics of **Core**).*

M	::=	$\langle s \parallel \Theta \rangle$	*Configurations*
A_p	::=	$K(v, \square, \sigma) \mid \square + p \mid n + \square$	*Producer Frames*
A_c	::=	$D(v, \square, \sigma)$	*Consumer Frames*
A	::=	$f(v, \square, \sigma) \mid \textbf{if}\ \square \equiv 0\ \{s\}\ \textbf{else}\ \{s\} \mid \textbf{exit}\ \square \mid \langle A_p \mid q \rangle \mid \langle v \mid A_c \rangle$	*Focusing Frames*
v	::=	$n \mid K(v) \mid \textbf{new}\ \{\ldots\} \mid p \quad \text{where } p : {}^{\text{prd}}\ \textbf{codata}\ T\ \{\ldots\}$	*Values*
q	::=	$D(v) \mid \textbf{case}\ \{\ldots\} \mid c \quad \text{where } \neg(c : {}^{\text{cns}}\ \textbf{codata}\ T\ \{\ldots\})$	*Covalues*
a	::=	$v \mid q$	*Argument Values*
v	::=	$\bullet \mid v, a$	

Machine configurations M consist of a statement in focus and the whole program Θ. The only typing rule for their typing judgment $\vdash M$ simply requires the statement in focus to be closed and well-typed, similar to **Fun**, and we omit its straightforward definition here. Final machine states are of the form $\langle \textbf{exit}\ n \parallel \Theta \rangle$ with the integer n being the exit code of the evaluated program. Program execution is started in the state $\langle s \parallel \Theta \rangle$, where s is the body of the main function. The focusing frames A determine where a focusing step is applied. Their definition makes use of producer frames A_p and consumer frames A_c, which result in a focusing frame when occuring in a cut with a covalue q or a value v, respectively. A focusing frame is typed as a producer if its hole is filled with a consumer and vice versa. The frame is well-typed if it becomes a well-typed statement when filling its hole with a fresh (co)variable, similar to evaluation frames in **Fun**. We again omit the straightforward typing rules. Also similarly to **Fun**, we define argument values a, but in **Core** they are symmetric, consisting of values and covalues. 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. We again write v for a list of argument values.

The evaluation steps are presented in Definition 4.5. As the program never changes in any way, we again leave it implicit, defining the rules directly on statements for better readability.

*Definition 4.5 (Stepping Relation for **Core**).*

$(\tilde{\mu})$	$\langle v \mid \tilde{\mu}x.s \rangle$	$\to$	$s[x \mapsto v]$	
(μ)	$\langle \mu\alpha.s \mid q \rangle$	$\to$	$s[\alpha \mapsto q]$	
(arg-p)	$A[p]$	$\to$	$\langle p \mid \tilde{\mu}x.A[x] \rangle$	if $p \notin v$
(arg-c)	$A[c]$	$\to$	$\langle \mu\alpha.A[\alpha] \mid c \rangle$	if $c \notin q$
(call)	$f(v)$	$\to$	$s[\Gamma \mapsto v]$	
	where $\Theta(f) = \textbf{def}\ f(\Gamma)\ \{s\}$			
(add)	$\langle n_1 + n_2 \mid c \rangle$	$\to$	$\langle n \mid c \rangle$	
	where $n \equiv n_1 + n_2$			
(ifz)	$\textbf{if}\ n \equiv 0\ \{s_1\}\ \textbf{else}\ \{s_2\}$	$\to$	s_1	if $n = 0$
			s_2	otherwise
(case)	$\langle K(v) \mid \textbf{case}\ \{\ldots, K(\Gamma) \Rightarrow s, \ldots\} \rangle$	$\to$	$s[\Gamma \mapsto v]$	
(dtor)	$\langle \textbf{new}\ \{\ldots, D(\Gamma) \Rightarrow p_0, \ldots\} \mid D(v) \rangle$	$\to$	$s[\Gamma \mapsto v]$	

The dual rules $\tilde{\mu}$ and μ define reduction for $\tilde{\mu}$- and μ-bindings, respectively. The opposite side of the cut must be a value or covalue, respectively, for these rules to fire. This ensures that the rules are deterministic: For each type, either $\tilde{\mu}$-bindings are covalues or μ-bindings are values, but never

both. In either case, we substitute the (co)value for the (co)variable in the body of the binding. The focusing rules *arg-p* and *arg-c* are also dual to each other. If the hole of a focusing frame A is filled with a non-value producer p, the latter is lifted and cut with a $\tilde{\mu}$-binding whose variable fills the hole of A. For non-covalue consumers, we use a μ-binding instead. The focusing rules are typically denoted with ς, but we have chosen names that indicate the correspondence with the rule *arg* in **Fun** instead. In either case, we implicitly assume the (co)variable of the binding to be fresh. All other rules are essentially the same as the corresponding rules in **Fun**, the main difference being that they operate on statements instead of producers.

Type safety. **Core** satisfies the usual theorems of progress (Theorem 4.6) and preservation (Theorem 4.7). The proof is a straightforward case analysis in both cases.

Theorem 4.6 (Progress).
If $\vdash M$, then either M is of the form $\langle \textbf{\textit{exit}}\, n \parallel \Theta \rangle$ or there exists a unique machine state M' such that $M \to M'$.

Theorem 4.7 (Preservation).
If $\vdash M$ and $M \to M'$, then $\vdash M'$.

4.2.1 Equational theory. We can turn the rewriting rules in Definition 4.5 into an equational theory by taking their compatible-reflexive-transitive-symmetric closure, as usual. That is, we apply rules frontwards and backwards, arbitrarily many times, and in every program context. As rules can be applied to open terms, we add (co)variables as (co)values.

$$\begin{array}{lcll} v & ::= & \ldots \mid x & \textit{Values} \\ q & ::= & \ldots \mid \alpha & \textit{Covalues} \end{array}$$

We can make this equational theory even richer by also adding η-laws. There are two kinds of such η-laws. First, we add the follwing laws for trivial μ- and $\tilde{\mu}$-abstractions:

$$\begin{array}{lcll} (\eta_\mu) & \mu\alpha.\langle p \mid \alpha \rangle & \to & p \\ (\eta_{\tilde{\mu}}) & \tilde{\mu}x.\langle x \mid c \rangle & \to & c \end{array}$$

As usual, we (implicitly) assume that α and x do not occur free in p and c, respectively. These laws are valid for all producers and consumers, independently of whether they are (co)values or not [Downen and Ariola 2018]. They allow us, for example, to prove more local focusing laws, such as:

$$\begin{aligned} K(v, p, \sigma) &=_{\eta_\mu} \mu\alpha.\langle K(v, p, \sigma) \mid \alpha \rangle \\ &=_{arg\text{-}p} \mu\alpha.\langle p \mid \tilde{\mu}x.\langle K(v, x, \sigma) \mid \alpha \rangle \rangle \end{aligned}$$

The second kind of η-laws we add are for data and codata types:

$$\begin{array}{lll} (\eta_D) & \textbf{case}\, \{ K_1(\Gamma_1) \Rightarrow \langle K_1(\Gamma_1) \mid q \rangle, \ldots \} \to q \\ & \quad \text{where } q :^{\text{cns}} \textbf{data}\, T \{ K_1(\Gamma_1), \ldots \} \\ (\eta_C) & \textbf{new}\, \{ D_1(\Gamma_1) \Rightarrow \langle v \mid D_1(\Gamma_1) \rangle, \ldots \} \to v \\ & \quad \text{where } v :^{\text{prd}} \textbf{codata}\, T \{ D_1(\Gamma_1), \ldots \} \end{array}$$

We again implicitly assume that neither of the (co)variables in the Γ_i occur in q and v, respectively. These laws must be restricted to (co)values to be valid [Downen 2017; Downen and Ariola 2020]. However, since we use call-by-value evaluation for data types and call-by-name evaluation for codata types, all consumers of data types are covalues and all producers of codata types are values. Hence, we can freely apply these η-laws.

We write $=_R$ to mean equality with respect to the equational theory described here. If we want to indicate the rule justifying an equation, we write $=_r$ with r the name of the rule.

5 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 5.3) and semantics (Theorem 5.5).

5.1 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.

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

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 that terminates the program with the value it receives. We use the consumer $\tilde{\mu}x.\mathbf{exit}\ x$ for this purpose, which binds the value it is cut with to the variable x and then exits the program with this value as its exit code.

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

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.

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

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

The translation for terms is shown in Figure 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 5.1 (Translation of Let Bindings). A naive translation might introduce too many μ-abstractions, such as in

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

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

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

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 6.1.

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. Section 6.1), 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 Section 6.1, 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 5.1. We hence choose to not lift these terms out of argument positions to keep the translation simpler.

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 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. To ensure that we can safely inline it, 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. Section 4.2). This is accomplished by the mutually recursive

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

$$\begin{array}{rcl@{\qquad}rcl}
\mathcal{C}[\![n]\!] & \coloneqq & n & \mathcal{C}[\![p_1 + p_2]\!] & \coloneqq & \mathcal{C}[\![p_1]\!] + \mathcal{C}[\![p_2]\!] \\
\mathcal{C}[\![x]\!] & \coloneqq & x & \mathcal{C}[\![\mathbf{new}\,\{\,D_1(\Gamma_1) \Rightarrow p_1, \ldots\,\}]\!] & \coloneqq & \mathbf{new}\,\{\,D_1(\Gamma_1, \alpha_1) \Rightarrow \mathcal{C}[\![p_1]\!]_{\alpha_1}, \ldots\,\} \\
\mathcal{C}[\![K(\sigma)]\!] & \coloneqq & K(\mathcal{C}[\![\sigma]\!]) & \mathcal{C}[\![\mathbf{label}\,\alpha\,\{\,p\,\}]\!] & \coloneqq & \mu\alpha.\mathcal{C}[\![p]\!]_\alpha
\end{array}$$

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

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

$$\begin{array}{rcl@{\qquad}rcl}
\mathcal{C}[\![n]\!]_c & \coloneqq & \langle n \mid c \rangle & \mathcal{C}[\![p_1 + p_2]\!]_c & \coloneqq & \langle \mathcal{C}[\![p_1]\!] + \mathcal{C}[\![p_2]\!] \mid c \rangle \\
\mathcal{C}[\![x]\!]_c & \coloneqq & \langle x \mid c \rangle & & & \\
\mathcal{C}[\![f(\sigma)]\!]_c & \coloneqq & f(\mathcal{C}[\![\sigma]\!], c) & \mathcal{C}[\![\mathbf{exit}\,p]\!]_c & \coloneqq & \mathbf{exit}\,\mathcal{C}[\![p]\!] \\
\mathcal{C}[\![\mathbf{label}\,\alpha\,\{\,p\,\}]\!]_c & \coloneqq & \langle \mu\alpha.\mathcal{C}[\![p]\!]_\alpha \mid c \rangle & \mathcal{C}[\![\mathbf{goto}\,\alpha\,(p)]\!]_c & \coloneqq & \mathcal{C}[\![p]\!]_\alpha \\
\mathcal{C}[\![\mathbf{let}\,x = p_1;\; p_2]\!]_c & \coloneqq & \begin{cases} \langle \mathcal{C}[\![p_1]\!] \mid \tilde{\mu}x.\mathcal{C}[\![p_2]\!]_c \rangle & \text{if } p_1 : \mathbf{codata}\,T\,\{\ldots\} \\ \mathcal{C}[\![p_1]\!]_{\tilde{\mu}x.\mathcal{C}[\![p_2]\!]_c} & \text{otherwise} \end{cases} & & \\
\mathcal{C}[\![K(\sigma)]\!]_c & \coloneqq & \langle K(\mathcal{C}[\![\sigma]\!]) \mid c \rangle & \mathcal{C}[\![p.D(\sigma)]\!]_c & \coloneqq & \mathcal{L}_v(\mathcal{C}[\![\sigma]\!])[\lambda as.\mathcal{C}[\![p]\!]_{D(as,c)}]
\end{array}$$

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

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

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

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

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

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

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

Fig. 5. Translation from **Fun** to **Core**.

functions $\mathcal{B}_v(\cdot)[\cdot]$, and $\mathcal{L}_v(\cdot)[\cdot]$[4], 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

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

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. This is illustrated in the following example:

Example 5.2 (Translation of Destructor Invocations). Consider the codata type of binary functions on integers **codata** Fun2 { ap2(x : **i64**, y : **i64**) : **i64** } and a function f that is applied to one value argument 3 and one non-value argument $2 + 2$. The translation of this destructor invocation proceeds as follows:

$$
\begin{aligned}
&\mathcal{C}[\![f.\mathsf{ap2}(3, 2+2)]\!]_c \\
&= \mathcal{L}_v(\mathcal{C}[\![3, 2+2]\!])[\lambda as.\mathcal{C}[\![f]\!]_{\mathsf{ap2}(as,c)}] \\
&= \mathcal{L}_v(3, \mu\alpha.\langle 2+2 \mid \alpha\rangle)[\lambda as.\mathcal{C}[\![f]\!]_{\mathsf{ap2}(as,c)}] \\
&= \mathcal{B}_v(3)[\lambda a_1.\mathcal{L}_v(\mu\alpha.\langle 2+2 \mid \alpha\rangle)[\lambda a_2.(\lambda as.\mathcal{C}[\![f]\!]_{\mathsf{ap2}(as,c)})(a_1, a_2)]] \\
&= \mathcal{B}_v(3)[\lambda a_1.\mathcal{L}_v(\mu\alpha.\langle 2+2 \mid \alpha\rangle)[\lambda a_2.\mathcal{C}[\![f]\!]_{\mathsf{ap2}(a_1,a_2,c)}]] \\
&= \mathcal{L}_v(\mu\alpha.\langle 2+2 \mid \alpha\rangle)[\lambda a_2.\mathcal{C}[\![f]\!]_{\mathsf{ap2}(3,a_2,c)}] \\
&= \mathcal{B}_v(\mu\alpha.\langle 2+2 \mid \alpha\rangle)[\lambda a.(\lambda a_2.\mathcal{C}[\![f]\!]_{\mathsf{ap2}(3,a_2,c)})a] \\
&= \mathcal{B}_v(\mu\alpha.\langle 2+2 \mid \alpha\rangle)[\lambda a.\mathcal{C}[\![f]\!]_{\mathsf{ap2}(3,a,c)}] \\
&= \langle \mu\alpha.\langle 2+2 \mid \alpha\rangle \mid \tilde{\mu}x.(\lambda a.\mathcal{C}[\![f]\!]_{\mathsf{ap2}(3,a,c)})x\rangle \\
&= \langle \mu\alpha.\langle 2+2 \mid \alpha\rangle \mid \tilde{\mu}x.\mathcal{C}[\![f]\!]_{\mathsf{ap2}(3,x,c)}\rangle
\end{aligned}
$$

The value argument 3 has not been lifted out of the destructor $\mathsf{ap2}(3, x, c)$, but the non-value argument $\mu\alpha.\langle 2 + 2 \mid \alpha\rangle$ has been lifted out.

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 [Cong et al. 2019; Kennedy 2007]. 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 (see Theorem 5.4).

5.2 Properties

The above translation preserves typeability and semantics. We start with typeability.

Theorem 5.3 (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**.*
If $\Theta \vdash \mathbf{def}\ f(\Gamma) : \tau\ \{\, p \,\}$ Ok, then $\mathcal{C}[\![\Theta]\!] \vdash \mathcal{C}[\![\mathbf{def}\ f(\Gamma) : \tau\ \{\, p \,\}]\!]$ Ok.
If $\Gamma \vdash p : \tau$ and $\Gamma \vdash c :^{cns} \tau$, then $\mathcal{C}[\![\Gamma]\!] \vdash \mathcal{C}[\![p]\!] :^{prd} \tau$ and $\mathcal{C}[\![\Gamma]\!] \vdash \mathcal{C}[\![p]\!]_c$.

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. □

For the semantics, we first show that the introduction of join points is semantically sound. Let $\mathcal{C}'[\![\cdot]\!]_\odot$ be the same as $\mathcal{C}[\![\cdot]\!]_\odot$ but without the introduction of join points, i.e., the version that simply duplicates the consumer input for pattern matches and conditionals. The two versions of the translation are equal with respect to the equational theory of **Core** (cf. Section 4.2.1).

Theorem 5.4 (Correctness for Join Points). *Let p be a producer in **Fun** with $\Gamma \vdash p : \tau$ and c a consumer in **Core** that can occur during the translation. Then $\mathcal{C}[\![p]\!] =_R \mathcal{C}'[\![p]\!]$ and $\mathcal{C}[\![p]\!]_c =_R \mathcal{C}'[\![p]\!]_c$.*

Proof Sketch. It suffices to show that the join points introduced in the translation are equal to the consumer c they wrap. This is immediate, however, since the join point results from η-expanding c to a trivial $\tilde{\mu}$-abstraction $\tilde{\mu}x.\langle x \mid c\rangle$, and then replacing the cut by a call to a top-level function $j(\Gamma)$ that can be performed immediately since all arguments are (co)variables. Both of these steps are justified by the equational theory of **Core** through rules $\eta_{\tilde{\mu}}$ and *call*. □

For the version of the translation without join points, we can now show a simulation theorem: One step in the abstract machine of **Fun** corresponds to 0, 1, or 2 steps in **Core**. To do so, we define the translation of machine states as follows:

$$\begin{aligned} \mathcal{C}'[\![\langle n \,\|\, \mathbf{exit} \,\|\, \Theta\rangle]\!] &:= \langle \mathbf{exit}\, n \,\|\, \mathcal{C}'[\![\Theta]\!]\rangle \\ \mathcal{C}'[\![\langle p \,\|\, S \,\|\, \Theta\rangle]\!] &:= \langle \mathcal{C}'[\![p]\!]_{\mathcal{C}'[\![S]\!]} \,\|\, \mathcal{C}'[\![\Theta]\!]\rangle \quad \text{if } p \neq n \vee S \neq \mathbf{exit} \end{aligned}$$

That is, final states are translated to final states, and otherwise we translate the producer in focus with the translation of the stack as consumer input. The definition of the translation of stacks is given in Appendix B. We also slightly generalize the translation of **goto** to account for stacks:

$$\mathcal{C}'[\![\mathbf{goto}\ S\ (p)]\!]_c := \mathcal{C}'[\![p]\!]_{\mathcal{C}'[\![S]\!]}$$

Now we can state the simulation theorem.

Theorem 5.5 (Simulation). *Let M, M' be machine states in **Fun** with $\vdash M$ and $M \to M'$, then $\mathcal{C}'[\![M]\!] \to^{[0-2]} \mathcal{C}'[\![M']\!]$, where $\to^{[0-2]}$ means 0, 1, or 2 reduction steps.*

Proof Sketch. The proof is by cases on the reduction step, using that the translation preserves values and translates stacks to covalues. More details are given in Appendix B. □

6 Transformations on Core

We now start to bring programs in the language **Core** into a normal form that is suitable for compilation to machine code. 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 (Theorem 6.2, Theorem 6.5) and semantics (Theorem 6.3, Theorem 6.6).

6.1 Focusing Transformation

The focusing transformation $\mathcal{F}(\cdot)$, which lifts out and names terms in argument position, is shown in Figure 6. Focusing is related to transformations to A-normal form [Binder and Piecha 2022; Flanagan et al. 1993] or continuation-passing style, which similarly lift out and name subterms. The focusing transformation we present is an extension of the usual static focusing transformation [Andreoli 1992; Curien and Herbelin 2000], which 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 Definition 4.2, 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 4.3.

Example 6.1 (Focusing of Mapping a Function over a List). There are two parts in the program in Example 4.3 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

$$\begin{aligned}&\mathcal{F}(\langle \mathtt{Cons}(\mu\beta.\langle f \mid \mathtt{apply}(y,\beta)\rangle, \mu\gamma.\mathtt{map}(ys,f,\gamma)) \mid \alpha\rangle)\\ &= \langle \mu\beta.\langle f \mid \mathtt{apply}(y,\beta)\rangle \mid \tilde{\mu}h.\langle \mu\gamma.\mathtt{map}(ys,f,\gamma) \mid \tilde{\mu}t.\langle \mathtt{Cons}(h,t) \mid \alpha\rangle\rangle\rangle\end{aligned}$$

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 constructor `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.

$$\begin{aligned}&\mathcal{F}(\mathtt{map}(xs, \mathbf{new}\,\{\,\mathtt{apply}(x,\beta) \Rightarrow \langle x+1 \mid \beta\rangle\,\}, \alpha))\\ &= \langle \mathbf{new}\,\{\,\mathtt{apply}(x,\beta) \Rightarrow \langle 1 \mid \tilde{\mu}y.\langle x+y \mid \beta\rangle\rangle\,\} \mid \tilde{\mu}f.\mathtt{map}(xs,f,\alpha)\rangle\end{aligned}$$

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 $\mathcal{L}(\cdot)[\cdot]$[5], 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 Section 5.1, 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.

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

$$\begin{array}{rcl}
\multicolumn{3}{c}{\mathcal{F}(\cdot) : \mathit{Definition}_{\textbf{Core}} \to \mathit{Definition}_{\textbf{Focused Core}}} \\
\mathcal{F}(\textbf{def}\ f(\Gamma)\{s\}) & \coloneqq & \textbf{def}\ f(\Gamma)\{\mathcal{F}(s)\}
\end{array}$$

$$\begin{array}{rcl}
\multicolumn{3}{c}{\mathcal{F}(\cdot) : \mathit{Statement}_{\textbf{Core}} \to \mathit{Statement}_{\textbf{Focused Core}}} \\
\mathcal{F}(\langle p_1 + p_2 \mid c\rangle) & \coloneqq & \mathcal{B}(p_1)[\lambda a_1.\mathcal{B}(p_2)[\lambda a_2.\langle a_1 + a_2 \mid \mathcal{F}(c)\rangle]] \\
\mathcal{F}(\langle K(\sigma) \mid c\rangle) & \coloneqq & \mathcal{L}(\sigma)[\lambda as.\langle K(as) \mid \mathcal{F}(c)\rangle] \\
\mathcal{F}(\langle p \mid D(\sigma)\rangle) & \coloneqq & \mathcal{L}(\sigma)[\lambda as.\langle \mathcal{F}(p) \mid D(as)\rangle] \\
\mathcal{F}(\langle p \mid c\rangle) & \coloneqq & \langle \mathcal{F}(p) \mid \mathcal{F}(c)\rangle \quad \text{(for all other cuts)} \\
\mathcal{F}(\textbf{if}\ p \equiv 0\{s_1\}\ \textbf{else}\{s_2\}) & \coloneqq & \mathcal{B}(p)[\lambda a.\textbf{if}\ a \equiv 0\{\mathcal{F}(s_1)\}\ \textbf{else}\{\mathcal{F}(s_2)\}] \\
\mathcal{F}(f(\sigma)) & \coloneqq & \mathcal{L}(\sigma)[\lambda as.f(as)] \\
\mathcal{F}(\textbf{exit}\ p) & \coloneqq & \mathcal{B}(p)[\lambda a.\textbf{exit}\ a]
\end{array}$$

$$\begin{array}{rcl}
\multicolumn{3}{c}{\mathcal{F}(\cdot) : \mathit{Producer}_{\textbf{Core}} \to \mathit{Producer}_{\textbf{Focused Core}}} \\
\mathcal{F}(x) & \coloneqq & x \\
\mathcal{F}(\mu\alpha.s) & \coloneqq & \mu\alpha.\mathcal{F}(s) \\
\mathcal{F}(\textbf{new}\{D_1(\Gamma_1) \Rightarrow s_1, \ldots\}) & \coloneqq & \textbf{new}\{D_1(\Gamma_1) \Rightarrow \mathcal{F}(s_1), \ldots\} \\
\mathcal{F}(K(\sigma)) & & \text{does not occur} \\
\mathcal{F}(n) & \coloneqq & n \\
\mathcal{F}(p_1 + p_2) & & \text{does not occur}
\end{array}$$

$$\begin{array}{rcl}
\multicolumn{3}{c}{\mathcal{F}(\cdot) : \mathit{Consumer}_{\textbf{Core}} \to \mathit{Consumer}_{\textbf{Focused Core}}} \\
\mathcal{F}(\alpha) & \coloneqq & \alpha \\
\mathcal{F}(\tilde{\mu}x.s) & \coloneqq & \tilde{\mu}x.\mathcal{F}(s) \\
\mathcal{F}(\textbf{case}\{K_1(\Gamma_1) \Rightarrow s_1, \ldots\}) & \coloneqq & \textbf{case}\{K_1(\Gamma_1) \Rightarrow \mathcal{F}(s_1), \ldots\} \\
\mathcal{F}(D(\sigma)) & & \text{does not occur}
\end{array}$$

$$\begin{array}{rcl}
\multicolumn{3}{c}{\mathcal{B}(\cdot)[\cdot] : \mathit{Producer}_{\textbf{Core}} \times (\mathit{Variable} \to \mathit{Statement}_{\textbf{Focused Core}}) \to \mathit{Statement}_{\textbf{Focused Core}}} \\
\mathcal{B}(x)[k] & \coloneqq & k(x) \\
\mathcal{B}(\mu\alpha.s)[k] & \coloneqq & \langle \mu\alpha.\mathcal{F}(s) \mid \tilde{\mu}x.k(x)\rangle \\
\mathcal{B}(K(\sigma))[k] & \coloneqq & \mathcal{L}(\sigma)[\lambda as.\langle K(as) \mid \tilde{\mu}x.k(x)\rangle] \\
\mathcal{B}(\textbf{new}\{D_1(\Gamma_1) \Rightarrow s_1, \ldots\})[k] & \coloneqq & \langle \textbf{new}\{D_1(\Gamma_1) \Rightarrow \mathcal{F}(s_1), \ldots\} \mid \tilde{\mu}x.k(x)\rangle \\
\mathcal{B}(n)[k] & \coloneqq & \langle n \mid \tilde{\mu}x.k(x)\rangle \\
\mathcal{B}(p_1 + p_2)[k] & \coloneqq & \mathcal{B}(p_1)[\lambda a_1.\mathcal{B}(p_2)[\lambda a_2.\langle a_1 + a_2 \mid \tilde{\mu}x.k(x)\rangle]]
\end{array}$$

$$\begin{array}{rcl}
\multicolumn{3}{c}{\mathcal{B}(\cdot)[\cdot] : \mathit{Consumer}_{\textbf{Core}} \times (\mathit{Covariable} \to \mathit{Statement}_{\textbf{Focused Core}}) \to \mathit{Statement}_{\textbf{Focused Core}}} \\
\mathcal{B}(\alpha)[k] & \coloneqq & k(\alpha) \\
\mathcal{B}(\tilde{\mu}x.s)[k] & \coloneqq & \langle \mu\alpha.k(\alpha) \mid \tilde{\mu}x.\mathcal{F}(s)\rangle \\
\mathcal{B}(D(\sigma))[k] & \coloneqq & \mathcal{L}(\sigma)[\lambda as.\langle \mu\alpha.k(\alpha) \mid D(as)\rangle] \\
\mathcal{B}(\textbf{case}\{K_1(\Gamma_1) \Rightarrow s_1, \ldots\})[k] & \coloneqq & \langle \mu\alpha.k(\alpha) \mid \textbf{case}\{K_1(\Gamma_1) \Rightarrow \mathcal{F}(s_1), \ldots\}\rangle
\end{array}$$

$$\begin{array}{rcl}
\multicolumn{3}{c}{\mathcal{L}(\cdot)[\cdot] : \mathit{Arguments}_{\textbf{Core}} \times (\mathit{Context} \to \mathit{Statement}_{\textbf{Focused Core}}) \to \mathit{Statement}_{\textbf{Focused Core}}} \\
\mathcal{L}(\bullet)[k] & \coloneqq & k(\bullet) \\
\mathcal{L}(e, \sigma)[k] & \coloneqq & \mathcal{B}(e)[\lambda a.\mathcal{L}(\sigma)[\lambda as.k(a, as)]]
\end{array}$$

Fig. 6. The focusing transformation.

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 and semantics.

Theorem 6.2 (Typeability Preservation).
Let s be a statement in **Core**, *c a consumer, p a producer, Γ a typing context, and τ a type.*
If $\Theta \vdash \textbf{def}\ f(\Gamma)\ \{\, s \,\}$ *Ok, then* $\mathcal{F}(\Theta) \vdash \mathcal{F}(\textbf{def}\ f(\Gamma)\ \{\, s \,\})$ *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. □

Theorem 6.3 (Semantics Preservation).
Let s be a statement in **Core** *with* $\Gamma \vdash s$. *Then* $s =_R \mathcal{F}(s)$.

Proof Sketch. This holds since the focusing transformation consists of repeated applications of the rules *arg-p* and *arg-c* for non-(co)values and rules μ and $\tilde{\mu}$ for (co)values. □

6.2 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 Definition 4.2, 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.

6.2.1 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.

$$
\begin{array}{rcl}
s & ::= & \langle K(\sigma) \mid \tilde{\mu} x.s \rangle \mid \langle \mu\alpha.s \mid D(\sigma) \rangle \mid \langle K(\sigma) \mid \alpha \rangle \mid \langle x \mid D(\sigma) \rangle \\
& \mid & \langle x \mid \mathbf{case}\,\{\, K(\Gamma) \Rightarrow s, \ldots \,\} \rangle \mid \langle \mu\alpha.s \mid \mathbf{case}\,\{\, K(\Gamma) \Rightarrow s, \ldots \,\} \rangle \\
& \mid & \langle \mathbf{new}\,\{\, D(\Gamma) \Rightarrow s, \ldots \,\} \mid \alpha \rangle \mid \langle \mathbf{new}\,\{\, D(\Gamma) \Rightarrow s, \ldots \,\} \mid \tilde{\mu} x.s \rangle \\
& \mid & \langle n \mid \tilde{\mu} x.s \rangle \mid \langle x + x \mid \tilde{\mu} x.s \rangle \\
& \mid & \langle K(\sigma) \mid \mathbf{case}\,\{\, K(\Gamma) \Rightarrow s, \ldots \,\} \rangle \mid \langle \mathbf{new}\,\{\, D(\Gamma) \Rightarrow s, \ldots \,\} \mid D(\sigma) \rangle \\
& \mid & \langle \mu\alpha.s \mid \beta \rangle \mid \langle y \mid \tilde{\mu} x.s \rangle \mid \langle \mu\alpha.s \mid \tilde{\mu} x.s \rangle \mid \langle x \mid \alpha \rangle \mid \langle n \mid \alpha \rangle \mid \langle x + x \mid \alpha \rangle
\end{array}
$$

We will now eliminate the lower eight of these.

6.2.2 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.

$$
\begin{array}{rcl}
\mathcal{S}(\langle \mu\alpha.s \mid \beta \rangle) & := & \mathcal{S}(s[\alpha \mapsto \beta]) \\
\mathcal{S}(\langle y \mid \tilde{\mu} x.s \rangle) & := & \mathcal{S}(s[x \mapsto y]) \\
\mathcal{S}(\langle K_j(\sigma) \mid \mathbf{case}\,\{\, K_1(\Gamma_1) \Rightarrow s_1, \ldots \,\} \rangle) & := & \mathcal{S}(s_j[\Gamma_j \mapsto \sigma]) \\
\mathcal{S}(\langle \mathbf{new}\,\{\, D_1(\Gamma_1) \Rightarrow s_1, \ldots \,\} \mid D_j(\sigma) \rangle) & := & \mathcal{S}(s_j[\Gamma_j \mapsto \sigma])
\end{array}
$$

6.2.3 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. As described in Section 4.2.1, using call-by-value evaluation order for data types and call-by-name evaluation order for codata types 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.

$$
\begin{array}{rcl}
\mathcal{S}(\langle \mu\alpha.s_1 \mid \tilde{\mu} x.s_2 \rangle_T) & := & \langle \mu\alpha.\mathcal{S}(s_1) \mid \mathbf{case}\,\{\, K_1(\Gamma_1) \Rightarrow \langle K_1(\Gamma_1) \mid \tilde{\mu} x.j(\Gamma) \rangle, \ldots \,\} \rangle \\
\text{where} & & \mathbf{data}\, T\,\{\, K_1(\Gamma_1), \ldots \,\} \in \Theta \quad (\Gamma_1, \ldots \text{ fresh}) \\
\text{with} & & \mathbf{def}\; j(\Gamma)\,\{\, \mathcal{S}(s_2) \,\} \\
\text{and} & & \Gamma := \mathsf{freeVars}(\mathcal{S}(s_2)) \\
\mathcal{S}(\langle \mu\alpha.s_1 \mid \tilde{\mu} x.s_2 \rangle_T) & := & \langle \mathbf{new}\,\{\, D_1(\Gamma_1) \Rightarrow \langle \mu\alpha.j(\Gamma) \mid D_1(\Gamma_1) \rangle, \ldots \,\} \mid \tilde{\mu} x.\mathcal{S}(s_2) \rangle \\
\text{where} & & \mathbf{codata}\, T\,\{\, D_1(\Gamma_1), \ldots \,\} \in \Theta \quad (\Gamma_1, \ldots \text{ fresh}) \\
\text{with} & & \mathbf{def}\; j(\Gamma)\,\{\, \mathcal{S}(s_1) \,\} \\
\text{and} & & \Gamma := \mathsf{freeVars}(\mathcal{S}(s_1))
\end{array}
$$

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.

$$\begin{array}{rcl} \mathcal{S}(\langle x \mid \alpha \rangle_T) & \coloneqq & \langle x \mid \mathbf{case}\,\{\, K_1(\Gamma_1) \Rightarrow \langle K_1(\Gamma_1) \mid \alpha \rangle, \ldots \,\} \rangle \\ \text{where} & & \mathbf{data}\, T\,\{\, K_1(\Gamma_1), \ldots \,\} \in \Theta \quad (\Gamma_1, \ldots \text{ fresh}) \\ \mathcal{S}(\langle x \mid \alpha \rangle_T) & \coloneqq & \langle \mathbf{new}\,\{\, D_1(\Gamma_1) \Rightarrow \langle x \mid D_1(\Gamma_1) \rangle, \ldots \,\} \mid \alpha \rangle \\ \text{where} & & \mathbf{codata}\, T\,\{\, D_1(\Gamma_1), \ldots \,\} \in \Theta \quad (\Gamma_1, \ldots \text{ fresh}) \end{array}$$

As an illustration, consider the following example.

Example 6.4. 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.

$$\begin{array}{l} \langle \mu\alpha.\mathsf{range}(n, \alpha) \mid \mathbf{case}\,\{\, \mathtt{Nil} \Rightarrow \langle \mathtt{Nil} \mid \tilde{\mu}x.\mathsf{j}(x) \rangle \\ \qquad\qquad\qquad\qquad \mathtt{Cons}(y, ys) \Rightarrow \langle \mathtt{Cons}(y, ys) \mid \tilde{\mu}x.\mathsf{j}(x) \rangle \,\} \rangle \\ \mathbf{def}\ \mathsf{j}(x :^{\mathrm{prd}} \mathtt{List[i64]})\,\{\, \mathcal{S}(\langle \mu\beta.\mathsf{double}(x, \beta) \mid \tilde{\mu}r.s \rangle) \,\} \end{array}$$

Here we have introduced the join point j to share the body of the expanded consumer, which is now the body of j. 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 j.

6.2.4 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:

$$\mathbf{data}\ \mathtt{Cont}\,\{\, \mathtt{Ret}(x :^{\mathrm{prd}} \mathbf{i64}) \,\}$$

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.

$$\begin{array}{rcll}
\mathcal{S}(\langle \mu\alpha.s_1 \mid \tilde{\mu}x.s_2 \rangle_{\mathbf{i64}}) & \coloneqq & \langle \mu\alpha.\mathcal{S}(s_1) \mid \mathbf{case}\,\{\,\mathtt{Ret}(x) \Rightarrow \mathcal{S}(s_2)\,\} \rangle & \\
\mathcal{S}(\langle x \mid \alpha \rangle_{\mathbf{i64}}) & \coloneqq & \langle \mathtt{Ret}(x) \mid \alpha \rangle & \\
\mathcal{S}(\langle n \mid \alpha \rangle) & \coloneqq & \langle n \mid \tilde{\mu}x.\langle \mathtt{Ret}(x) \mid \alpha \rangle \rangle & (x \text{ fresh}) \\
\mathcal{S}(\langle x_1 + x_2 \mid \alpha \rangle) & \coloneqq & \langle x_1 + x_2 \mid \tilde{\mu}x.\langle \mathtt{Ret}(x) \mid \alpha \rangle \rangle & (x \text{ fresh}) \\
\\
\mathcal{S}(\Gamma, \alpha :^{\mathrm{cns}} \mathbf{i64}) & \coloneqq & \mathcal{S}(\Gamma), \alpha :^{\mathrm{cns}} \mathtt{Cont} & \\
\mathcal{S}(\Gamma, v :^{\chi} \tau) & \coloneqq & \mathcal{S}(\Gamma), v :^{\chi} \tau &
\end{array}$$

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 2.8 in Section 2.

The above transformation again preserves typeability and semantics.

Theorem 6.5 (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 $\mathcal{S}(\Theta) \vdash \mathcal{S}(\mathbf{def}\, f(\Gamma)\,\{\,s\,\})$ 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. □

The most interesting case for semantics preservation is how the transformation deals with built-in types. All other cases are covered by the equational theory of **Core**. More precisely, let $\mathcal{S}'(\cdot)$ be the same as $\mathcal{S}(\cdot)$, except that for cuts at integers it only gives literals and results of arithmetic operations a name:

$$\begin{array}{rcll}
\mathcal{S}'(\langle n \mid \alpha \rangle) & \coloneqq & \langle n \mid \tilde{\mu}x.\langle x \mid \alpha \rangle \rangle & (x \text{ fresh}) \\
\mathcal{S}'(\langle x_1 + x_2 \mid \alpha \rangle) & \coloneqq & \langle x_1 + x_2 \mid \tilde{\mu}x.\langle x \mid \alpha \rangle \rangle & (x \text{ fresh})
\end{array}$$

Theorem 6.6 (Semantics Preservation).
*Let s be a statement in **Focused Core** with $\Gamma \vdash s$. Then $s =_R \mathcal{S}'(s)$.*

Proof Sketch. This holds since all cases of this version of the shrinking transformation are applications of rules of the equational theory. Renaming is either μ or $\tilde{\mu}$, or *case* or *dtor*. For critical pairs and unknown cuts of data and codata types the rules η_D and η_C are used. And giving literals and results of arithmetic operations a name uses rule $\eta_{\tilde{\mu}}$. □

What is left to show is that replacing all $\tilde{\mu}x.s$ of type **i64** by $\mathbf{case}\,\{\,\mathtt{Ret}(x) \Rightarrow s\,\}$ and at the same time replacing all variables y of type **i64** in cuts by $\mathtt{Ret}(y)$ is sound. But note that both $\tilde{\mu}x.s$ and

case $\{\,\mathtt{Ret}(x) \Rightarrow s\,\}$ are covalues, which are bound to a covariable, and variables of type **i64** are only ever cut with such covariables. Thus, it suffices that

$$\langle y \mid \tilde{\mu} x.s \rangle =_{\tilde{\mu}} s[x \mapsto y] =_{case} \langle \mathtt{Ret}(y) \mid \textbf{case}\,\{\,\mathtt{Ret}(x) \Rightarrow s\,\} \rangle$$

7 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 $\textbf{let}\, v = X(\sigma);\ s$, where v is either a variable or a covariable, and X is either the name of a constructor or a destructor (binding the constructor or destructor to a (co)variable), as indicated in Definition 4.2. Similarly, cuts of a constructor or destructor with a (co)variable are represented by **invoke** statements (invoking a method on the object bound to the (co)variable); cuts of (co)pattern matches with (co)variables become **switch** statements (switching on the (co)variable); and (co)pattern matches bound by a μ- or $\tilde{\mu}$-abstraction are turned into **create** statements (creating a new object and binding it to a (co)variable). 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:

$$\begin{array}{rcl} s & ::= & \textbf{let}\, v = X(\sigma);\ s \mid \textbf{invoke}\, v\, X(\sigma) \\ & \mid & \textbf{switch}\, v\, \{\, X(\Gamma) \Rightarrow s, \ldots\,\} \mid \textbf{create}\, v = \Gamma\, \{\, X(\Gamma) \Rightarrow s, \ldots\,\};\ s \\ & \mid & \textbf{lit}\, v \leftarrow n;\ s \mid v \leftarrow v + v;\ s \mid \textbf{if}\, v \equiv 0\, \{\, s\,\}\, \textbf{else}\, \{\, s\,\} \\ & \mid & f(\sigma) \mid \textbf{substitute}[\Gamma \coloneqq \sigma];\ s \mid \textbf{exit}\, v \end{array}$$

There is one new kind of statement that has not appeared so far, namely explicit substitutions [Abadi et al. 1991] $\textbf{substitute}[\Gamma \coloneqq \sigma];\ s$. 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 $\Gamma \coloneqq \sigma$, 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 7.1 and give an operational semantics in Section 7.2. Finally, we present the translation from **Shrunk Core** to **AxCut** in Section 7.3.

7.1 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.

$$\frac{\Gamma \vdash \sigma : \Gamma' \qquad \Gamma' \vdash s}{\Gamma \vdash \mathbf{substitute}[\Gamma' := \sigma];\ s}\ \textsc{Substitute}$$

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.

$$\frac{\Gamma, v :^{\text{prd}} \mathbf{i64} \vdash s}{\Gamma \vdash \mathbf{lit}\, v \leftarrow n;\ s}\ \textsc{Lit} \qquad \frac{v_1 :^{\text{prd}} \mathbf{i64} \in \Gamma \qquad v_2 :^{\text{prd}} \mathbf{i64} \in \Gamma \qquad \Gamma, v :^{\text{prd}} \mathbf{i64} \vdash s}{\Gamma \vdash v \leftarrow v_1 + v_2;\ s}\ \textsc{Plus}$$

$$\frac{v :^{\text{prd}} \mathbf{i64} \in \Gamma}{\Gamma \vdash \mathbf{exit}\, v}\ \textsc{Exit} \qquad \frac{v :^{\text{prd}} \mathbf{i64} \in \Gamma \qquad \Gamma \vdash s_1 \qquad \Gamma \vdash s_2}{\Gamma \vdash \mathbf{if}\ v \equiv 0\ \{\, s_1 \,\}\ \mathbf{else}\ \{\, s_2 \,\}}\ \textsc{IfZ}$$

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.

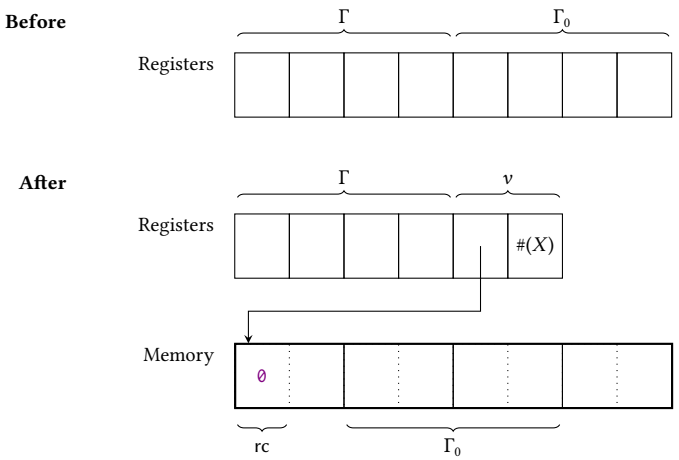
$$\chi_1(\textbf{data}) := \text{prd} \qquad \chi_1(\textbf{codata}) := \text{cns} \qquad \chi_2(\textbf{data}) := \text{cns} \qquad \chi_2(\textbf{codata}) := \text{prd}$$

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. Again, similar to calls of top-level labels, we write $X(\Gamma_0)$ to mean that the constructor or destructor is applied to the variables bound in Γ_0, in that order. This part of the environment is consumed before the new binding is added for the subderivation of s.

$$\frac{\pi\, T\{\dots, X(\Gamma_0)\dots\} \in \Theta \qquad \Gamma, v :^{\chi_1(\pi)} T \vdash s}{\Theta \mid \Gamma, \Gamma_0 \vdash \textbf{let}\, v = X(\Gamma_0);\, s}\ \textsc{Let}\text{-}\pi$$

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 Section 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.

$$\frac{\pi\, T\{X_1(\Gamma_1), \dots\} \in \Theta \qquad \Gamma, v :^{\chi_2(\pi)} T \vdash s \qquad \forall i : \Gamma_i, \Gamma_0 \vdash s_i}{\Theta \mid \Gamma, \Gamma_0 \vdash \mathbf{create}\, v = \Gamma_0\, \{X_1(\Gamma_1) \Rightarrow s_1, \dots\};\, s}\ \textsc{Create}\text{-}\pi$$

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. Notably, we do not perform further closure conversion, but generate code directly for these closures. 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.

$$\frac{\pi\, T\{X_1(\Gamma_1), \dots\} \in \Theta \qquad \forall i : \Gamma, \Gamma_i \vdash s_i}{\Theta \mid \Gamma, v :^{\chi_1(\pi)} T \vdash \mathbf{switch}\, v\, \{X_1(\Gamma_1) \Rightarrow s_1, \dots\}}\ \textsc{Switch}\text{-}\pi$$

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.

$$\frac{\pi\, T\{\dots, X(\Gamma), \dots\} \in \Theta}{\Theta \mid \Gamma, v :^{\chi_2(\pi)} T \vdash \mathbf{invoke}\, v\, X(\Gamma)}\ \textsc{Invoke}\text{-}\pi$$

*Example 7.1 (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 3.2, 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 Operational Semantics

We define the semantics of **AxCut** in terms of an abstract machine, whose syntax is defined in Definition 7.2. The corresponding typing rules are straightforward and are given in Appendix A.

*Definition 7.2 (Syntax of Machine States of **AxCut**).*

$$\begin{array}{rcll} V & ::= & \{\, X;\, E \,\} \mid \{\, E;\, b \,\} \mid n & \textit{Values} \\ E & ::= & x \mapsto V, \ldots & \textit{Environments} \\ M & ::= & \langle s \parallel E \parallel \Theta \rangle & \textit{Configurations} \end{array}$$

To ease presentation, we use the non-terminal b to denote a list of branches,

$$b \quad ::= \quad \{\, X(\Gamma) \Rightarrow s, \ldots \,\}$$

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 $\{\, X;\ E \,\}$ consisting of a name and a field environment, or closures $\{\, E;\ b \,\}$ consisting of a closure environment and branches, or machine integers. In contrast to **Core**, we use environments instead of substitution to make sure that the statement in focus stays within the syntax of **AxCut**. Final machine states are of the form $\langle \mathbf{exit}\ v \parallel E \parallel \Theta \rangle$ with a mapping $v \mapsto n$ in the environment E and the integer n being the exit code of the evaluated program. Program execution is started in the state $\langle s \parallel \bullet \parallel \Theta \rangle$ with an empty environment and the statement s in focus being the body of the main function.

Definition 7.3 defines the evaluation steps of the abstract machine. There is one rule for each kind of statement, which mirrors its typing rule.

*Definition 7.3 (Stepping Relation for **AxCut**).*

(let)	$\langle \mathbf{let}\ v = X(\Gamma_0);\ s \parallel E, E_0 \rangle$	$\to$	$\langle s \parallel E, v \mapsto \{\, X;\ E_0 \,\} \rangle$	where $E_0 : \Gamma_0$
(create)	$\langle \mathbf{create}\ v = \Gamma_0\ b;\ s \parallel E, E_0 \rangle$	$\to$	$\langle s \parallel E, v \mapsto \{\, E_0; b \,\} \rangle$	where $E_0 : \Gamma_0$
(switch)	$\langle \mathbf{switch}\ v\ b \parallel E, v \mapsto \{\, X;\ E_0 \,\} \rangle$	$\to$	$\langle b(X) \parallel E, E_0 \rangle$	
(invoke)	$\langle \mathbf{invoke}\ v\ X(\sigma) \parallel E, v \mapsto \{\, E_0;\ b \,\} \rangle$	$\to$	$\langle b(X) \parallel E, E_0 \rangle$	
(jump)	$\langle f(\sigma) \parallel E \rangle$	$\to$	$\langle \Theta(f) \parallel E \rangle$	
(subst)	$\langle \mathbf{substitute}[v_1' := v_1, \ldots];\ s \parallel E \rangle$	$\to$	$\langle s \parallel v_1' \mapsto E(v_1), \ldots \rangle$	
(lit)	$\langle \mathbf{lit}\ v \leftarrow n;\ s \parallel E \rangle$	$\to$	$\langle s \parallel E, v \mapsto n \rangle$	
(add)	$\langle v \leftarrow v_1 + v_2;\ s \parallel E \rangle$	$\to$	$\langle s \parallel E, v \mapsto E(v_1) + E(v_2) \rangle$	
(ifz)	$\langle \mathbf{if}\ v \equiv 0\ \{\, s_1 \,\}\ \mathbf{else}\ \{\, s_2 \,\} \parallel E \rangle$	$\to$	$\langle s_1 \parallel E \rangle$ $\langle s_2 \parallel E \rangle$	if $E(v) = 0$ otherwise

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 $\{\, X;\ E_0 \,\}$ 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 any machine state $\langle \mathbf{exit} \parallel E \parallel \Theta \rangle$ as 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 7.4) and preservation (Theorem 7.5). 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. The typing judgment $\vdash M$ for machine configurations is defined in Appendix A.

Theorem 7.4 (Progress).
If $\vdash M$, then either M is of the form $\langle \textbf{\textit{exit}}\, v \parallel E \parallel \Theta \rangle$ with $v \mapsto n$ in E or there exists a unique machine state M' such that $M \to M'$.

Theorem 7.5 (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 6 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 7.6. The top-level function consTwice receives an integer and a list and returns the same list with the integer prepended twice.

$$\begin{aligned}&\textbf{def}\ \mathsf{consTwice}(v :^{\mathsf{prd}} \textbf{i64}, l_0 :^{\mathsf{prd}} \mathtt{List}[\textbf{i64}], \alpha :^{\mathsf{cns}} \mathtt{List}[\textbf{i64}])\,\{\\ &\quad \langle \mathsf{Cons}(v, l_0) \mid \tilde{\mu} l_1.\langle \mathsf{Cons}(v, l_1) \mid \alpha \rangle \rangle\\ &\}\end{aligned}$$

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.

$$\begin{aligned}&\textbf{def}\ \mathsf{consTwice}(v :^{\mathsf{prd}} \textbf{i64}, l_0 :^{\mathsf{prd}} \mathtt{List}[\textbf{i64}], \alpha :^{\mathsf{cns}} \mathtt{List}[\textbf{i64}])\,\{\\ &\quad \textbf{substitute}[v := v, \alpha := \alpha, v_0 := v, l_0 := l_0];\\ &\quad \textbf{let}\ l_1 = \mathsf{Cons}(v_0, l_0);\\ &\quad \textbf{substitute}[v := v, l_1 := l_1, \alpha := \alpha];\\ &\quad \textbf{invoke}\ \alpha\ \mathsf{Cons}(v, l_1)\\ &\}\end{aligned}$$

The arguments of the Cons constructors now exactly match the appropriate part of the context.

The translation is presented in Figure 7. 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

$$\mathcal{X}[\![\cdot]\!] : \mathit{Definition}_{\textbf{Shrunk Core}} \to \mathit{Definition}_{\textbf{AxCut}}$$
$$\mathcal{X}[\![\textbf{def}\ f(\Gamma)\,\{\,s\,\}]\!] \coloneqq \textbf{def}\ f(\Gamma)\,\{\,\mathcal{X}[\![s]\!]_\Gamma\,\}$$

$$\mathcal{X}[\![\cdot]\!]_\odot : \mathit{Statement}_{\textbf{Shrunk Core}} \times \mathit{Context}_{\textbf{AxCut}} \to \mathit{Statement}_{\textbf{AxCut}}$$

$$\begin{aligned}
\mathcal{X}[\![\langle K(\Gamma_0) \mid \tilde{\mu}x.s\rangle]\!]_\Gamma &\coloneqq \textbf{substitute}[\Gamma' \coloneqq \Gamma', \Gamma_0^{\mathsf{f}} \coloneqq \Gamma_0];\\
&\quad\ \textbf{let}\ x = K(\Gamma_0^{\mathsf{f}});\ \mathcal{X}[\![s]\!]_{\Gamma',x}\\
&\quad \text{where}\quad \Gamma' = \mathsf{freeVars}(s) \subset \Gamma\\
\mathcal{X}[\![\langle \mu\alpha.s \mid D(\Gamma_0)\rangle]\!]_\Gamma &\coloneqq \textbf{substitute}[\Gamma' \coloneqq \Gamma', \Gamma_0^{\mathsf{f}} \coloneqq \Gamma_0];\\
&\quad\ \textbf{let}\ \alpha = D(\Gamma_0^{\mathsf{f}});\ \mathcal{X}[\![s]\!]_{\Gamma',\alpha}\\
&\quad \text{where}\quad \Gamma' = \mathsf{freeVars}(s) \subset \Gamma\\
\mathcal{X}[\![\langle K(\Gamma_0) \mid \alpha\rangle]\!]_\Gamma &\coloneqq \textbf{substitute}[\Gamma_0^{\mathsf{f}} \coloneqq \Gamma_0, \alpha \coloneqq \alpha];\ \textbf{invoke}\ \alpha\ K(\Gamma_0^{\mathsf{f}})\\
\mathcal{X}[\![\langle x \mid D(\Gamma_0)\rangle]\!]_\Gamma &\coloneqq \textbf{substitute}[\Gamma_0^{\mathsf{f}} \coloneqq \Gamma_0, x \coloneqq x];\ \textbf{invoke}\ x\ D(\Gamma_0^{\mathsf{f}})\\
\mathcal{X}[\![\langle x \mid \textbf{case}\,\{\,K_1(\Gamma_1) \Rightarrow s_1, \ldots\,\}\rangle]\!]_\Gamma &\coloneqq \textbf{substitute}[\Gamma' \coloneqq \Gamma', x^{\mathsf{f}} \coloneqq x];\\
&\quad\ \textbf{switch}\ x^{\mathsf{f}}\,\{\,K_1(\Gamma_1) \Rightarrow \mathcal{X}[\![s_1]\!]_{\Gamma',\Gamma_1}, \ldots\,\}\\
&\quad \text{where}\quad \Gamma' = \textstyle\bigcup_i \mathsf{freeVars}(s_i) \subset \Gamma\\
\mathcal{X}[\![\langle \textbf{new}\,\{\,D_1(\Gamma_1) \Rightarrow s_1, \ldots\,\} \mid \alpha\rangle]\!]_\Gamma &\coloneqq \textbf{substitute}[\Gamma' \coloneqq \Gamma', \alpha^{\mathsf{f}} \coloneqq \alpha];\\
&\quad\ \textbf{switch}\ \alpha^{\mathsf{f}}\,\{\,D_1(\Gamma_1) \Rightarrow \mathcal{X}[\![s_1]\!]_{\Gamma',\Gamma_1}, \ldots\,\}\\
&\quad \text{where}\quad \Gamma' = \textstyle\bigcup_i \mathsf{freeVars}(s_i) \subset \Gamma\\
\mathcal{X}[\![\langle \mu\alpha.s \mid \textbf{case}\,\{\,K_1(\Gamma_1) \Rightarrow s_1, \ldots\,\}\rangle]\!]_\Gamma &\coloneqq \textbf{substitute}[\Gamma'^{\mathsf{f}} \coloneqq \Gamma', \Gamma_0 \coloneqq \Gamma_0];\\
&\quad\ \textbf{create}\ \alpha = \Gamma_0\,\{\,K_1(\Gamma_1) \Rightarrow \mathcal{X}[\![s_1]\!]_{\Gamma_1,\Gamma_0}, \ldots\,\};\\
&\quad\ \mathcal{X}[\![s[\Gamma' \mapsto \Gamma'^{\mathsf{f}}]]\!]_{\Gamma'^{\mathsf{f}},\alpha}\\
&\quad \text{where}\quad \Gamma_0 = \textstyle\bigcup_i \mathsf{freeVars}(s_i) \subset \Gamma \quad \Gamma' = \mathsf{freeVars}(s) \subset \Gamma\\
\mathcal{X}[\![\langle \textbf{new}\,\{\,D_1(\Gamma_1) \Rightarrow s_1, \ldots\,\} \mid \tilde{\mu}x.s\rangle]\!]_\Gamma &\coloneqq \textbf{substitute}[\Gamma'^{\mathsf{f}} \coloneqq \Gamma', \Gamma_0 \coloneqq \Gamma_0];\\
&\quad\ \textbf{create}\ x = \Gamma_0\,\{\,D_1(\Gamma_1) \Rightarrow \mathcal{X}[\![s_1]\!]_{\Gamma_1,\Gamma_0}, \ldots\,\};\\
&\quad\ \mathcal{X}[\![s[\Gamma' \mapsto \Gamma'^{\mathsf{f}}]]\!]_{\Gamma'^{\mathsf{f}},x}\\
&\quad \text{where}\quad \Gamma_0 = \textstyle\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 &\coloneqq \textbf{substitute}[\Gamma' \coloneqq \Gamma'];\ \textbf{lit}\ x \leftarrow n;\ \mathcal{X}[\![s]\!]_{\Gamma',x}\\
&\quad \text{where}\quad \Gamma' = \mathsf{freeVars}(s) \subset \Gamma\\
\mathcal{X}[\![\langle x_1 + x_2 \mid \tilde{\mu}x.s\rangle]\!]_\Gamma &\coloneqq \textbf{substitute}[\Gamma' \coloneqq \Gamma'];\ x \leftarrow x_1 + x_2;\ \mathcal{X}[\![s]\!]_{\Gamma',x}\\
&\quad \text{where}\quad \Gamma' = (\{x_1\} \cup \{x_2\} \cup \mathsf{freeVars}(s)) \subset \Gamma\\
\mathcal{X}[\![\textbf{if}\ x \equiv 0\,\{\,s_1\,\}\ \textbf{else}\,\{\,s_2\,\}]\!]_\Gamma &\coloneqq \textbf{if}\ x \equiv 0\,\{\,\mathcal{X}[\![s_1]\!]_\Gamma\,\}\ \textbf{else}\,\{\,\mathcal{X}[\![s_2]\!]_\Gamma\,\}\\
\mathcal{X}[\![f(\Gamma_0)]\!]_\Gamma &\coloneqq \textbf{substitute}[\Gamma_0^{\mathsf{f}} \coloneqq \Gamma_0];\ f(\Gamma_0^{\mathsf{f}})\\
\mathcal{X}[\![\textbf{exit}\ x]\!]_\Gamma &\coloneqq \textbf{exit}\ x
\end{aligned}$$

Fig. 7. Translation from **Shrunk Core** to **AxCut**.

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).

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 pre-pass. 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 7.7 (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 $\mathcal{X}[\![\Theta]\!] \vdash \mathcal{X}[\![\mathbf{def}\ f(\Gamma)\ \{\, s \,\}]\!]$ 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. Some details are given in Appendix C. □

For semantics preservation, we only give a sketch, refraining from making a more precise statement here. First, note that we could also give an operational semantics for **Shrunk Core** using an abstract machine with environments instead of substitution. It is well-known that these two approaches are equivalent [Biernacka and Danvy 2007]. When also using the unified syntax, this machine would look like the one of **AxCut** with two differences: There would be no rule *subst* for explicit substitutions, and the rules would not consume parts of the environment. A step in the machine of **Shrunk Core** would then be simulated by two steps in the machine of **AxCut**: first a step for the explicit substitution inserted by the translation $\mathcal{X}[\![\cdot]\!]_{\odot}$ that adapts the environment appropriately, and then the step corresponding to the one in **Shrunk Core**.

8 Code Generation

On the one hand, **AxCut** is, at its core, a normal form of classical sequent calculus as derived in previous sections, 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[6]. 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`.

Definition 8.1 (Syntax of RISC-V).

I	::=	l : | `add` r r r | `addi` r r i | `mv` r r | `li` r i | `la` r l	*Instructions*
	|	`j` o | `jr` r o | `beq` r r o | `lw` r i r | `sw` r i r | `ecall`	
r	::=	`x0`, … , `x31`	*Registers*
o	::=	l | i	*Offsets*
l	::=	`l0` | `l1` | …	*Labels*
i	::=	`0` | `1` | …	*Immediates*

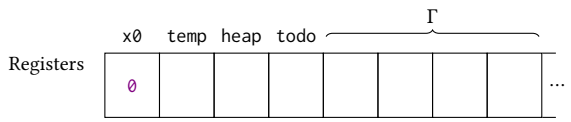
The addition instructions `add` and `addi` add their second and third operand and store the result into the first operand register. The difference is that the third operand is a register for `add` but an immediate for `addi`. Instructions for other arithmetic operations are similar. The `mv` instruction 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

[6]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.

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 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 Definition 7.3. 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.

In the next subsections, we illustrate code generation on example programs, before giving the full formal definition.

8.3 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.

8.3.1 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.

8.3.2 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 $\Gamma = x : {}^{\mathrm{prd}}\ \mathbf{i64}, y : {}^{\mathrm{prd}}\ \mathbf{i64}$, the new environment is $\Gamma' = y : {}^{\mathrm{prd}}\ \mathbf{i64}, x : {}^{\mathrm{prd}}\ \mathbf{i64}$, and the right-hand side in the **substitute** statement is $\sigma = y, x$.

AxCut

$$\frac{x : {}^{\mathrm{prd}}\ \mathbf{i64}, y : {}^{\mathrm{prd}}\ \mathbf{i64} \vdash y, x \ :\ y : {}^{\mathrm{prd}}\ \mathbf{i64}, x : {}^{\mathrm{prd}}\ \mathbf{i64} \qquad y : {}^{\mathrm{prd}}\ \mathbf{i64}, x : {}^{\mathrm{prd}}\ \mathbf{i64} \vdash \ldots}{x : {}^{\mathrm{prd}}\ \mathbf{i64}, y : {}^{\mathrm{prd}}\ \mathbf{i64} \vdash \mathbf{substitute}[y := y, x := x]; \ldots}\ \textsc{Substitute}$$

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 8.5.

8.3.3 Externs are System Calls and Hardware Instructions. Every program runs in an external environment and eventually needs to interact with the outside world in some way, e.g., terminate or output a result. 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

$$\frac{v : {}^{\mathrm{prd}}\ \mathbf{i64} \in \Gamma}{\Gamma \vdash \mathbf{exit}\, v}\ \textsc{Exit}$$

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`.

8.3.4 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.

8.4 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.

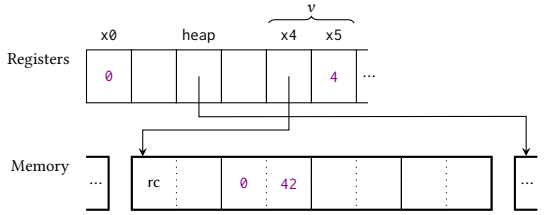
AxCut

```
data Maybe {
  None(), Some(x :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 8.5, 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.

8.4.1 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 $v_0 :^{\text{prd}} \textbf{i64}$ 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 7.1 again).

AxCut

$$\dfrac{\dfrac{\vdash \ldots \qquad x :^{\mathsf{prd}} \mathbf{i64} \vdash \ldots}{v :^{\mathsf{prd}} \mathsf{Maybe} \vdash \mathbf{switch}\, v \{\mathsf{None}() \Rightarrow \ldots, \mathsf{Some}(x) \Rightarrow \ldots\}}\ \textsc{Switch}}{v_0 :^{\mathsf{prd}} \mathbf{i64} \vdash \mathbf{let}\, v = \mathsf{Some}(v_0); \mathbf{switch}\, v \{\mathsf{None}() \Rightarrow \ldots, \mathsf{Some}(x) \Rightarrow \ldots\}}\ \textsc{Let}$$

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 8.5. 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 7.1 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 8.5. The fields are moved into registers, which overwrites the matched variable.

8.4.2 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 3.2. 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 ($v :^{\mathsf{prd}} \mathbf{i64}$) 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

$$\textsc{Create}\ \frac{f : {}^{\text{prd}}\ \mathsf{Fun} \vdash \ldots\ \mathbf{invoke}\ f\ \mathtt{apply}(r, \beta) \qquad x : {}^{\text{prd}}\ \mathbf{i64}, \alpha : {}^{\text{cns}}\ \mathbf{i64}, v : {}^{\text{prd}}\ \mathbf{i64} \vdash \ldots}{v : {}^{\text{prd}}\ \mathbf{i64} \vdash \mathbf{create}\ f = (v : {}^{\text{prd}}\ \mathbf{i64})\{\mathtt{apply}(x, \alpha) \Rightarrow \ldots\};\ \ldots\ \mathbf{invoke}\ f\ \mathtt{apply}(r, \beta)}$$

where the premise is

$$\frac{}{r : {}^{\text{prd}}\ \mathbf{i64}, \beta : {}^{\text{cns}}\ \mathbf{i64}, f : {}^{\text{prd}}\ \mathsf{Fun} \vdash \mathbf{invoke}\ f\ \mathtt{apply}(r, \beta)}\ \textsc{Invoke}\ \ldots$$

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.

8.4.3 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

$$\frac{k :^{\text{cns}} \texttt{Result} \vdash f(k) \qquad x :^{\text{prd}} \mathbf{i64} \vdash \ldots \qquad i :^{\text{prd}} \mathbf{i64} \vdash \ldots}{\vdash \mathbf{create}\, k = (\,)\{\,\texttt{Ok}(x) \Rightarrow \ldots, \texttt{Err}(i) \Rightarrow \ldots\,\};\, f(k)}\ \textsc{Create}$$

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 [Marlow et al. 2007; Peyton Jones 1992]. 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**.

8.4.4 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.

8.5 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.

8.5.1 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 8.6. Alternatively, we could prepare the heap to be a linked list of free blocks at program startup to make just this code work.

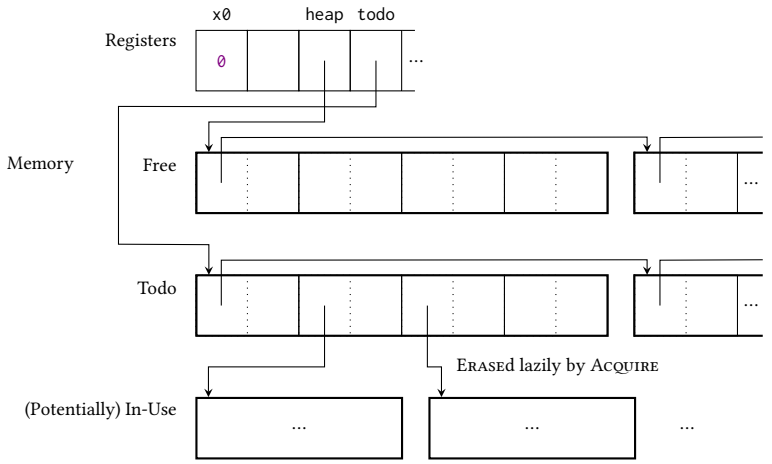
When we load values, we release the block of memory. We store the current head of the free list at offset `0` in the block in the destination register, and move this block into register `heap`.

```
Acquire x4 = mv x4 heap
             lw heap 0 heap
```

```
Release x4 = sw heap 0 x4
             mv heap x4
```

8.5.2 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 8.4, 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 8.6.

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

$$\frac{f :^{\mathsf{cns}} \mathsf{Fun} \vdash f, f : f_1 :^{\mathsf{cns}} \mathsf{Fun}, f_2 :^{\mathsf{cns}} \mathsf{Fun} \qquad f_1 :^{\mathsf{cns}} \mathsf{Fun}, f_2 :^{\mathsf{cns}} \mathsf{Fun} \vdash \ldots}{f :^{\mathsf{cns}} \mathsf{Fun} \vdash \mathbf{substitute}[f_1 := f, f_2 := f]; \ldots}\ \textsc{Substitute}$$

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.

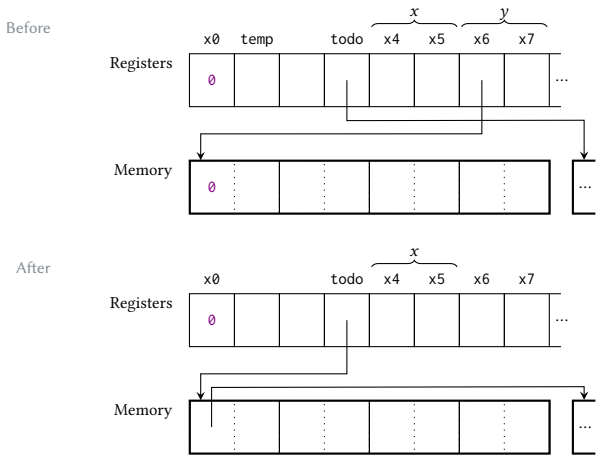
AxCut

$$\frac{x :^{\mathsf{prd}} \mathsf{Result}, y :^{\mathsf{prd}} \mathsf{Maybe} \vdash x : x :^{\mathsf{prd}} \mathsf{Result} \qquad x :^{\mathsf{prd}} \mathsf{Result} \vdash \ldots}{x :^{\mathsf{prd}} \mathsf{Result}, y :^{\mathsf{prd}} \mathsf{Maybe} \vdash \mathbf{substitute}[x := x]; \ldots}\ \textsc{Substitute}$$

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.

$$\mathcal{M}[\![\cdot]\!] : \mathit{Definition}_{\mathbf{AxCut}} \to I^*$$
$$\mathcal{M}[\![\mathbf{def}\ f(\Gamma)\,\{\,s\,\}]\!] \coloneqq f : \mathcal{M}[\![s]\!]$$

$$\begin{array}{rcll}
\mathcal{M}[\![\cdot]\!] : \mathit{Statement}_{\mathbf{AxCut}} \to I^* & & & \\
\mathcal{M}[\![f(\Gamma)]\!] & \coloneqq & \texttt{j}\ f & \\
\mathcal{M}[\![\mathbf{substitute}[\Gamma' \coloneqq \sigma];\ s]\!] & \coloneqq & \textsc{Share}\ [\Gamma' \coloneqq \sigma] & \\
 & & \textsc{Erase}\ [\Gamma' \coloneqq \sigma] & \\
 & & \textsc{Move}\ [\Gamma' \coloneqq \sigma] & \\
 & & \mathcal{M}[\![s]\!] & \\
\mathcal{M}[\![\mathbf{exit}\ v]\!] & \coloneqq & \texttt{li x17 93} & \\
 & & \texttt{mv x10}\ (\textsc{Reg}_2\ v) & \\
 & & \texttt{ecall} & \\
\mathcal{M}[\![\mathbf{lit}\ v \leftarrow n;\ s]\!] & \coloneqq & \texttt{li}\ (\textsc{Reg}_2\ v)\ n & \\
 & & \mathcal{M}[\![s]\!] & \\
\mathcal{M}[\![v \leftarrow v_1 + v_2;\ s]\!] & \coloneqq & \texttt{add}\ (\textsc{Reg}_2\ v)\ (\textsc{Reg}_2\ v_1)\ (\textsc{Reg}_2\ v_2) & \\
 & & \mathcal{M}[\![s]\!] & \\
\mathcal{M}[\![\mathbf{if}\ v \equiv 0\,\{\,s_1\,\}\ \mathbf{else}\,\{\,s_2\,\}]\!] & \coloneqq & \texttt{beq}\ (\textsc{Reg}_2\ v)\ \texttt{x0}\ l & \\
 & & \quad \mathcal{M}[\![s_2]\!] & \\
 & & l : \mathcal{M}[\![s_1]\!] & (l \text{ fresh}) \\
\mathcal{M}[\![\mathbf{let}\ v = X(\Gamma_0);\ s]\!] & \coloneqq & \textsc{Store}\ (\textsc{Reg}_1\ v)\ \Gamma_0 & \\
 & & \texttt{li}\ (\textsc{Reg}_2\ v)\ (\textsc{Index}\ X) & \\
 & & \mathcal{M}[\![s]\!] & \\
\mathcal{M}[\![\mathbf{create}\ v = \Gamma_0\ b;\ s]\!] & \coloneqq & \textsc{Store}\ (\textsc{Reg}_1\ v)\ \Gamma_0 & \\
 & & \texttt{la}\ (\textsc{Reg}_2\ v)\ l & \\
 & & \mathcal{M}[\![s]\!] & \\
 & & l : \textsc{VTable}\ b\ \Gamma_0 & (l \text{ fresh}) \\
\mathcal{M}[\![\mathbf{switch}\ v\ b]\!] & \coloneqq & \texttt{jr}\ (\textsc{Reg}_2\ v)\ l & \\
 & & l : \textsc{JTable}\ b\ \Gamma & (l \text{ fresh}) \\
\mathcal{M}[\![\mathbf{invoke}\ v\ X(\Gamma)]\!] & \coloneqq & \texttt{jr}\ (\textsc{Reg}_2\ v)\ (\textsc{Index}\ X) &
\end{array}$$

Fig. 8. Translation to RISC-V.

8.5.3 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.

8.6 Translation from AxCut to RISC-V

The translation from **AxCut** into RISC-V is defined in Figure 8. It uses auxiliary definitions from Figure 9 and Figure 10. 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.

Auxiliary Definitions

$$
\begin{array}{rcl}
\textsc{Share}\ [\Gamma' \coloneqq \sigma]\ (\Gamma, v :^{\chi} \tau) & \coloneqq & \textbf{if}\ \textsc{NumRefs}\ v\ [\Gamma' \coloneqq \sigma] > 1 \\
 & & \quad \textsc{ShareBlock}\ (\textsc{Reg}_1\ v)\ (\textsc{NumRefs}\ v\ [\Gamma' \coloneqq \sigma] - 1) \\
 & & \textsc{Share}\ [\Gamma' \coloneqq \sigma]\ \Gamma \\
\textsc{Erase}\ [\Gamma' \coloneqq \sigma]\ (\Gamma, v :^{\chi} \tau) & \coloneqq & \textbf{if}\ \textsc{NumRefs}\ v\ [\Gamma' \coloneqq \sigma] = 1 \\
 & & \quad \textsc{EraseBlock}\ (\textsc{Reg}_1\ v) \\
 & & \textsc{Erase}\ [\Gamma' \coloneqq \sigma]\ \Gamma \\
\textsc{ShareBlock}\ r\ n & \coloneqq & \texttt{beq}\ r\ \texttt{x0}\ l \qquad\qquad (l\ \text{fresh}) \\
 & & \quad \texttt{lw temp 0}\ r \\
 & & \quad \texttt{addi temp temp}\ n \\
 & & \quad \texttt{sw temp 0}\ r \\
 & & l : \\
\textsc{EraseBlock}\ r & \coloneqq & \texttt{beq}\ r\ \texttt{x0}\ l_1 \qquad\qquad (l_1, l_2\ \text{fresh}) \\
 & & \quad \texttt{lw temp 0}\ r \\
 & & \quad \texttt{beq temp x0}\ l_2 \\
 & & \quad\quad \texttt{addi temp temp -1} \\
 & & \quad\quad \texttt{sw temp 0}\ r \\
 & & \quad\quad \texttt{j}\ l_1 \\
 & & \quad l_2 : \texttt{sw todo 0}\ r \\
 & & \quad\quad \texttt{mv todo}\ r \\
 & & l_1 :
\end{array}
$$

$\textsc{Reg}_1\ v$	first register of variable v
$\textsc{Reg}_2\ v$	second register of variable v
$\textsc{Index}\ X$	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
$\textsc{Move}\ [\Gamma' \coloneqq \sigma]$	parallel moves for assignments $[\Gamma' \coloneqq \sigma]$
$\textsc{NumRefs}\ v\ [\Gamma' \coloneqq \sigma]$	number of variables v is assigned to in $[\Gamma' \coloneqq \sigma]$
$\textsc{ShareFields}\ r$	$\textsc{ShareBlock}\ f\ 1$ 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

Fig. 9. Auxiliary definitions for the translation.

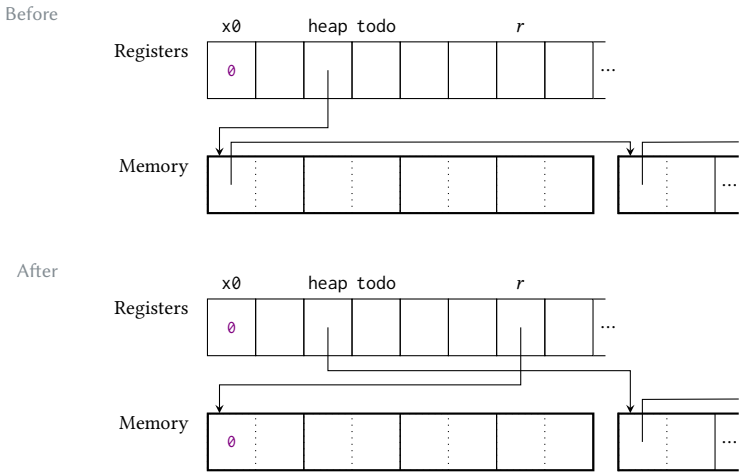
We translate each top-level definition to a label in RISC-V followed by the translation of the corresponding statement, and we translate calls to top-level definitions to direct jumps. We translate explicit substitutions to *erase* and *share* the variables appropriately, and to perform parallel moves 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 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.

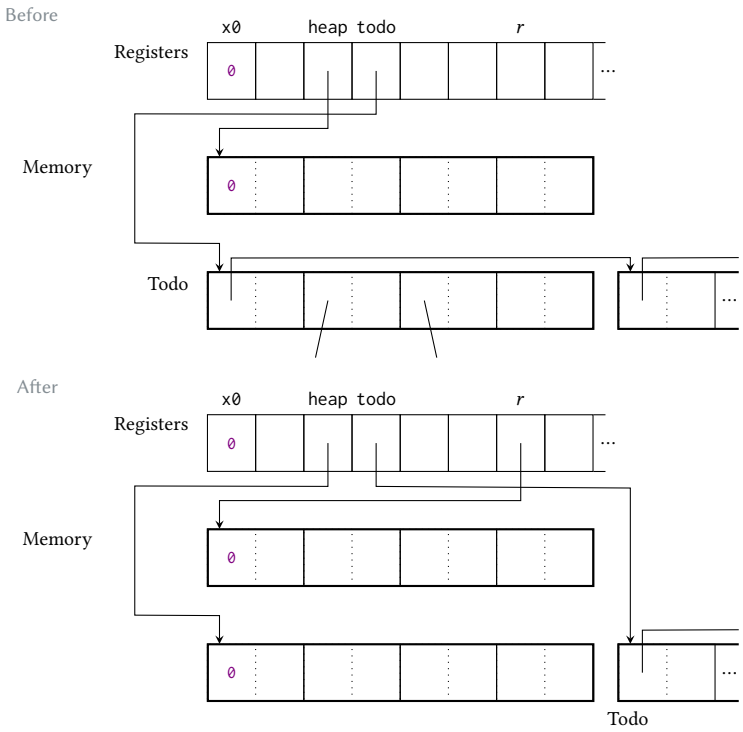
The remaining statements are those concerned with producers and consumers. The duality of producers and consumers leads to dual translations.

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 10) 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.

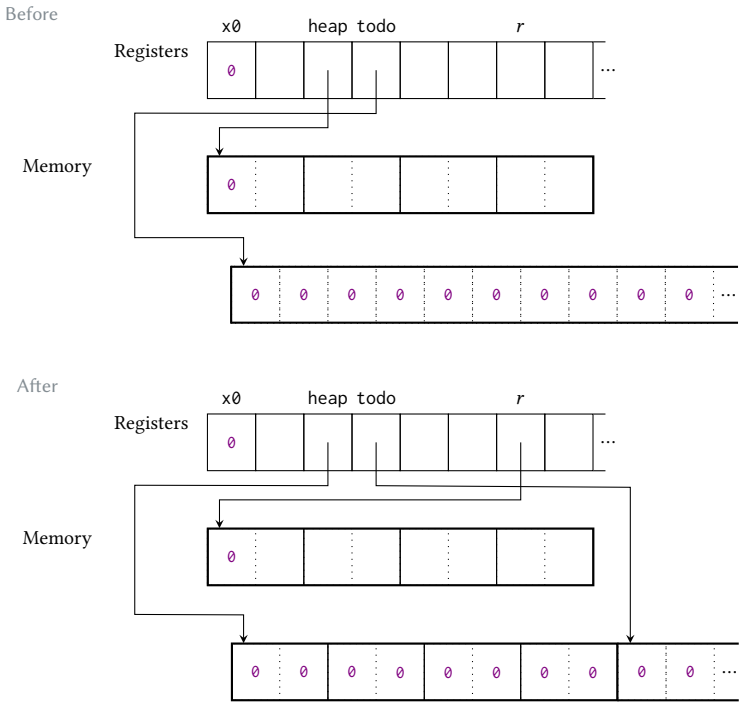
When the environment is stored, we have to restore the invariant that `heap` always points to a free block of memory. To do so, Acquire distinguishes three cases. If the linear free list pointed to by `heap` contains a further block, then the first field of the previously free block is non-zero as it 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.



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 10). 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 10).

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.

Auxiliary Definitions (continued)

$$
\begin{array}{rcll}
\textsc{JTable}\,\{\,\overline{X_i(\Gamma_i) \Rightarrow s_i}\,\}\,\Gamma & := & \texttt{j}\ \overline{l_i} & (\overline{l_i}\ \text{fresh}) \\
 & & \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, \bar{l}) & := & l_1 : \textsc{Load}\ (\textsc{Reg}_1\ x)\ \Gamma_1 & (x\ \text{fresh after}\ \Gamma) \\
 & & \quad \mathcal{M}[\![s_1]\!] & \\
 & & \textsc{JTableB}\ b\ \Gamma\ \bar{l} & \\
\textsc{VTable}\,\{\,\overline{X_i(\Gamma_i) \Rightarrow s_i}\,\}\,\Gamma_0 & := & \texttt{j}\ \overline{l_i} & (\overline{l_i}\ \text{fresh}) \\
 & & \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, \bar{l}) & := & l_1 : \textsc{Load}\ (\textsc{Reg}_1\ x)\ \Gamma_0 & (x\ \text{fresh after}\ \Gamma_1) \\
 & & \quad \mathcal{M}[\![s_1]\!] & \\
 & & \textsc{VTableB}\ b\ \Gamma_0\ \bar{l} & \\
\textsc{Load}\ r\ \Gamma & := & \textsc{Release}\ r & \\
 & & \textsc{LoadV}\ r\ \Gamma & \\
\textsc{LoadV}\ r\ (\Gamma, v :^{\chi} \tau) & := & \texttt{lw}\ (\textsc{Reg}_2\ v)\ (\textsc{Offset}_2\ v)\ r & \\
 & & \texttt{lw}\ (\textsc{Reg}_1\ v)\ (\textsc{Offset}_1\ v)\ r & \\
 & & \textsc{LoadV}\ r\ \Gamma & \\
\textsc{Store}\ r\ \Gamma & := & \textsc{StoreV}\ \Gamma & \\
 & & \textsc{Acquire}\ r & \\
\textsc{StoreV}\ (\Gamma, v :^{\chi} \tau) & := & \texttt{sw}\ (\textsc{Reg}_2\ v)\ (\textsc{Offset}_2\ v)\ \texttt{heap} & \\
 & & \texttt{sw}\ (\textsc{Reg}_1\ v)\ (\textsc{Offset}_1\ v)\ \texttt{heap} & \\
 & & \textsc{StoreV}\ \Gamma & \\
\textsc{Acquire}\ r & := & \texttt{mv}\ r\ \texttt{heap} & (l_1, l_2, l_3\ \text{fresh}) \\
 & & \texttt{lw heap 0 heap} & \\
 & & \texttt{beq heap x0}\ l_1 & \\
 & & \quad \texttt{sw x0 0}\ r & \\
 & & \quad \texttt{j}\ l_2 & \\
 & & l_1 : \texttt{mv heap todo} & \\
 & & \quad \texttt{lw todo 0 todo} & \\
 & & \quad \texttt{beq todo x0}\ l_3 & \\
 & & \quad\quad \texttt{sw x0 0 heap} & \\
 & & \quad\quad \textsc{EraseFields}\ \texttt{heap} & \\
 & & \quad\quad \texttt{j}\ l_2 & \\
 & & \quad l_3 : \texttt{addi todo heap 32} & \\
 & & l_2 : & \\
\textsc{Release}\ r & := & \texttt{lw temp 0}\ r & (l_1, l_2\ \text{fresh}) \\
 & & \texttt{beq temp x0}\ l_1 & \\
 & & \quad \texttt{addi temp temp -1} & \\
 & & \quad \texttt{sw temp 0}\ r & \\
 & & \quad \textsc{ShareFields}\ r & \\
 & & \quad \texttt{j}\ l_2 & \\
 & & l_1 : \texttt{sw heap 0}\ r & \\
 & & \quad \texttt{mv heap}\ r & \\
 & & l_2 : &
\end{array}
$$

Fig. 10. Auxiliary definitions for the translation (continued).

8.7 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 sections, 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 implementation of the compilation pipeline and report benchmarking results. Both are attached as supplementary material.

9.1 Implementation

We have implemented the full compilation pipeline in Rust[7]. 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 3.1. 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 Section 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.

9.1.1 RISC-V. The implementation of the translation to RISC-V closely follows the presentation in Section 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.

9.1.2 x86-64. The code we generate for x86-64 is again very similar to the translation in Section 8. However, since we want to create an executable to run on a Linux system, we emit code for the Yasm Assembler[8], a more performant rewrite of the Netwide Assembler (NASM)[9]. 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 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

[7] https://www.rust-lang.org
[8] https://github.com/yasm/yasm
[9] https://nasm.us

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.

9.1.3 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[10]. 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`[11], 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.

9.2.1 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[12] (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[13] (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.

OCaml[14] (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[15] [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`).

[10] https://www.gnu.org/software/binutils/
[11] https://github.com/sharkdp/hyperfine
[12] https://www.smlnj.org
[13] http://mlton.org
[14] https://ocaml.org
[15] https://koka-lang.github.io/koka/doc/index.html

Effekt[16] [Brachthäuser et al. 2020] (0.45.0) generates LLVM[17] 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.

9.2.2 Programs. Table 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[18] (marked with ∗) and a few are taken and adapted from Haskell's NoFib suite [Chen and Parreaux 2024; Partain 1993] (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 9.2.4.

9.2.3 Results. Figure 12 and Figure 13 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 implementation 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`.

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

[16] https://effekt-lang.org
[17] https://llvm.org
[18] https://github.com/ManticoreProject/benchmark

Table 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.

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

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 11.

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.

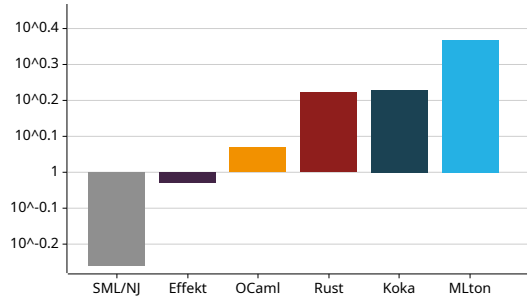


Fig. 11. Geometric means of relative performance results on a logarithmic scale.

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 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.

9.2.4 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 use control operators in a few places with the goal of measuring

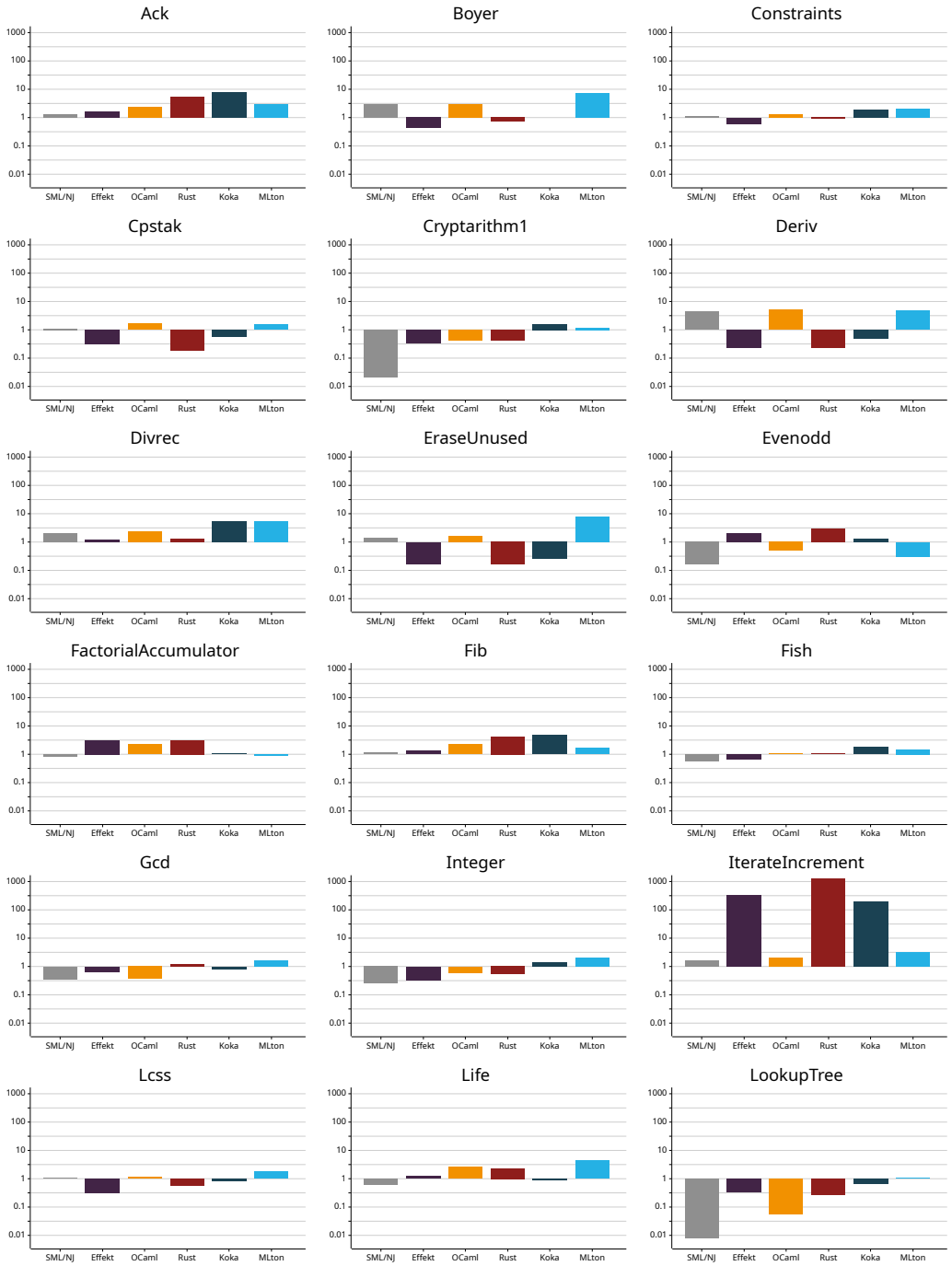


Fig. 12. Relative performance results on a logarithmic scale (A-L).

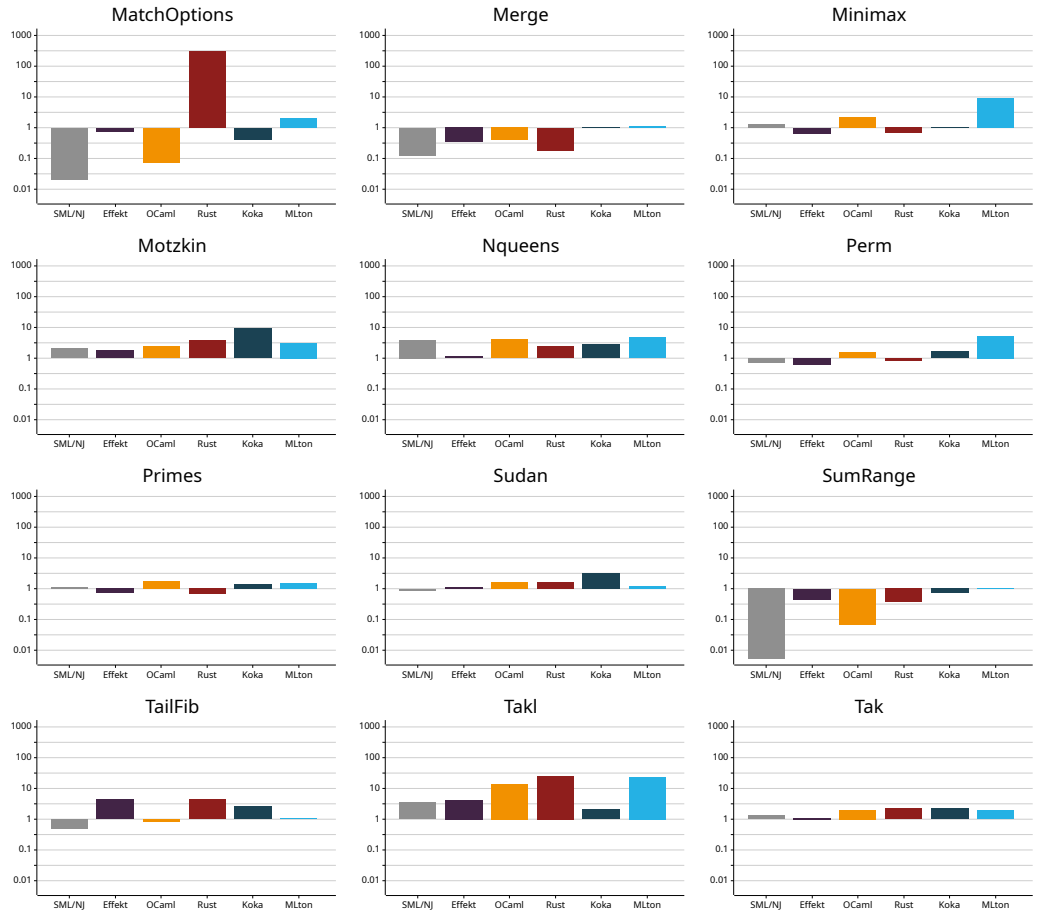


Fig. 13. Relative performance results on a logarithmic scale (M-Z).

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

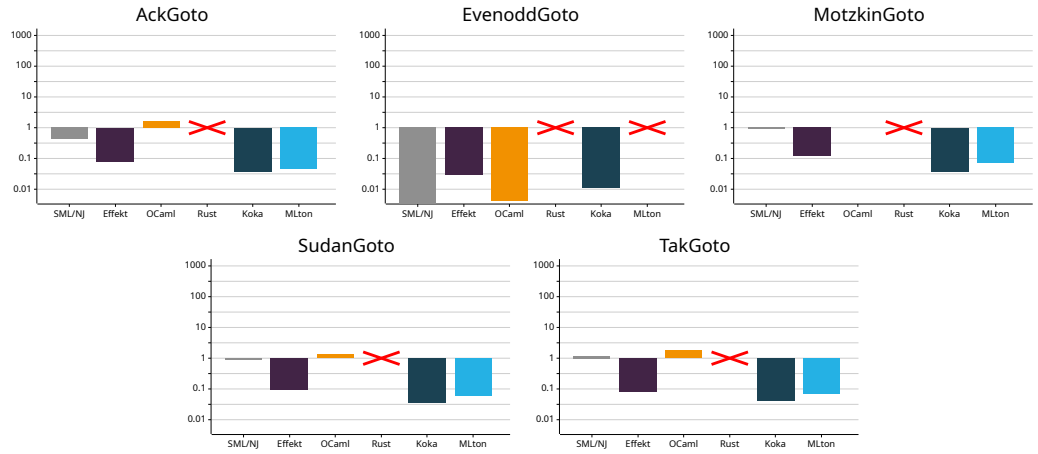


Fig. 14. Relative performance results on a logarithmic scale for `Goto`-benchmarks.

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 14 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.

10 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.

10.1 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.

10.2 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].

10.3 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 [Felleisen 1987; Landin 1965; Thielecke 1998] 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].

10.4 Intermediate Representations

Compiling programming languages via continuation-passing style is a long and ongoing tradition [Appel 1992; Kennedy 2007; Steele 1978]. Calculi in continuation-passing style have been thoroughly investigated, ranging from foundational systems such as Thielecke [1998]'s CPS calculus [Torrens et al. 2024] or extensions such as Levy [2003]'s Jump-With-Argument calculus, to more practice-oriented systems such as Appel [1992]'s full-fledged compiler IR. In CPS, 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 [Kennedy 2007; Thielecke 2002] for implementing exceptions. We can mix different such calling and returning conventions within the same program. Control flow in CPS is explicit, just as it is in our IRs. This enables expressing complex non-local control flow directly, as we have demonstrated in Section 2.2. It further makes many optimizations simpler [Appel 1992; Kennedy 2007], and it facilitates rather direct code generation. However, in contrast to CPS, our consumers generalize continuations: Contexts are fully first-class, following classical sequent calculus. This makes the consumers scrutable, opening up possibilities for optimizations that would be harder to perform in CPS (cf. Section 11).

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 or 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 [Flanagan et al. 1993; Sabry and Felleisen 1992]. 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 **Core** language or for its focused fragment, 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. In **Core**, we can hence retain much of the nesting structure of the original program. In contrast to CPS and to our approach, languages in A-normal form cannot express complex non-local control flow directly, making them less expressive in this regard.

Maurer et al. [2017] and Cong et al. [2019] each present an intermediate representation in direct style 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, as its main purpose is to model efficient join points. 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. While being more expressive, this makes our consumers less suitable for expressing efficient join points. The same is true for continuations in CPS, unless restricted to be second-class [Kennedy 2007]. However, our (second-class) top-level definitions do give us a simple way to express efficient join points, as we have seen in the translation to **Core** (Figure 5) and the shrinking transformation (Section 6.2.3).

Graph-based intermediate representations track data and control flow dependencies in a graph structure. They are most commonly found in the compilation of imperative languages, but there also exist systems designed for higher-order languages. Leißa et al. [2015] present a flat graph-based intermediate representation for higher-order languages 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. This is in contrast to the graph-based intermediate representation for higher-order languages presented by Bračevac et al. [2023], which supports nesting of blocks and bindings, expressing lexical restrictions on data dependencies. Similar to the IR of Leißa et al. [2015], our explicit substitutions in **AxCut** allow us to view our top-level functions as not taking parameters, but instead being annotated with the free variables they assume to be present. Notably, the IR of Leißa et al. [2015] is in CPS. However, complex non-local control flow is not specifically explored.

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, so the system can be viewed as a sequent-calculus variant of the direct-style system of Maurer et al. [2017], for which the calculus in Downen et al. [2016] has, in fact, been the inspiration. In contrast to Downen et al. [2016]'s work, consumers in our symmetric languages are first-class and hence can model complex non-local control flow. 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.

11 Future Work

While our compilation pipeline is complete in that it can compile programs written in the surface language down to machine code, its purpose is not only to show that intermediate representations based on classical sequent calculus are a promising option for compilation, but also to serve as a basis for further investigations. Here, we outline several directions for future work.

Expressive Control Flow. Our surface language **Fun** enables non-trivial control flow through the use of the control operators **label** and **goto**, which can be expressed in a straightforward way in our intermediate languages. However, these control operators only provide undelimited continuations, which on their own are rather limited. In the future, it will be interesting to investigate the compilation of more expressive high-level control primitives. In particular, features for structuring interactive programs, such as async/await, coroutines, or effect handlers have become increasingly mainstream in the past decade. Effect handlers [Plotkin and Pretnar 2009, 2013] are especially promising, as they are able to express many other control structures. While efficient implementations of effect handlers often rely on a runtime system, our goal is to directly compile them efficiently via our pipeline, without resorting to any runtime support. For a particularly promising approach to do so, we can look at the literature on lexical effect handlers [Biernacki et al. 2019; Brachthäuser et al. 2020; Zhang and Myers 2019] and translations for them into iterated CPS [Müller et al. 2023; Schuster et al. 2022]. We are confident that it is possible to adapt this approach to our setting by using multiple consumer parameters to model delimited continuations.

Optimizations. We saw in Section 9.2 that our compiler produces reasonably performant code that can be compared to state-of-the-art compilers, even though we do not perform optimizations. However, we also saw that optimizations do make a significant difference for many programs. To get on par with or exceed the performance of heavily optimizing compilers, our compilation pipeline must hence be augmented with optimization passes. 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, making our pipeline particularly promising in this regard.

It is well-known that the explicit control flow in continuation-passing style is well-suited for performing optimizations [Appel 1992; Kennedy 2007]. Even optimizations that can be challenging to implement in direct style, such as contification or optimizations for advanced control flow, often amount to standard inlining and reduction [Kennedy 2007; Schuster et al. 2020]. Our sequent-calculus-based intermediate representations also offer explicit control flow, and the equational theory of **Core** allows us to freely perform many rewrites. The abstract machine semantics for **Core** and **AxCut** further provides a useful cost model for optimizations. Moreover, in the stages where all terms are named, we can easily perform shrinking rewrites exhaustively [Kennedy 2007]. Consequently, we expect to be able to perform the same optimizations as in CPS in a similarly simple fashion [Downen et al. 2016]. In contrast to CPS, however, we do not have to make the evaluation order explicit, as consumers are fully first-class. Hence, we can preserve most of the nesting structure of the original program, and, unlike continuations, the structure of our consumers is scrutable. This facilitates several optimizations that would be harder to perform in CPS, such as the application of custom rewrite rules to enable user optimizations [Downen et al. 2016].

Furthermore, while data types can be found in several intermediate languages, codata types are rare, particularly in combination with advanced control flow. We believe that this brings us into the position to perform new interesting optimizations, in particular for asynchronous language features structuring interactive programs. The symmetry of our languages further makes it possible to apply the same optimizations for call-by-value data types and call-by-name codata types in a uniform way. In addition, with our intermediate languages being rooted in logic, we can try to leverage this connection and identify a fragment for which full *cut elimination* is possible, i.e., where all indirections in the program can be eliminated statically. We expect such cut-free terms to also enjoy the *subformula property*, guaranteeing that no dynamic allocations occur. Another

interesting direction is the further integration of linearity and uniqueness into our type system, which can be helpful, for example, for optimizing our memory management scheme.

Additional Expressivity. The combination of explicit control flow, general data and codata types, and arbitrary recursion via top-level labels makes our intermediate representations fairly expressive. Still, turning them into an attractive target for a multitude of surface languages requires additional features. Apart from the necessary addition of further built-in types and externals for interacting with the outside world, there are several interesting extensions. A particularly important one is support for polymorphism. One straightforward way to add polymorphism is by monomorphization, for which **Core** with its codata types is an excellent candidate [Lutze et al. 2025]. While monomorphization is sufficient for many use cases, not all cases of polymorphism can be monomorphized. An alternative is to use a uniform representation for all types. A problem arising with uniform representation in our approach is that we rely on η-expansion during the shrinking transformation. However, we are confident that this problem can be solved by treating polymorphic types in a way similar to built-in types during shrinking.

Another interesting addition would be mutable state. One possible way to implement mutable state is via state passing. An alternative way is to support first-class mutable reference cells throughout our compilation pipeline. Since mutable reference cells allow for the creation of cycles on the heap, the integration with our memory management scheme in an efficient way is not easy. The same is true for the addition of mutable arrays. We conjecture that a further incorporation of linearity and uniqueness into our type system can help us in designing and implementing these features.

Mechanized Correctness. Compilers are complex pieces of software and the correctness of all programs compiled with them relies on their correctness. Thus, it is valuable to mechanically verify that a compiler indeed produces correct machine code, in particular for safety-critical systems. Our compilation pipeline is divided into several steps, each of which can be individually verified against the given semantics of the involved languages. Moreover, we hope that the simplicity of code generation from **AxCut** makes an equally simple mechanized proof of its correctness possible. The result would be a mechanically verified compilation pipeline from the surface language down to machine code. We further conjecture that each of the steps can be formulated in a way that allows us to reason about the time and space usage of compiled programs.

12 Conclusion

In this paper, 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 direct 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.

In the future, it will be 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. 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. Finally, we hope that the simplicity of code generation from **AxCut** makes an equally simple mechanized proof of its correctness possible.

Data Availability Statement

The Rust implementation of the compiler described in this article is available at [Müller et al. 2025a]. The benchmarks are available at [Müller et al. 2025b].

References

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 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 27–38. doi:10.1145/2480359.2429075

Jean-Marc Andreoli. 1992. Logic Programming with Focusing Proofs in Linear Logic. *Journal of Logic and Computation* 2 (1992), 297–347. Issue 3. https://doi.org/10.1093/logcom/2.3.297

Andrew W. Appel. 1992. *Compiling with Continuations.* Cambridge University Press.

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

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, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019. Binders by Day, Labels by Night: Effect Instances via Lexically Scoped Handlers. *Proc. ACM Program. Lang.* 4, POPL, Article 48 (Dec. 2019), 29 pages. doi:10.1145/3371116

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, Article 250 (aug 2024), 31 pages. 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, Article 126 (nov 2020), 30 pages. doi:10.1145/3428194

Oliver Bračevac, Guannan Wei, Songlin Jia, Supun Abeysinghe, Yuxuan Jiang, Yuyan Bao, and Tiark Rompf. 2023. Graph IRs for Impure Higher-Order Languages: Making Aggressive Optimizations Affordable with Precise Effect Dependencies. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 236 (Oct. 2023), 31 pages. doi:10.1145/3622813

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, Article 79 (July 2019), 28 pages. 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. https://doi.org/10.1145/357766.351262

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 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

Paul Downen. 2017. *Sequent Calculus: A Logic and a Language for Computation and Duality.* Ph. D. Dissertation. University of Oregon.

Paul Downen and Zena M. Ariola. 2014. The Duality of Construction. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410 (ESOP '14)*. Springer, Berlin, Heidelberg, 249–269. https://doi.org/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). https://doi.org/10.1017/S0956796818000023

Paul Downen and Zena M. Ariola. 2020. Compiling With Classical Connectives. *Logical Methods in Computer Science* Volume 16, Issue 3 (Aug. 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 (LIPIcs, Vol. 195)*, Naoki Kobayashi (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 1–32. https://doi.org/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*. 74–88.

Paul Downen, Zachary Sullivan, Zena M. Ariola, and Simon Peyton Jones. 2019. Codata in Action. In *European Symposium on Programming (ESOP '19)*. Springer, 119–146. doi:10.1007/978-3-030-17184-1_5

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* (Canterbury, United Kingdom) *(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

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* (Albuquerque, New Mexico, USA) *(PLDI '93)*. Association for Computing Machinery, New York, NY, USA, 237–247.

Gerhard Gentzen. 1935a. Untersuchungen über das logische Schließen. I. *Mathematische Zeitschrift* 35 (1935), 176–210.

Gerhard Gentzen. 1935b. Untersuchungen über das logische Schließen. II. *Mathematische Zeitschrift* 39 (1935), 405–431.

Jean-Yves Girard. 1987. Linear Logic. *Theoretical Computer Science* 50, 1 (1987), 1–101. https://doi.org/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* (San Francisco, California, USA) *(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

Martin Hofmann. 1997. Syntax and Semantics of Dependent Types. In *Extensional Constructs in Intensional Type Theory*. Springer, London, 13–54. doi:10.1007/978-1-4471-0963-1_2

Andrew Kennedy. 2007. Compiling with continuations, continued. *SIGPLAN Not.* 42, 9 (oct 2007), 177–190. doi:10.1145/1291220.1291179

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* (Kumamoto, Japan). Springer-Verlag, Berlin, Heidelberg, 188–216. doi:10.1007/978-981-97-2300-3_11

Peter John Landin. 1965. Correspondence between ALGOL 60 and Church's Lambda-notation: part I. *Commun. ACM* 8, 2 (feb 1965), 89–101. doi:10.1145/363744.363749

Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types, In MSFP. *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* (San Francisco, California) *(CGO '15)*. IEEE Computer Society, USA, 202–212. doi:10.5555/2738600.2738626

Paul Blain Levy. 2003. *Call-By-Push-Value: A Functional/Imperative Synthesis*. Semantics Structures in Computation, Vol. 2. Springer, Dordrecht. doi:10.1007/978-94-007-0954-6

Anton Lorenzen, Daan Leijen, and Wouter Swierstra. 2023. FP²: Fully in-Place Functional Programming. *Proc. ACM Program. Lang.* 7, ICFP, Article 198 (aug 2023), 30 pages. doi:10.1145/3607840

Matthew Lutze, Philipp Schuster, and Jonathan Immanuel Brachthäuser. 2025. The Simple Essence of Monomorphization. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 116 (April 2025), 27 pages. doi:10.1145/3720472

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* (Freiburg, Germany) *(ICFP '07)*. Association for Computing Machinery, New York, NY, USA, 277–288. doi:10.1145/1291151.1291194

Luke Maurer, Paul Downen, Zena M. Ariola, and Simon Peyton Jones. 2017. Compiling without Continuations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 482–494. doi:10.1145/3062341.3062380

Robin Milner, Robert Harper, David MacQueen, and Mads Tofte. 1997. *The Definition of Standard ML*. The MIT Press. doi:10.7551/mitpress/2319.001.0001

Serkan Muhcu, Philipp Schuster, Michel Steuwer, and Jonathan Immanuel Brachthäuser. 2025. Multiple Resumptions and Local Mutable State, Directly. *Proc. ACM Program. Lang.* 9, ICFP, Article 260 (Aug. 2025), 30 pages. doi:10.1145/3747529

Marius Müller, Philipp Schuster, Jonathan Lindegaard Starup, Klaus Ostermann, and Jonathan Immanuel Brachthäuser. 2023. From Capabilities to Regions: Enabling Efficient Compilation of Lexical Effect Handlers. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 255 (oct 2023), 30 pages. doi:10.1145/3622831

Guillaume Munch-Maccagnoni. 2009. Focalisation and Classical Realisability. In *Computer Science Logic: 23rd international Workshop, CSL 2009, 18th Annual Conference of the EACSL* (Coimbra, Portugal) *(CSL '09)*, Erich Grädel and Reinhard Kahle (Eds.). Springer, Berlin, Heidelberg, 409–423. https://doi.org/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, 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

Klaus Ostermann, David Binder, Ingo Skupin, Tim Süberkrüb, and Paul Downen. 2022. Introduction and Elimination, Left and Right. *Proc. ACM Program. Lang.* 6, ICFP, Article 106 (2022), 28 pages. doi:10.1145/3547637

Zoe Paraskevopoulou and Anvay Grover. 2021. Compiling with continuations, correctly. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 114 (Oct. 2021), 29 pages. 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*, John Launchbury and Patrick Sansom (Eds.). Springer London, London, 195–202. doi:10.1007/978-1-4471-3215-8_17

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

Gordon Plotkin and Matija Pretnar. 2009. Handlers of algebraic effects. In *European Symposium on Programming*. Springer, 80–94. doi:10.1007/978-3-642-00590-9_7

Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9, 4 (2013). doi:10.2168/LMCS-9(4:23)2013

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 Charles Reynolds. 1972. Definitional Interpreters for Higher-Order Programming Languages. In *ACMConf* (Boston). Association for Computing Machinery, New York, NY, USA, 717–740. https://doi.org/10.1145/800194.805852

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* (San Francisco, California, USA) *(LFP '92)*. Association for Computing Machinery, New York, NY, USA, 288–298.

Philipp Schuster, Jonathan Immanuel Brachthäuser, Marius Müller, and Klaus Ostermann. 2022. A Typed Continuation-Passing Translation for Lexical Effect Handlers. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 566–579. doi:10.1145/3519939.3523710

Philipp Schuster, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. 2020. Compiling Effect Handlers in Capability-Passing Style. *Proc. ACM Program. Lang.* 4, ICFP, Article 93 (Aug. 2020), 28 pages. doi:10.1145/3408975

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, Article 142 (apr 2025), 28 pages. 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* (Virtual, Canada) *(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

Hayo Thielecke. 2002. Comparing Control Constructs by Double-Barrelled CPS. *Higher Order Symbol. Comput.* 15, 2–3 (Sept. 2002), 141–160. doi:10.1023/A:1020887011500

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, Article 241 (Aug. 2024), 30 pages. 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* (Uppsala, Sweden) *(ICFP '03)*. Association for Computing Machinery, New York, NY, USA, 189–201. https://doi.org/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). arXiv:1903.00982 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, Article 71 (aug 2021), 30 pages. doi:10.1145/3473576

Noam Zeilberger. 2008. On the Unity of Duality. *Annals of Pure and Applied Logic* 153, 1-3 (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. Advisor(s) Pfenning, Frank and Lee, Peter.

Yizhou Zhang and Andrew C. Myers. 2019. Abstraction-safe Effect Handlers via Tunneling. *Proc. ACM Program. Lang.* 3, POPL, Article 5 (Jan. 2019), 29 pages. doi:10.1145/3290318

Appendices

A Typing Rules for Operational Semantics of AxCut

The typing rules for values, environments, and machine configurations of the operational semantics of **AxCut** are straightforward. Machine configurations are typed in rule Execution. The program Θ must be well-formed, the statement s must be well-typed in Θ and a type environment Γ, and the value environment E must be well-typed against Γ in Θ.

Configuration Typing $\vdash M$

$$\frac{\Theta \mid \Gamma \vdash s \qquad \Theta \vdash E : \Gamma \qquad \vdash \Theta \text{ Ok}}{\vdash \langle s \parallel E \parallel \Theta \rangle} \textsc{Execution}$$

Well-typedness of a value environment E against a context Γ means that for each binding v in E there has to be a corresponding binding in Γ and the value bound to v must be of the specified type.

Environment Typing $\Theta \vdash E : \Gamma$

$$\frac{\Theta \vdash V :^{\chi} \tau \qquad \ldots}{\Theta \vdash v \mapsto V, \ldots \; : \; v :^{\chi} \tau, \ldots} \textsc{Bindings}$$

Values are also typed in the context of a program Θ. Typing for constructor and destructor values in rules Xtor-π requires the value environment E to cover the parameters of the constructor or destructor X. For closures, the typing rules Closure-π require the closure environment E to cover the common part Γ of the type environment for each branch and that the statement of each branch is well-typed in the context of its parameter list extended with Γ.

Value Typing $\Theta \vdash V :^{\chi} \tau$

$$\frac{\pi\, T\{X_1(\Gamma_1) \ldots\} \in \Theta \qquad \Theta \vdash E : \Gamma \qquad \Theta \mid \Gamma_1, \Gamma \vdash s_1 \quad \ldots}{\Theta \vdash \{E;\; X_1(\Gamma_1) \Rightarrow s_1, \ldots\} :^{\chi_2(\pi)} T} \textsc{Closure-}\pi$$

$$\frac{\pi\, T\{\ldots, X(\Gamma) \ldots\} \in \Theta \qquad \Theta \vdash E : \Gamma}{\Theta \vdash \{X;\; E\} :^{\chi_1(\pi)} T} \textsc{Xtor-}\pi \qquad \frac{}{\Theta \vdash n :^{\text{prd}} \mathbf{i64}} \textsc{Int}$$

B Simulation for the Translation from Fun to Core

We first define the translation of stacks as follows:

$$\begin{aligned}
\mathcal{C}'[\![\mathbf{exit}]\!] &:= \tilde{\mu}x.\mathbf{exit}\, x \quad (x \text{ fresh}) \\
\mathcal{C}'[\![\Box.D(v) :: S]\!] &:= D(\mathcal{C}'[\![v]\!], \mathcal{C}'[\![S]\!]) \\
\mathcal{C}'[\![\Box.\mathbf{case}\,\{K_1(\Gamma_1) \Rightarrow p_1, \ldots\} :: S]\!] &:= \mathbf{case}\,\{K_1(\Gamma_1) \Rightarrow \mathcal{C}'[\![p_1]\!]_{\mathcal{C}'[\![S]\!]}, \ldots\} \\
\mathcal{C}'[\![\mathbf{let}\; x = \Box;\; p :: S]\!] &:= \tilde{\mu}x.\mathcal{C}'[\![p]\!]_{\mathcal{C}'[\![S]\!]} \\
\mathcal{C}'[\![A :: S]\!] &:= \tilde{\mu}x.\mathcal{C}'[\![A[x]]\!]_{\mathcal{C}'[\![S]\!]} \quad \text{if } A \neq \mathbf{let}\; x = \Box;\; p \quad (x \text{ fresh})
\end{aligned}$$

Note that with this definition, initial states are translated to initial states: If p is the body of the main function in **Fun**, the translation of the initial state is

$$\begin{aligned}
\mathcal{C}'[\![\langle p \parallel \mathbf{exit} \parallel \Theta \rangle]\!] &= \langle \mathcal{C}'[\![p]\!]_{\mathcal{C}'[\![\mathbf{exit}]\!]} \parallel \mathcal{C}'[\![\Theta]\!] \rangle \\
&= \langle \mathcal{C}'[\![p]\!]_{\tilde{\mu}x.\mathbf{exit}\, x} \parallel \mathcal{C}'[\![\Theta]\!] \rangle
\end{aligned}$$

This is an initial state in **Core**, since $\mathcal{C}'[\![p]\!]_{\tilde{\mu}x.\mathbf{exit}\, x}$ is the body of the translated main function.

Before we can prove the desired result, we need a few lemmas. First, the translations of values that are not stacks are values and the translations of stacks are covalues (note that argument frames are never of codata types, $\neg(A :^{\text{cns}} \textbf{codata}\, T\,\{\dots\})$), i.e., the translations of argument values are argument values. We write v' for the syntactic category of values without stacks.

Lemma B.1 ((Co)Values).
*Let v' be a well-typed value that is not a stack, S a well-typed stack, and a a well-typed argument value in **Fun**. Then $\mathcal{C}'[\![v']\!]$ is a value, $\mathcal{C}'[\![S]\!]$ is a covalue, and $\mathcal{C}'[\![a]\!]$ is an argument value in **Core**.*

Similarly, the translation of producers that are not argument values are not values.

Lemma B.2 (Non-Values).
*Let p be a well-typed producer in **Fun** that is not an argument value, $p \notin a$. Then $\mathcal{C}'[\![p]\!]$ is not a value in **Core**.*

Moreover, the translation is compatible with substitutions.

Lemma B.3 (Substitutions).
*Let p be a well-typed producer and S a well-typed stack in **Fun**, and c a consumer in **Core** occurring during the translation. Further, let Γ be a context and v a type-compatible list of argument values in **Fun**. Then*

$$
\begin{aligned}
(\mathcal{C}'[\![p]\!]_c)[\Gamma \mapsto \mathcal{C}'[\![v]\!]] &= \mathcal{C}'[\![p[\Gamma \mapsto v]]\!]_{c[\Gamma \mapsto \mathcal{C}'[\![v]\!]]} \\
(\mathcal{C}'[\![p]\!])[\Gamma \mapsto \mathcal{C}'[\![v]\!]] &= \mathcal{C}'[\![p[\Gamma \mapsto v]]\!] \\
(\mathcal{C}'[\![S]\!])[\Gamma \mapsto \mathcal{C}'[\![v]\!]] &= \mathcal{C}'[\![S[\Gamma \mapsto v]]\!]
\end{aligned}
$$

Finally, we show how the lifting function $\mathcal{L}_v(\cdot)[\cdot]$ behaves.

Lemma B.4 (Lifting).
*Let p be a well-typed producer, v a list of well-typed argument values, and σ a list of well-typed arguments in **Fun**. Further, let c be a consumer in **Core** occurring during the translation. Then*

$$\mathcal{L}_v(\mathcal{C}'[\![v, \sigma]\!])[\lambda as.\mathcal{C}'[\![p]\!]_{D(as,c)}] = \mathcal{L}_v(\mathcal{C}'[\![\sigma]\!])[\lambda as.\mathcal{C}'[\![p]\!]_{D(\mathcal{C}'[\![v]\!],as,c)}]$$

In particular

$$\mathcal{C}'[\![p.D(v)]\!]_c = \mathcal{L}_v(\mathcal{C}'[\![v]\!])[\lambda as.\mathcal{C}'[\![p]\!]_{D(as,c)}] = \mathcal{C}'[\![p]\!]_{D(\mathcal{C}'[\![v]\!],c)}$$

Moreover, if e is well-typed and not an argument value, $e \notin a$, then

$$\mathcal{L}_v(\mathcal{C}'[\![v, e, \sigma]\!])[\lambda as.\mathcal{C}'[\![p]\!]_{D(as,c)}] = \langle \mathcal{C}'[\![e]\!] \mid \tilde{\mu}x.\mathcal{L}_v(\mathcal{C}'[\![\sigma]\!])[\lambda as.\mathcal{C}'[\![p]\!]_{D(\mathcal{C}'[\![v]\!],x,as,c)}]\rangle$$

Proof Sketch for Theorem 5.5. We prove the claim by cases on the reduction step. As in the definition of the semantics, we omit the program from the machine configurations and only write the statement resulting from the translation of the machine state in **Fun**.

(arg) By cases on A.
$A = \textbf{let}\ x = \square;\ p_0$:
Note that $p \notin a$ means that $\neg(p : \textbf{codata}\, T\,\{\dots\})$, so

$$
\begin{aligned}
\mathcal{C}'[\![\textbf{let}\ x = p;\ p_0]\!]_{\mathcal{C}'[\![S]\!]} &= \mathcal{C}'[\![p]\!]_{\tilde{\mu}x.\mathcal{C}'[\![p_0]\!]_{\mathcal{C}'[\![S]\!]}} \\
&= \mathcal{C}'[\![p]\!]_{\mathcal{C}'[\![\textbf{let}\ x=\square;\ p_0 :: S]\!]}
\end{aligned}
$$

$A = \square.\mathbf{case}\,\{\,K_1(\Gamma_1) \Rightarrow p_1, \ldots\,\}$:
We have

$$\begin{aligned}\mathcal{C}'[\![p.\mathbf{case}\,\{\,K_1(\Gamma_1) \Rightarrow p_1, \ldots\,\}]\!]_{\mathcal{C}'[\![S]\!]} &= \mathcal{C}'[\![p]\!]_{\mathbf{case}\,\{\,K_1(\Gamma_1)\Rightarrow \mathcal{C}'[\![p_1]\!]_{\mathcal{C}'[\![S]\!]}, \ldots\,\}} \\ &= \mathcal{C}'[\![p]\!]_{\mathcal{C}'[\![\square.\mathbf{case}\,\{\,K_1(\Gamma_1)\Rightarrow p_1, \ldots\,\} :: S]\!]}\end{aligned}$$

$A = p_0.D(v, \square, \sigma)$:
If $p = K(\sigma)$ or $p = p_1 + p_2$ or $p = \mathbf{label}\ \alpha\ \{\,p_0\,\}$, then $\mathcal{C}'[\![p]\!]_c = \langle p \mid c\rangle$. Thus, using Lemma B.4, we find

$$\begin{aligned}\mathcal{C}'[\![p_0.D(v, p, \sigma)]\!]_{\mathcal{C}'[\![S]\!]} &= \mathcal{L}_v(\mathcal{C}'[\![v, p, \sigma]\!])[\lambda as.\mathcal{C}'[\![p_0]\!]_{D(as,\mathcal{C}'[\![S]\!])}] \\ &= \langle \mathcal{C}'[\![p]\!] \mid \tilde{\mu}x.\mathcal{L}_v(\mathcal{C}'[\![\sigma]\!])[\lambda as.\mathcal{C}'[\![p_0]\!]_{D(\mathcal{C}'[\![v]\!],x,as,\mathcal{C}'[\![S]\!])}]\rangle \\ &= \langle \mathcal{C}'[\![p]\!] \mid \tilde{\mu}x.\mathcal{L}_v(\mathcal{C}'[\![v, x, \sigma]\!])[\lambda as.\mathcal{C}'[\![p_0]\!]_{D(as,\mathcal{C}'[\![S]\!])}]\rangle \\ &= \langle \mathcal{C}'[\![p]\!] \mid \tilde{\mu}x.\mathcal{C}'[\![p_0.D(v, x, \sigma)]\!]_{\mathcal{C}'[\![S]\!]}\rangle \\ &= \langle \mathcal{C}'[\![p]\!] \mid \mathcal{C}'[\![p_0.D(v, \square, \sigma) :: S]\!]\rangle \\ &= \mathcal{C}'[\![p]\!]_{\mathcal{C}'[\![p_0.D(v,\square,\sigma) :: S]\!]}\end{aligned}$$

Otherwise, $\mathcal{C}'[\![p]\!] = \mu\alpha.\mathcal{C}'[\![p]\!]_\alpha$ with α fresh. Thus, using Lemma B.1, we find

$$\begin{aligned}\mathcal{C}'[\![p_0.D(v, p, \sigma)]\!]_{\mathcal{C}'[\![S]\!]} &= \langle \mathcal{C}'[\![p]\!] \mid \mathcal{C}'[\![p_0.D(v, \square, \sigma) :: S]\!]\rangle \\ &= \langle \mu\alpha.\mathcal{C}'[\![p]\!]_\alpha \mid \mathcal{C}'[\![p_0.D(v, \square, \sigma) :: S]\!]\rangle \quad (\alpha \text{ fresh}) \\ &\to_\mu \mathcal{C}'[\![p]\!]_{\mathcal{C}'[\![p_0.D(v,\square,\sigma) :: S]\!]}\end{aligned}$$

In all other cases we have $\mathcal{C}'[\![A[p]]\!]_{\mathcal{C}'[\![S]\!]} = A'[\mathcal{C}'[\![p]\!]]$ for some A' in **Core** for any p (since by Lemma B.1 $\mathcal{C}'[\![S]\!]$ is a covalue):
If $p = K(\sigma)$ or $p = p_1 + p_2$ or $p = \mathbf{label}\ \alpha\ \{\,p_0\,\}$, then $\mathcal{C}'[\![p]\!]_c = \langle p \mid c\rangle$. Thus, using Lemma B.2, we have

$$\begin{aligned}\mathcal{C}'[\![A[p]]\!]_{\mathcal{C}'[\![S]\!]} &= A'[\mathcal{C}'[\![p]\!]] \\ &\to_{arg\text{-}p} \langle \mathcal{C}'[\![p]\!] \mid \tilde{\mu}x.A'[x]\rangle \\ &= \langle \mathcal{C}'[\![p]\!] \mid \tilde{\mu}x.\mathcal{C}'[\![A[x]]\!]_{\mathcal{C}'[\![S]\!]}\rangle \\ &= \langle \mathcal{C}'[\![p]\!] \mid \mathcal{C}'[\![A :: S]\!]\rangle \\ &= \mathcal{C}'[\![p]\!]_{\mathcal{C}'[\![A :: S]\!]}\end{aligned}$$

Otherwise, $\mathcal{C}'[\![p]\!] = \mu\alpha.\mathcal{C}'[\![p]\!]_\alpha$ with α fresh. Thus, using Lemma B.2, we have

$$\begin{aligned}\mathcal{C}'[\![A[p]]\!]_{\mathcal{C}'[\![S]\!]} &\to_{arg\text{-}p} \langle \mathcal{C}'[\![p]\!] \mid \mathcal{C}'[\![A :: S]\!]\rangle \\ &= \langle \mu\alpha.\mathcal{C}'[\![p]\!]_\alpha \mid \mathcal{C}'[\![A :: S]\!]\rangle \quad (\alpha \text{ fresh}) \\ &\to_\mu \mathcal{C}'[\![p]\!]_{\mathcal{C}'[\![A :: S]\!]}\end{aligned}$$

(force) We have

$$\begin{aligned}\mathcal{C}'[\![p.D(\mathcal{C}'[\![v]\!])]\!]_{\mathcal{C}'[\![S]\!]} &= \mathcal{C}'[\![p]\!]_{D(\mathcal{C}'[\![v]\!],\mathcal{C}'[\![S]\!])} \\ &= \mathcal{C}'[\![p]\!]_{\mathcal{C}'[\![\square.D(v) :: S]\!]}\end{aligned}$$

(ret) By cases on F. Note that the producer v cannot be a stack, as the latter are typed as consumers. We hence write v' here to indicate this.

$$\begin{aligned}
\mathcal{C}'[\![v']\!]_{\mathcal{C}'[\![\square.D(v)::S]\!]} &= \mathcal{C}'[\![v']\!]_{D(\mathcal{C}'[\![v]\!],\mathcal{C}'[\![S]\!])} \\
&= \mathcal{C}'[\![v'.D(\mathcal{C}'[\![v]\!])]\!]_{\mathcal{C}'[\![S]\!]} \\
\mathcal{C}'[\![v']\!]_{\mathcal{C}'[\![\square.\mathbf{case}\,\{K_1(\Gamma_1)\Rightarrow p_1,\dots\}::S]\!]} &= \mathcal{C}'[\![v']\!]_{\mathbf{case}\,\{K_1(\Gamma_1)\Rightarrow \mathcal{C}'[\![p_1]\!]_{\mathcal{C}'[\![S]\!]},\dots\}} \\
&= \mathcal{C}'[\![v'.\mathbf{case}\,\{K_1(\Gamma_1)\Rightarrow p_1,\dots\}]\!]_{\mathcal{C}'[\![S]\!]}
\end{aligned}$$

Otherwise, since v' is a value, so is $\mathcal{C}'[\![v']\!]$ by Lemma B.1, and, moreover, $\mathcal{C}'[\![v']\!]_c = \langle \mathcal{C}'[\![v']\!] \mid c\rangle$. Thus, using Lemma B.3, we get

$$\begin{aligned}
\mathcal{C}'[\![v']\!]_{\mathcal{C}'[\![\mathbf{let}\ x=\square;\ p::S]\!]} &= \mathcal{C}'[\![v']\!]_{\tilde{\mu}x.\mathcal{C}'[\![p]\!]_{\mathcal{C}'[\![S]\!]}} \\
&= \langle \mathcal{C}'[\![v']\!] \mid \tilde{\mu}x.\mathcal{C}'[\![p]\!]_{\mathcal{C}'[\![S]\!]}\rangle \\
&= \mathcal{C}'[\![\mathbf{let}\ x = v';\ p]\!]_{\mathcal{C}'[\![S]\!]} \\
\mathcal{C}'[\![v']\!]_{\mathcal{C}'[\![A::S]\!]} &= \mathcal{C}'[\![v']\!]_{\tilde{\mu}x.\mathcal{C}'[\![A[x]]\!]_{\mathcal{C}'[\![S]\!]}} \quad (x \text{ fresh}) \\
&= \langle \mathcal{C}'[\![v']\!] \mid \tilde{\mu}x.\mathcal{C}'[\![A[x]]\!]_{\mathcal{C}'[\![S]\!]}\rangle \\
&\to_{\tilde{\mu}} \mathcal{C}'[\![A[v']]\!]_{\mathcal{C}'[\![S]\!]}
\end{aligned}$$

(let) Since a is a producer, either $a \in v'$ so that $\mathcal{C}'[\![a]\!]_c = \langle a \mid c\rangle$, or $a : \mathbf{codata}\,T\,\{\dots\}$. Hence, using Lemma B.1 and since x cannot be free in S, we find

$$\begin{aligned}
\mathcal{C}'[\![\mathbf{let}\ x = a;\ p]\!]_{\mathcal{C}'[\![S]\!]} &= \langle \mathcal{C}'[\![a]\!] \mid \tilde{\mu}x.\mathcal{C}'[\![p]\!]_{\mathcal{C}'[\![S]\!]}\rangle \\
&\to_{\tilde{\mu}} (\mathcal{C}'[\![p]\!]_{\mathcal{C}'[\![S]\!]})[x \mapsto \mathcal{C}'[\![a]\!]] \\
&= \mathcal{C}'[\![p[x \mapsto a]]\!]_{\mathcal{C}'[\![S]\!]}
\end{aligned}$$

(call) Using Lemma B.1, and since $\mathcal{C}'[\![\mathbf{def}\ f(\Gamma) : \tau\,\{\,p\,\}]\!] = \mathbf{def}\ f(\Gamma, \alpha :^{\mathrm{cns}} \tau)\,\{\,\mathcal{C}'[\![p]\!]_\alpha\,\}$ with α fresh, we find

$$\begin{aligned}
\mathcal{C}'[\![f(v)]\!]_{\mathcal{C}'[\![S]\!]} &= f(\mathcal{C}'[\![v]\!], \mathcal{C}'[\![S]\!]) \\
&\to_{call} (\mathcal{C}'[\![p]\!]_\alpha)[\Gamma \mapsto v, \alpha \mapsto \mathcal{C}'[\![S]\!]] \\
&= \mathcal{C}'[\![p[\Gamma \mapsto v]]\!]_{\mathcal{C}'[\![S]\!]}
\end{aligned}$$

(add) Let $n \equiv n_1 + n_2$, then

$$\begin{aligned}
\mathcal{C}'[\![n_1 + n_2]\!]_{\mathcal{C}'[\![S]\!]} &= \langle n_1 + n_2 \mid \mathcal{C}'[\![S]\!]\rangle \\
&\to_{add} \langle n \mid \mathcal{C}'[\![S]\!]\rangle \\
&= \mathcal{C}'[\![n]\!]_{\mathcal{C}'[\![S]\!]}
\end{aligned}$$

(ifz)

$$\begin{aligned}
\mathcal{C}'[\![\mathbf{if}\ n \equiv 0\,\{\,p_1\,\}\,\mathbf{else}\,\{\,p_2\,\}]\!]_{\mathcal{C}'[\![S]\!]} &= \mathbf{if}\ n \equiv 0\,\{\,\mathcal{C}'[\![p_1]\!]_{\mathcal{C}'[\![S]\!]}\,\}\,\mathbf{else}\,\{\,\mathcal{C}'[\![p_2]\!]_{\mathcal{C}'[\![S]\!]}\,\} \\
&\to_{ifz} \begin{cases} \mathcal{C}'[\![p_1]\!]_{\mathcal{C}'[\![S]\!]} & \text{if } n = 0 \\ \mathcal{C}'[\![p_2]\!]_{\mathcal{C}'[\![S]\!]} & \text{otherwise} \end{cases}
\end{aligned}$$

(case-r) Using Lemma B.1, and since no (co)variables in Γ can be free in S, we find

$$\begin{aligned}
&\mathcal{C}'[\![K(v).\mathbf{case}\,\{\ldots, K(\Gamma) \Rightarrow p, \ldots\}]\!]_{\mathcal{C}'[\![S]\!]} \\
&= \mathcal{C}'[\![K(v)]\!]_{\mathbf{case}\,\{\ldots, K(\Gamma)\Rightarrow \mathcal{C}'[\![p]\!]_{\mathcal{C}'[\![S]\!]}, \ldots\}} \\
&= \langle K(\mathcal{C}'[\![v]\!]) \mid \mathbf{case}\,\{\ldots, K(\Gamma) \Rightarrow \mathcal{C}'[\![p]\!]_{\mathcal{C}'[\![S]\!]}, \ldots\}\rangle \\
&\to_{case} (\mathcal{C}'[\![p]\!]_{\mathcal{C}'[\![S]\!]})[\Gamma \mapsto \mathcal{C}'[\![v]\!]] \\
&= \mathcal{C}'[\![p[\Gamma \mapsto v]]\!]_{\mathcal{C}'[\![S]\!]}
\end{aligned}$$

(dtor-r) Using Lemma B.1 and Lemma B.4 we find

$$\begin{aligned}
&\mathcal{C}'[\![\mathbf{new}\,\{\ldots, D(\Gamma) \Rightarrow p, \ldots\}.D(v)]\!]_{\mathcal{C}'[\![S]\!]} \\
&= \mathcal{C}'[\![\mathbf{new}\,\{\ldots, D(\Gamma) \Rightarrow p, \ldots\}]\!]_{D(\mathcal{C}'[\![v]\!], \mathcal{C}'[\![S]\!])} \\
&= \langle \mathbf{new}\,\{\ldots, D(\Gamma, \alpha) \Rightarrow \mathcal{C}'[\![p]\!]_\alpha, \ldots\} \mid D(\mathcal{C}'[\![v]\!], \mathcal{C}'[\![S]\!])\rangle \quad (\alpha \text{ fresh}) \\
&\to_{dtor} (\mathcal{C}'[\![p]\!]_\alpha)[\Gamma, \alpha \mapsto \mathcal{C}'[\![v]\!], \mathcal{C}'[\![S]\!]] \\
&= \mathcal{C}'[\![p[\Gamma \mapsto v]]\!]_{\mathcal{C}'[\![S]\!]}
\end{aligned}$$

(label)

$$\mathcal{C}'[\![\mathbf{goto}\,S\,(p)]\!]_{\mathcal{C}'[\![S']\!]} = \mathcal{C}'[\![p]\!]_{\mathcal{C}'[\![S]\!]}$$

(exit) If $p = n$, we have

$$\begin{aligned}
\mathcal{C}'[\![\mathbf{exit}\,n]\!]_{\mathcal{C}'[\![S]\!]} &= \mathbf{exit}\,n \\
&= \mathcal{C}'[\![\langle n \parallel \mathbf{exit}\rangle]\!]
\end{aligned}$$

Otherwise p is not an argument value.
If $p = K(\sigma)$ or $p = p_1 + p_2$ or $p = \mathbf{label}\,\alpha\,\{p_0\}$, then $\mathcal{C}'[\![p]\!]_c = \langle p \mid c\rangle$. Thus, using Lemma B.2, we have

$$\begin{aligned}
\mathcal{C}'[\![\mathbf{exit}\,p]\!]_{\mathcal{C}'[\![S]\!]} &= \mathbf{exit}\,\mathcal{C}'[\![p]\!] \\
&\to_{arg\text{-}p} \langle \mathcal{C}'[\![p]\!] \mid \tilde{\mu}x.\mathbf{exit}\,x\rangle \\
&= \mathcal{C}'[\![p]\!]_{\mathcal{C}'[\![\mathbf{exit}]\!]}
\end{aligned}$$

Otherwise, $\mathcal{C}'[\![p]\!] = \mu\alpha.\mathcal{C}'[\![p]\!]_\alpha$ with α fresh. Thus, using Lemma B.1, we have

$$\begin{aligned}
\mathcal{C}'[\![\mathbf{exit}\,p]\!]_{\mathcal{C}'[\![S]\!]} &= \mathbf{exit}\,\mathcal{C}'[\![p]\!] \\
&\to_{arg\text{-}p} \langle \mathcal{C}'[\![p]\!] \mid \tilde{\mu}x.\mathbf{exit}\,x\rangle \\
&= \langle \mu\alpha.\mathcal{C}'[\![p]\!]_\alpha \mid \mathcal{C}'[\![\mathbf{exit}]\!]\rangle \quad (\alpha \text{ fresh}) \\
&\to_{\mu} \mathcal{C}'[\![p]\!]_{\mathcal{C}'[\![\mathbf{exit}]\!]}
\end{aligned}$$

□

C Typeability Preservation for the Translation from Shrunk Core to AxCut

Proof Sketch for Theorem 7.7. We proceed by induction on the typing derivation. We sketch two cases, the other cases proceed in a similar fashion.

(1) First, we consider the case where a constructor is bound to a variable. We have

$$\dfrac{\dfrac{\mathbf{data}\,T\,\{\ldots, K(\Gamma_0), \ldots\} \in \Theta \qquad \Theta \mid \Gamma \vdash \Gamma_0 \,:\, \Gamma_0}{\Theta \mid \Gamma \vdash K(\Gamma_0) \,:\, T}\,\textsc{Ctor} \qquad \dfrac{\Theta \mid \Gamma, x :^{\mathrm{prd}} T \vdash s}{\Theta \mid \Gamma \vdash \tilde{\mu}x.s :^{\mathrm{cns}} T}\,\textsc{Act-L}}{\Theta \mid \Gamma \vdash \langle K(\Gamma_0) \mid \tilde{\mu}x.s\rangle}\,\textsc{Cut}$$

The translation of the statement is

$$\textbf{substitute}[\Gamma' \coloneqq \Gamma', \Gamma_0^f \coloneqq \Gamma_0];\ \textbf{let}\, x = K(\Gamma_0^f);\ \mathcal{X}[\![s]\!]_{\Gamma',x}$$

where $\Gamma' = \mathsf{freeVars}(s) \subset \Gamma$. Since Γ' contains the free variables of the statement s, the context Γ in the subderivation for this statement can be strengthened. As the order can be exchanged arbitrarily in **Shrunk Core**, we obtain

$$\Theta \mid \Gamma', x :^{\mathsf{prd}} T \vdash s$$

and the induction hypothesis yields

$$\mathcal{X}[\![\Theta]\!] \mid \Gamma', x :^{\mathsf{prd}} T \vdash \mathcal{X}[\![s]\!]_{\Gamma',x}$$

From rule Let-**data** we hence find

$$\mathcal{X}[\![\Theta]\!] \mid \Gamma', \Gamma_0^f \vdash \textbf{let}\, x = K(\Gamma_0^f);\ \mathcal{X}[\![s]\!]_{\Gamma',x}$$

Since $\Gamma_0 \subset \Gamma$ and $\Gamma' \subset \Gamma$ and because Γ_0^f contains the same types as Γ_0, we get

$$\Gamma \vdash \Gamma', \Gamma_0 \,:\, \Gamma', \Gamma_0^f$$

and rule Substitute thus yields

$$\mathcal{X}[\![\Theta]\!] \mid \Gamma \vdash \textbf{substitute}[\Gamma' \coloneqq \Gamma', \Gamma_0^f \coloneqq \Gamma_0];\ \textbf{let}\, x = K(\Gamma_0^f);\ \mathcal{X}[\![s]\!]_{\Gamma',x}$$

(2) Now consider the case where a copattern match is bound to a variable. We have

$$\dfrac{\dfrac{\textbf{codata}\, T\,\{D_1(\Gamma_1), \ldots\} \in \Theta \qquad \forall i \,:\, \Theta \mid \Gamma, \Gamma_i \vdash s_i}{\Theta \mid \Gamma \vdash \textbf{new}\,\{D_1(\Gamma_1) \Rightarrow s_1, \ldots\} :^{\mathsf{prd}} T}\ \textsc{New} \qquad \dfrac{\Theta \mid \Gamma, x :^{\mathsf{prd}} T \vdash s}{\Theta \mid \Gamma \vdash \tilde{\mu} x.s :^{\mathsf{cns}} T}\ \textsc{Act-L}}{\Theta \mid \Gamma \vdash \langle \textbf{new}\,\{D_1(\Gamma_1) \Rightarrow s_1, \ldots\} \mid \tilde{\mu} x.s \rangle}\ \textsc{Cut}$$

The translation of the statement is

$$\begin{aligned}&\textbf{substitute}[\Gamma'^f \coloneqq \Gamma', \Gamma_0 \coloneqq \Gamma_0];\\ &\textbf{create}\, x = \Gamma_0\,\{D_1(\Gamma_1) \Rightarrow \mathcal{X}[\![s_1]\!]_{\Gamma_1,\Gamma_0}, \ldots\};\ \mathcal{X}[\![s[\Gamma' \mapsto \Gamma'^f]]\!]_{\Gamma'^f,x}\end{aligned}$$

where $\Gamma_0 = \bigcup_i \mathsf{freeVars}(s_i) \subset \Gamma$ and $\Gamma' = \mathsf{freeVars}(s) \subset \Gamma$. Since Γ_0 contains the free variables of the statements in all branches, the context Γ in the subderivations for these statements can again be strengthened. As the order can be exchanged arbitrarily in **Shrunk Core**, we obtain

$$\forall i \,:\, \Theta \mid \Gamma_i, \Gamma_0 \vdash s_i$$

and the induction hypothesis yields

$$\forall i \,:\, \mathcal{X}[\![\Theta]\!] \mid \Gamma_i, \Gamma_0 \vdash \mathcal{X}[\![s_i]\!]_{\Gamma_i,\Gamma_0}$$

Additionally, we again obtain

$$\Theta \mid \Gamma', x :^{\mathsf{prd}} T \vdash s$$

and the induction hypothesis yields

$$\mathcal{X}[\![\Theta]\!] \mid \Gamma', x :^{\mathsf{prd}} T \vdash \mathcal{X}[\![s]\!]_{\Gamma',x}$$

We can show that renaming the variables in a context Γ_0 to Γ_0' satisfies

$$\mathcal{X}[\![s]\!]_{\Gamma}[\Gamma_0 \mapsto \Gamma_0'] = \mathcal{X}[\![s[\Gamma_0 \mapsto \Gamma_0']]\!]_{\Gamma[\Gamma_0 \mapsto \Gamma_0']}$$

where the renaming also renames left-hand sides in explicit substitutions and we assume that no variable is shadowed. Hence, we get from the above that

$$\mathcal{X}[\![\Theta]\!] \mid \Gamma'^{f}, x :^{\mathrm{prd}} T \vdash \mathcal{X}[\![s[\Gamma' \mapsto \Gamma'^{f}]]\!]_{\Gamma'^{f},x}$$

By rule Create-**codata** we thus find

$$\mathcal{X}[\![\Theta]\!] \mid \Gamma'^{f}, \Gamma_0 \vdash \mathbf{create}\, x = \Gamma_0 \{ D_1(\Gamma_1) \Rightarrow \mathcal{X}[\![s_1]\!]_{\Gamma_1,\Gamma_0}, \dots \};\ \mathcal{X}[\![s[\Gamma' \mapsto \Gamma'^{f}]]\!]_{\Gamma'^{f},x}$$

Since $\Gamma_0 \subset \Gamma$ and $\Gamma' \subset \Gamma$ and because Γ'^{f} contains the same types as Γ', we get

$$\Gamma \vdash \Gamma', \Gamma_0 \ :\ \Gamma'^{f}, \Gamma_0$$

and rule Substitute thus yields

$$\mathcal{X}[\![\Theta]\!] \mid \Gamma \vdash \mathbf{substitute}[\Gamma'^{f} \coloneqq \Gamma', \Gamma_0 \coloneqq \Gamma_0];\ \mathbf{create}\, x = \Gamma_0 \{ D_1(\Gamma_1) \Rightarrow \mathcal{X}[\![s_1]\!]_{\Gamma_1,\Gamma_0}, \dots \};\ \mathcal{X}[\![s[\Gamma' \mapsto \Gamma'^{f}]]\!]_{\Gamma'^{f},x}$$

□