Compiling Classical Sequent Calculus to Stock Hardware: The Duality of Compilation

Philipp Schuster
University of Tübingen, Germany

Marius Müller
University of Tübingen, Germany

Klaus Ostermann
University of Tübingen, Germany

Jonathan Immanuel Brachthäuser
University of Tübingen, Germany

Abstract

Compiler intermediate representations have to strike a balance between being high-level enough to allow for easy translation of surface languages into them and being low-level enough to make code generation easy. An intermediate representation based on a logical system typically has the former property and additionally satisfies several meta-theoretical properties which are valuable when optimizing code. Recently, classical sequent calculus, which is widely recognized and impactful within the fields of logic and proof theory, has been proposed as a natural candidate in this regard, due to its symmetric treatment of data and control flow. For such a language to be useful, however, it must eventually be compiled to machine code. In this paper, we introduce an intermediate representation that is based on classical sequent calculus and demonstrate that this language can be directly translated to conventional hardware. We provide both a formal description and an implementation. Preliminary performance evaluations indicate that our approach is viable.

Main Reference Philipp Schuster, Marius Müller, Klaus Ostermann, and Jonathan Immanuel Brachthäuser. 2025. Compiling Classical Sequent Calculus to Stock Hardware: The Duality of Compilation. Proc. ACM Program. Lang. 9, OOPSLA1, Article 142 (April 2025), 28 pages. https://doi.org/10.1145/3720507

Comments This is an extended version of the main reference. Compared to the published paper, this report contains the full appendix: typing rules for the states of the abstract machine of AxCut, full formal definition of the translation from AxCut to machine code, and detailed results and discussion of the benchmarks.

1 Introduction

Every compiler intermediate representation exists in a force field between high-level source languages and low-level target languages. On the one hand, it should be close to a high-level language and come equipped with a reduction semantics that enables equational reasoning and semantics-preserving rewrites. Ideally, it should be typed and safe, typically manifested as theorems of progress and preservation. This is usually achieved through compound language constructs that make invariants hold by construction. On the other hand, it should be close to low-level machine code and come equipped with an operational semantics, which yields an accurate cost model for execution on an actual machine. This is usually achieved by having simple but unsafe language constructs.

Historically, many programming languages and their intermediate representations have been designed and optimized to excel at synchronous computation: inputs are available at the start of the program and outputs are available upon termination. However, modern programs are more and more interactive. Not only do they continually interact with the outside world, through input and output devices, but also with special computation devices which perform computation-intensive tasks. In both cases, to avoid unresponsive programs, computation needs to be suspended without blocking other computations from being performed. In

Philipp Schuster, Marius Müller, Klaus Ostermann, and Jonathan Immanuel Brachthäuser. "Compiling Classical Sequent Calculus to Stock Hardware: The Duality of Compilation". Technical Report. 2025. *University of Tübingen, Germany.*

other words, communication with these devices should be asynchronous. Moreover, this asynchronous communication has to be reconciled with other control effects like exceptions and generators.

To implement support for these control effects, compiler developers currently mostly choose between two options: encoding asynchronous computation in terms of low-level features, for example, as a state machine [5], or adding runtime support for a high-level control operator, for example, a form of stack switching [37]. Both of these options are not fully satisfactory. The encoding as a state machine is a leaky abstraction and causes the duplication of higher-order functions either by the user or the compiler. Moreover, it leads to unsafe low-level code which violates invariants guaranteed by high-level constructs. On the other hand, additional runtime constructs complicate the cost model and equational reasoning, and are often unsafe, merely establishing invariants at run time. Both of these options hard-code a single specific form of control effect and the interaction with other control effects has to be carefully considered.

An attractive alternative to traditional intermediate representations is offered by modern term assignments for classical sequent calculus LK [20], as argued recently [6]. These term assignments have two symmetric syntactic categories of producers p (variously called programs, terms, proofs, etc.) and consumers c (variously called contexts, coterms, refutations, etc.), and a third syntactic category of statements s (variously called commands, etc.) [10, 44, 50, 12]. Producers, which construct an element of some type, are a familiar concept present in many programming languages. Consumers, in contrast, are not typically found as first-class entities in programming languages. They destruct an element of some type, and one can think of them as continuations or program contexts. Finally, statements are cuts between a producer and a consumer of the same type T.

$$\frac{\Gamma \vdash p : \mathbf{prd}\ T \qquad \Gamma' \vdash c : \mathbf{cns}\ T}{\Gamma,\ \Gamma' \vdash \langle\, p \mid c \,\rangle}\ [\textsc{Cut}]$$

When a producer and a consumer of the same type meet in a cut, they interact. Operationally, cuts correspond to redexes, but do not return a result and thus do not have a return type. For example, for a producer of an algebraic data type, *i.e.*, a constructor, and a consumer of the same type, *i.e.*, a pattern match, the resulting cut can be reduced to the statement of the clause for the constructor. This rewriting only terminates when control is yielded to an external authority. From a logical perspective, cuts correspond to contradictions where a proof and a refutation of the same proposition meet. Rewriting is done to eliminate cuts and only terminates when a *daimon* [22] is reached. Computation hence corresponds to cut elimination.

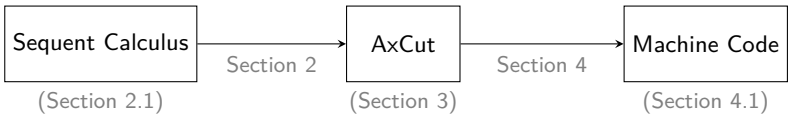
The symmetry between producers and consumers [13, 33] is what makes these term assignments particularly well-suited to the expression of control effects without the need for a low-level encoding or support from a runtime system. Consumers are first-class, as they can be named, stored, duplicated, and dropped. The resulting logical system is classical and thereby more expressive than intuitionistic logic. It allows us to derive classical reasoning principles such as the law of excluded middle or Peirce's law [44, 14], which correspond to particular control operators [23].

Moreover, in classical sequent calculus, every logical connective naturally comes in two polarities: positive and negative [15, 16, 12], which mediate between call-by-value and call-by-name [50]. 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. Finally, it straightforwardly supports linear logic connectives [21], which facilitates reasoning about

resources. All of these features make classical sequent calculus a natural basis for a high-level compiler intermediate representation for programs with control effects.

Eventually, however, programs are run on real computers and a compiler intermediate representation should lend itself to generation of code for existing hardware. Such hardware typically does not exhibit the high-level structure of a logical system. Registers could contain anything at any point, memory is an unstructured sequence of bytes, and code is an unstructured sequence of instructions. How is it possible to bridge this gap between theoretical logical systems and practical computer hardware?

We present *AxCut*, an intermediate representation that serves as a bridge between high-level logical proofs and low-level machine code. On the one hand, it is a fragment of classical sequent calculus and enjoys safety in the form of progress and preservation. On the other hand, it admits a direct translation to machine code and has a small-step operational semantics with an abstract machine. We demonstrate how the concepts of logic can be used to impose invariants, giving structure to machine code, registers, and memory. The below figure gives an overview of our compilation pipeline.



Because it is based on classical sequent calculus, AxCut excels at expressing control effects and interaction protocols, which we illustrate next.

1.1 Control Effects

Consider the following example in classical sequent calculus, taken from Binder et al. [6], starting where they left off. We define a function mult, that multiplies a list of numbers. When we encounter an element that is zero we abort and immediately return zero as the overall result.

```
define mult(k : cns Cont, l : prd List) = loop(k, k, l)

define loop(h : cns Cont, k : cns Cont, l : prd List) = l case {
  nil() ⇒ k.ret(1)
  cons(x, xs) ⇒ if (x ≡ 0) { h.ret(0) } else { loop(h, case { ret(z) ⇒ k.ret(x * z) }, xs) }
}
```

The implementation is split into a wrapper mult and a worker loop. Function loop receives the list l and two contexts k (the return continuation) and h (the exception continuation or "handler"). In case the list is empty, we immediately return 1 to k. In case the element is 0, we abort the computation by returning 0 to the handler h. Otherwise, we recurse with a larger context to remember that we have to multiply the element with the result. In the wrapper function mult, we duplicate the calling context k and pass it both as the return continuation and the handler to loop.

Existing intermediate representations typically represent this example with a special language construct for handling and throwing exceptions, requiring special runtime support. In contrast, in classical sequent calculus, this form of control effect is expressible directly. We translate this example to AxCut, and from there directly generate the following machine code for the highlighted parts.

```
mult:
  SHARE x4
  mv x8 x6
  mv x9 x7
  mv x6 x4
  mv x7 x5
  j loop
```

```
nil:
  li x9 1
  ERASE x4
  mv x5 x9
  jr x7 0
```

```
then:
  li x13 0
  ERASE x10
  ERASE x6
  mv x6 x4
  mv x7 x5
  mv x5 x13
  jr x7 0
```

```
ret:
  RELEASE x6
  lw x9 20 x6
  lw x7 12 x6
  lw x6 8 x6
  mul x11 x9 x5
  mv x5 x11
  jr x7 0
```

The registers contain the variables, with two registers per variable. In `mult`, we duplicate the context k which amounts to sharing its associated memory. In `nil` and `then`, we drop one of the contexts, which amounts to erasing its associated memory. Sharing and erasing manipulate reference counts. In `ret`, we restore a context, which amounts to loading values corresponding to its environment from memory. The details will become clear in the course of the paper. We do not use any runtime system. There is no difference between throwing an exception and an ordinary return, both are constant-time operations as there is no unwinding. This example, chosen for familiarity, demonstrates that in AxCut exceptions are one expressible form of control effect. However, other less familiar forms are expressible too, as we demonstrate next.

1.2 Interaction Protocols

Algebraic data types [8] describe invariants on the shape of data. The slogan that popularizes them is "make illegal states unrepresentable". Their logical dual, algebraic codata types [24, 39], are much less well known [18]. They describe the observable behavior of processes over time and their slogan could be "make illegal actions unperformable". While these processes do not necessarily terminate, they continually interact with each other and with the outside world.

As an example of codata types, consider the following two mutually recursive declarations. Each of them defines two destructors, respectively stop and push, and done and pull.

```
codata Receiver {
  stop(),
  push(byte : ext Byte, rest : prd Sender)
}
```

```
codata Sender {
  done(),
  pull(next : prd Receiver)
}
```

When we push a byte, we yield control to the receiver, describing the rest of the sending behavior. Dually, when we pull, we describe the next receiving behavior. Together, they specify the protocol of interaction between a sender and a receiver of bytes. Potentially infinite processes like these are best modeled with codata types, which naturally follow a call-by-name evaluation order [15]. Values of codata type must be prepared for exactly the listed destructors. We create them with **cocase**, describing the behavior for each destructor. In the following, we create a sender zeroes, which always responds with byte 0, and a receiver discard, which discards all bytes pushed into it.

```
define zeroes : prd Sender = cocase {
  done() ⇒ exit,
  pull(next) ⇒ next.push(0, zeroes)
}
```

```
define discard : prd Receiver = cocase {
  stop() ⇒ exit,
  push(byte, rest) ⇒ rest.pull(discard)
}
```

In the definition of zeroes, when a byte is pulled, we push a 0 and continue to behave as zeroes. In the definition of discard, when a byte is pushed, we ignore it and continue to

behave as discard. Both of them exit the program upon completion, returning control to the operating system.

In this paper, we translate this example of communicating processes to AxCut, and from there directly generate the following machine code.

```
zeroes:
  j done
  j pull
done:
  EXIT
pull:
  j zeroes_next
```

```
zeroes_next:
  li x6 null
  la x7 zeroes
  li x9 0
  SWAP x4 x8
  SWAP x5 x9
  jr x9 4
```

```
discard:
  j stop
  j push
stop:
  EXIT
push:
  j discard_rest
```

```
discard_rest:
  li x8 null
  la x9 discard
  mv x4 x8
  mv x5 x9
  jr x7 4
```

We connect a sender and a receiver, by pulling from the former with the latter. They will coroutine between each other, until one of them is finished, a potentially infinite process.

define connect(r : **prd** Receiver, s : **prd** Sender) = s.pull(r)

```
connect:
  jr x7 4
```

In machine code, connecting the two amounts to an indirect jump. 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. A context switch amounts to some register moves and an indirect jump to the other process.

1.3 Contributions

To summarize, we make the following contributions:

- We identify a fragment of the term assignment for sequent calculus with algebraic data and codata types which is suitable for compilation to machine code.
- Terms in this fragment can be considered to be in a certain normal form and we show how to transform classical sequent calculus proofs into this normal form.
- We add several practical extensions to the fragment, guided by ideas from logic, to obtain the compiler intermediate representation AxCut.
- We define the operational semantics for AxCut in terms of an abstract machine that enjoys type safety in the form of standard theorems for progress (1) and preservation (2).
- We present direct generation of executable machine code from AxCut, without the need for a runtime system.
- We show how the design of AxCut facilitates a real-time garbage collection technique with a fast path for linear usage.

In Section 2, we go from a standard sequent calculus to AxCut. In Section 3, we formally introduce AxCut. In Section 4, we go from AxCut to machine code. Then, we discuss implementation details and present benchmarks in Section 5. We compare with related work in Section 6 and finally conclude in Section 7.

2 From Sequent Calculus to AxCut

In this section, we start with a term assignment for classical sequent calculus and then show how to arrive at a normal form that allows for a direct translation to machine code. We call the resulting system AxCut. For a source language in direct style, we would of course need a translation to sequent calculus as a first step. This is somewhat comparable to a translation to continuation-passing style.

Syntax

Producers	p	$::=$	x	Consumers	c	$::=$	α
		$\mid$	$\mu\alpha.s$			$\mid$	$\tilde{\mu}x.s$
		$\mid$	$C(e,\ ...)$			$\mid$	$D(e,\ ...)$
		$\mid$	**cocase** $\{\ D(\Gamma) \Rightarrow s,\ ...\ \}$			$\mid$	**case** $\{\ C(\Gamma) \Rightarrow s,\ ...\ \}$
Statements	s	$::=$	$\langle\ p \mid c\ \rangle$	Expressions	e	$::=$	$p \mid c$
Constructors	C	$::=$	nil $\mid$ cons $\mid$ ...	(Co)Variables	v	$::=$	$x \mid \alpha \mid$ ...
Destructors	D	$::=$	apply $\mid$ first $\mid$ ...	Chirality	χ	$::=$	**prd** $\mid$ **cns**
Type Names	T	$::=$	List $\mid$ Func $\mid$ ...	Types	τ	$::=$	$\chi\ T$
Type Definitions	Σ	$::=$	**data** $T\ \{\ C(\Gamma),\ ...\ \}$	Type Environment	Γ	$::=$	$v : \tau,\ ...$
		$\mid$	**codata** $T\ \{\ D(\Gamma),\ ...\ \}$				

Figure 1 Syntax for a sequent calculus with data and codata.

2.1 A Standard Sequent Calculus

Figure 1 defines the terms for a classical sequent calculus, following **Core** of Binder et al. [6] and System $\mathcal{CD}$ of Downen and Ariola [15]. As usual, we distinguish between producers and consumers. Producers are variables, μ-bindings, constructors of data types and copattern matches for codata types [50, 2], denoted with **cocase**. Dually, consumers are covariables, denoted by lower-case Greek letters, $\tilde{\mu}$-bindings, destructors of codata types and pattern matches for data types, denoted with **case**. Computation happens when a producer and a consumer of the same type meet in a cut, which is the only statement. The $\tilde{\mu}$- and μ-bindings allow us to abstract over a producer or a consumer in a statement. If we want to refer to an expression, which can be either a producer or a consumer, we denote it with e, and similarly, we write v to stand for either a variable or a covariable.

Notationally, we follow Binder et al. [6] in that we collect all variables and covariables in one typing context Γ instead of partitioning them into two separate contexts Γ and Δ on the left- and right-hand side of the turnstile. We therefore annotate whether a binding is a producer or a consumer with **prd** or **cns**, respectively. We call this the *chirality* of the expression, which is derived from the Greek word for hand. A type is either **prd** T or **cns** T, where T is a name.

Figure 2 defines the typing rules for this sequent calculus. We have typing judgments for producers and consumers, $\Gamma \vdash p : \textbf{prd}\ T$ and $\Gamma \vdash c : \textbf{cns}\ T$. Producer typing includes the usual axiom (Ax-R) and activation (Act-R) rules for variables and μ-bindings, as well as the right introduction rules for data (Data-R) and codata (Codata-R), *i.e.*, constructors and copattern matches. A constructor C of data type T is well-typed if all its arguments are well-typed according to the signature of the constructor. For a copattern match of a codata type to be well-typed, there has to be one well-typed statement for each branch, *i.e.*, for each destructor. The rules for consumer typing are exactly dual to those for producer typing. The typing rule for cut is standard. Note that all typing rules are formulated such that they conform to linear typing. We will come back to this point later in this section.

2.2 Finding a Suitable Normal Form

Starting from this standard sequent calculus, we identify a fragment AxCut, which can be considered a normal form suitable for direct code generation. We first give a name

Producer Typing $\boxed{\Gamma \vdash p : \textbf{prd}\ T}$

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

$$\frac{\Sigma(T) = \{\ ...,\ C(v_1 : \tau_1,\ ...),\ ...\ \} \qquad \Gamma_1 \vdash e_1 : \tau_1 \quad ...}{\Gamma_1,\ ... \vdash C(e_1,\ ...) : \textbf{prd}\ T}\ [\textsc{Data-R}]$$

$$\frac{\Sigma(T) = \{\ D_1(\Gamma_1),\ ...\ \} \qquad \Gamma,\ \Gamma_1 \vdash s_1 \quad ...}{\Gamma \vdash \textbf{cocase}\ \{\ D_1(\Gamma_1) \Rightarrow s_1,\ ...\ \} : \textbf{prd}\ T}\ [\textsc{Codata-R}]$$

Consumer Typing $\boxed{\Gamma \vdash c : \textbf{cns}\ T}$

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

$$\frac{\Sigma(T) = \{\ ...,\ D(v_1 : \tau_1,\ ...),\ ...\ \} \qquad \Gamma_1 \vdash e_1 : \tau_1 \quad ...}{\Gamma_1,\ ... \vdash D(e_1,\ ...) : \textbf{cns}\ T}\ [\textsc{Codata-L}]$$

$$\frac{\Sigma(T) = \{\ C_1(\Gamma_1),\ ...\ \} \qquad \Gamma,\ \Gamma_1 \vdash s_1 \quad ...}{\Gamma \vdash \textbf{case}\ \{\ C_1(\Gamma_1) \Rightarrow s_1,\ ...\ \} : \textbf{cns}\ T}\ [\textsc{Data-L}]$$

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

$$\frac{\Gamma \vdash p : \textbf{prd}\ T \qquad \Gamma' \vdash c : \textbf{cns}\ T}{\Gamma,\ \Gamma' \vdash \langle\ p \mid c\ \rangle}\ [\textsc{Cut}]$$

Figure 2 Typing for a sequent calculus with data and codata.

to all subterms, then we shrink the language by removing redundant constructs. While the description of this normalization procedure $\mathcal{N}[\![\ \cdot\ ]\!]$ in this section is semi-formal, we have implemented it on intrinsically-typed terms in Idris 2 [7], which consequently shows typeability preservation. It will be obvious that all normalization steps are instances of β- or η-equalities so that the normalization procedure is clearly semantics-preserving with respect to the standard substitution-based operational semantics.

Notably, the whole normalization procedure can be done in a single pass, which is compositional in the sense that recursive calls are only done on subterms, and in fact only once for each subterm. This is demonstrated by our implementation in Idris 2. The idea behind this efficient transformation is similar to the standard one-pass continuation-passing style translation. We have chosen not to present it in this way here, to make it less technical and more comprehensible.

2.2.1 Naming Subterms

Producers and consumers can occur either directly in cuts or as nested arguments of constructors and destructors. In AxCut, we demand that all arguments of constructors and destructors are variables, a restriction similar to A-normal form [19]. To express programs under this restriction, we use μ- and $\tilde{\mu}$-bindings to lift producers and consumers out of argument positions. For example, suppose that C is a constructor with a producer argument p, which is not a variable, and $C(p)$ is cut with some consumer c.

$$\mathcal{N}[\![\ \langle\ C(p) \mid c\ \rangle\ ]\!] = \langle\ p \mid \tilde{\mu}x.\langle\ C(x) \mid c\ \rangle\ \rangle$$

The producer p is bound to variable x using the $\tilde{\mu}$-binding so that the argument of C becomes a variable. More generally, we use the following transformation rules to give a name to each non-variable subexpression.

$$
\begin{array}{lcll}
\mathcal{N}[\![\langle\, C(..., \ p, \ ...) \mid c_0 \,\rangle]\!] & = & \mathcal{N}[\![\langle\, p \mid \tilde{\mu}x.\mathcal{N}[\![\langle\, C(..., \ x, \ ...) \mid c_0 \,\rangle]\!] \,\rangle]\!] & \text{where } x \text{ is fresh} \\
\mathcal{N}[\![\langle\, C(..., \ c, \ ...) \mid c_0 \,\rangle]\!] & = & \mathcal{N}[\![\langle\, \mu\alpha.\mathcal{N}[\![\langle\, C(..., \ \alpha, \ ...) \mid c_0 \,\rangle]\!] \mid c \,\rangle]\!] & \text{where } \alpha \text{ is fresh} \\
\mathcal{N}[\![\langle\, p_0 \mid D(..., \ p, \ ...) \,\rangle]\!] & = & \mathcal{N}[\![\langle\, p \mid \tilde{\mu}x.\mathcal{N}[\![\langle\, p_0 \mid D(..., \ x, \ ...) \,\rangle]\!] \,\rangle]\!] & \text{where } x \text{ is fresh} \\
\mathcal{N}[\![\langle\, p_0 \mid D(..., \ c, \ ...) \,\rangle]\!] & = & \mathcal{N}[\![\langle\, \mu\alpha.\mathcal{N}[\![\langle\, p_0 \mid D(..., \ \alpha, \ ...) \,\rangle]\!] \mid c \,\rangle]\!] & \text{where } \alpha \text{ is fresh}
\end{array}
$$

In these rules, we assume that the arguments p or c, respectively, are the first non-variable arguments. While the above transformation rules include the well-known rules for static focusing [3, 10], we additionally lift subterms that are values.

In typing derivations, this *variable restriction* enforces that all subderivations of right rules for data types and left rules for codata types have to be axioms. Here h stands for L or R.

$$
\frac{\Sigma(T) = \{ \ ..., \ C(v_1 : \tau_1, \ ...), \ ... \ \} \qquad \dfrac{}{v_1 : \tau_1 \vdash v_1 : \tau_1}\,[\text{Ax-}h] \qquad ...}{v_1 : \tau_1, \ ... \vdash C(v_1, \ ...) : \mathbf{prd}\ T}\,[\textsc{Data}\text{-R}]
$$

$$
\frac{\Sigma(T) = \{ \ ..., \ D(v_1 : \tau_1, \ ...), \ ... \ \} \qquad \dfrac{}{v_1 : \tau_1 \vdash v_1 : \tau_1}\,[\text{Ax-}h] \qquad ...}{v_1 : \tau_1, \ ... \vdash D(v_1, \ ...) : \mathbf{cns}\ T}\,[\textsc{Codata}\text{-L}]
$$

2.2.2 Shrinking the Language

Since there are four different syntactic alternatives for both producers and consumers, there are a total of 16 different forms of statements. Two of them, a constructor meeting a destructor and a pattern match meeting a copattern match, cannot occur due to typing. This leaves us with the following list of 14 statement forms. Since subexpressions of constructors and destructors now have to be variables, we write these lists as Γ, indicating that they are applied to a part of the environment. Next, we show how to remove the lower six forms.

$$
\begin{array}{llcll}
\text{Statements} & s & ::= & \langle\, C(\Gamma) \mid \tilde{\mu}x.s \,\rangle & \mid \ \langle\, \mu\alpha.s \mid D(\Gamma) \,\rangle \\
 & & \mid & \langle\, x \mid \mathbf{case}\ \{\ C(\Gamma) \Rightarrow s, \ ...\ \} \,\rangle & \mid \ \langle\, \mathbf{cocase}\ \{\ D(\Gamma) \Rightarrow s, \ ...\ \} \mid \alpha \,\rangle \\
 & & \mid & \langle\, \mu\alpha.s \mid \mathbf{case}\ \{\ C(\Gamma) \Rightarrow s, \ ...\ \} \,\rangle & \mid \ \langle\, \mathbf{cocase}\ \{\ D(\Gamma) \Rightarrow s, \ ...\ \} \mid \tilde{\mu}x.s \ \rangle \\
 & & \mid & \langle\, C(\Gamma) \mid \alpha \,\rangle & \mid \ \langle\, x \mid D(\Gamma) \,\rangle \\
 & & \mid & \langle\, x \mid \alpha \,\rangle & \mid \ \langle\, \mu\alpha.s \mid \tilde{\mu}x.s \,\rangle \\
 & & \mid & \langle\, C(\Gamma) \mid \mathbf{case}\ \{\ C(\Gamma) \Rightarrow s, \ ...\ \} \,\rangle & \mid \ \langle\, \mathbf{cocase}\ \{\ D(\Gamma) \Rightarrow s, \ ...\ \} \mid D(\Gamma) \,\rangle \\
 & & \mid & \langle\, x \mid \tilde{\mu}y.s \,\rangle & \mid \ \langle\, \mu\beta.s \mid \alpha \,\rangle
\end{array}
$$

Removing renaming The two forms in the last row cut a variable with a binding, renaming variables unnecessarily. To transform them away, we simply reduce the cuts by substitution:

$$
\mathcal{N}[\![\langle\, x \mid \tilde{\mu}y.s \,\rangle]\!] \ = \ \mathcal{N}[\![\ s[y \mapsto x] \ ]\!] \qquad\qquad \mathcal{N}[\![\langle\, \mu\beta.s \mid \alpha \,\rangle]\!] \ = \ \mathcal{N}[\![\ s[\beta \mapsto \alpha] \ ]\!]
$$

Removing completely known cuts The two forms in the second-to-last row are completely known cuts. We eliminate them again by reduction, substituting the arguments for the variables bound in the corresponding clause. This substitution is a renaming that never introduces new known cuts.

$$
\begin{array}{lcl}
\mathcal{N}[\![\langle\, C(\Gamma_0) \mid \mathbf{case}\ \{ \ ..., \ C(\Gamma) \Rightarrow s, \ ...\ \} \,\rangle]\!] & = & \mathcal{N}[\![\ \ s[\Gamma \mapsto \Gamma_0] \ ]\!] \\
\mathcal{N}[\![\langle\, \mathbf{cocase}\ \{ \ ..., \ D(\Gamma) \Rightarrow s, \ ...\ \} \mid D(\Gamma_0) \,\rangle]\!] & = & \mathcal{N}[\![\ \ s[\Gamma \mapsto \Gamma_0] \ ]\!]
\end{array}
$$

Removing completely unknown cuts Next, we consider cuts where both sides are variables. We assume a call-by-value evaluation order for data types and a call-by-name evaluation order for codata types, which ensures that all η-laws hold for both kinds of types [6, 15]. We transform away completely unknown cuts by η-expansion. For cuts at data types we expand the consumer variable and for cuts at codata types we expand the producer variable, based on their set of constructors and destructors, respectively.

$$\mathcal{N}[\![\langle\, x \mid \alpha \,\rangle_T]\!] = \langle\, x \mid \textbf{case}\ \{\ C_1(\Gamma_1) \Rightarrow \langle\, C_1(\Gamma_1) \mid \alpha \,\rangle,\ \ldots\ \} \,\rangle \quad \text{where } \textbf{data}\ T\ \{\ C_1(\Gamma_1),\ \ldots\ \}$$

$$\mathcal{N}[\![\langle\, x \mid \alpha \,\rangle_T]\!] = \langle\, \textbf{cocase}\ \{\ D_1(\Gamma_1) \Rightarrow \langle\, x \mid D_1(\Gamma_1) \,\rangle,\ \ldots\ \} \mid \alpha \,\rangle \quad \text{where } \textbf{codata}\ T\ \{\ D_1(\Gamma_1),\ \ldots\}$$

Removing critical pairs Finally, we consider critical pairs $\langle\, \mu\alpha.s_1 \mid \tilde{\mu}x.s_2 \,\rangle$ [10]. They arise during focusing and their semantics determines the evaluation order. We again use η-expansion, for data types on the $\tilde{\mu}$-binding and for codata types on the μ-binding. Again, since data types follow call-by-value and codata types follow call-by-name, all η-laws hold.

$$\mathcal{N}[\![\langle\, \mu\alpha.s_1 \mid \tilde{\mu}x.s_2 \,\rangle_T]\!] = \langle\, \mu\alpha.\mathcal{N}[\![s_1]\!] \mid \textbf{case}\ \{\ C_1(\Gamma_1) \Rightarrow \langle\, C_1(\Gamma_1) \mid \tilde{\mu}x.\mathcal{N}[\![s_2]\!] \,\rangle,\ \ldots\ \} \,\rangle$$
where $\textbf{data}\ T\ \{\ C_1(\Gamma_1),\ \ldots\ \}$

$$\mathcal{N}[\![\langle\, \mu\alpha.s_1 \mid \tilde{\mu}x.s_2 \,\rangle_T]\!] = \langle\, \textbf{cocase}\ \{\ D_1(\Gamma_1) \Rightarrow \langle\, \mu\alpha.\mathcal{N}[\![s_1]\!] \mid D_1(\Gamma_1) \,\rangle,\ \ldots\ \} \mid \tilde{\mu}x.\mathcal{N}[\![s_2]\!] \,\rangle$$
where $\textbf{codata}\ T\ \{\ D_1(\Gamma_1),\ \ldots\}$

Collapsing the data-codata symmetry This leaves us with the following eight statements, where in each of them one side of the cut is a constructor, destructor, pattern match, or copattern match and the other side is either a variable, a μ-, or a $\tilde{\mu}$-binding.

$$\begin{array}{lrlll}
\text{Statements} & s & ::= & \langle\, C(\Gamma) \mid \tilde{\mu}x.s \,\rangle & \mid\ \langle\, \mu\alpha.s \mid D(\Gamma) \,\rangle \\
& & \mid & \langle\, x \mid \textbf{case}\ \{\ C(\Gamma) \Rightarrow s,\ \ldots\ \} \,\rangle & \mid\ \langle\, \textbf{cocase}\ \{\ D(\Gamma) \Rightarrow s,\ \ldots\ \} \mid \alpha \,\rangle \\
& & \mid & \langle\, \mu\alpha.s \mid \textbf{case}\ \{\ C(\Gamma) \Rightarrow s,\ \ldots\ \} \,\rangle & \mid\ \langle\, \textbf{cocase}\ \{\ D(\Gamma) \Rightarrow s,\ \ldots\ \} \mid \tilde{\mu}x.s \,\rangle \\
& & \mid & \langle\, C(\Gamma) \mid \alpha \,\rangle & \mid\ \langle\, x \mid D(\Gamma) \,\rangle
\end{array}$$

For the typing rules, this means that in every cut one of the subderivations is an introduction rule and the other subderivation is either an activation or axiom rule. Cuts with an axiom rule are also known as deactivation rules [11]. Thus, all typing derivations are of the following forms, where j stands for Data or Codata, and k stands for Ax or Act.

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash p : \textbf{prd}\ T}\,[j\text{-R}] \qquad \dfrac{\vdots}{\Gamma' \vdash c : \textbf{cns}\ T}\,[k\text{-L}]}{\Gamma, \Gamma' \vdash \langle\, p \mid c \,\rangle}\,[\text{Cut}] \qquad\qquad \dfrac{\dfrac{\vdots}{\Gamma \vdash p : \textbf{prd}\ T}\,[k\text{-R}] \qquad \dfrac{\vdots}{\Gamma' \vdash c : \textbf{cns}\ T}\,[j\text{-L}]}{\Gamma, \Gamma' \vdash \langle\, p \mid c \,\rangle}\,[\text{Cut}]$$

In our final step of removing redundant statement forms, we exploit the symmetry of statements standing next to each other on the same line in the above grammar. Producers of data (constructors) are perfectly symmetric to consumers of codata (destructors) and consumers of data (pattern matches) are perfectly symmetric to producers of codata (copattern matches).

We therefore can collapse either the chirality in the system, which means unifying producers and consumers, or we can collapse the polarity, which means unifying data and codata types. It seems more intuitive to collapse the polarity, as this still makes a producer and a consumer of the same type meet in a cut, a one-sided sequent calculus [21]. We use the following unified syntax for the four remaining statement forms.

$$
\begin{array}{lll}
\langle\, C(\Gamma) \mid \tilde{\mu}x.s \,\rangle & \& \langle\, \mu\alpha.s \mid D(\Gamma) \,\rangle & \to \textbf{let}\ v\ =\ m(\Gamma);\ s \\
\langle\, x \mid \textbf{case}\ \{\ C(\Gamma) \Rightarrow s,\ \ldots\ \}\,\rangle & \& \langle\, \textbf{cocase}\ \{\ D(\Gamma) \Rightarrow s,\ \ldots\ \} \mid \alpha \,\rangle & \to \textbf{switch}\ v\ \{\ m(\Gamma) \Rightarrow s,\ \ldots\ \} \\
\langle\, \mu\alpha.s \mid \textbf{case}\ \{\ C(\Gamma) \Rightarrow s,\ \ldots\ \}\,\rangle & \& \langle\, \textbf{cocase}\ \{\ D(\Gamma) \Rightarrow s,\ \ldots\ \} \mid \tilde{\mu}x.s \,\rangle & \to \textbf{new}\ v\ =\ \{\ m(\Gamma) \Rightarrow s,\ \ldots\ \};\ s \\
\langle\, C(\Gamma) \mid \alpha \,\rangle & \& \langle\, x \mid D(\Gamma) \,\rangle & \to \textbf{invoke}\ v\ m(\Gamma)
\end{array}
$$

Here, m is a unified notation for constructors C and destructors D, similar to how we use v for variables and covariables. On the type level, we uniformly describe both data and codata types by signatures, without polarities.

$$
\textbf{data}\ T\ \{\ C(\Gamma),\ \ldots\ \} \qquad \& \ \textbf{codata}\ T\ \{\ D(\Gamma),\ \ldots\ \} \qquad \to \textbf{signature}\ T\ \{\ m(\Gamma),\ \ldots\ \}
$$

With this unification, a producer of a signature now represents a constructor or destructor and a consumer now represents a pattern match or copattern match. We can choose to view a signature as a data type or a codata type. We will see examples of this in Section 4.

2.3 Practical Extensions

After shrinking down sequent calculus, we now extend it with language constructs that make it practically usable as a compiler intermediate representation, and call the result AxCut.

2.3.1 Explicit Substitutions

We have formulated all typing rules in accordance with linear type systems or, when reading the typing environments as ordered lists, even ordered type systems. Now we add explicit structural rules for weakening, contraction, and exchange. We combine all of them into one construct for explicit substitutions [1] of variables for variables.

$$
\text{Statements} \quad s \ ::=\ \ldots \ \mid\ \textbf{substitute}\ [v \mapsto v,\ \ldots];\ s
$$

Substitution statements define a new environment for the remaining statement s based on the variables in scope. This includes reordering variables by using them once, 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 [25].

While we have used an ordered system in Section 2.2, the transformation presented there can as well be applied to a source calculus with unrestricted variable use. For the resulting system, we can then infer explicit substitutions, following a principled approach which can be easily automated. To give an idea of how this works, consider the following program in a version of AxCut with unrestricted variable use. It receives an integer and a list and returns the same list with the integer prepended twice.

```
define consTwice(v : ext Int, l0 : prd List, k : cns List) =
  let l1 = cons(v, l0);
  invoke k cons(v, l1)
```

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. In this example, we get the below program in AxCut with explicit substitutions. The `cons` in the **invoke** statement does not need explicit arguments anymore, because they now exactly match the context.

$$
\begin{array}{l}
\textbf{define}\ \mathsf{consTwice}(v\ :\ \textbf{ext}\ \mathsf{Int},\ l_0\ :\ \textbf{prd}\ \mathsf{List},\ k\ :\ \textbf{cns}\ \mathsf{List})\ = \\
\quad \textbf{substitute}\ [v \mapsto v,\ k \mapsto k,\ v_0 \mapsto v,\ l_0 \mapsto l_0]; \\
\quad \textbf{let}\ l_1\ =\ \mathsf{cons}(v_0,\ l_0); \\
\quad \textbf{substitute}\ [v \mapsto v,\ l_1 \mapsto l_1,\ k \mapsto k]; \\
\quad \textbf{invoke}\ k\ \mathsf{cons}
\end{array}
$$

To avoid calculating the free variables of substatements repeatedly, we can annotate them in one pre-pass. However, in some cases we have to substitute renamed variables into substatements, so we have to update the annotations accordingly then.

The placement of explicit substitutions may impact performance. Following garbage-free reference counting [38], we duplicate variables as late as possible and drop them as early as possible. These 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.

2.3.2 Labels and Jumps

Following Binder et al. [6], we define programs to be a list of top-level label definitions, each consisting of one statement in its body. These labels can be jumped to from other statements. Each label l expects an environment Γ to be present when we jump to it. The jump site has to arrange for the environment to exactly fit.

$$
\text{Programs}\quad P\ ::=\ \textbf{define}\ l\ :\ \Gamma\ =\ s,\ \ldots \qquad \text{Statements}\quad s\ ::=\ \ldots\ \mid\ \textbf{jump}\ l
$$

Labels allow us to reuse statements multiple times, for example, when control flow joins back together [32]. Moreover, they can be mutually recursive and therefore allow us to express loops, also irreducible ones. Logically speaking, labels can be thought of as lemmas and jumps as appeals to them. Their termination or productivity must be checked separately.

2.3.3 Extern Statements and Types

Every program runs in an external environment. We therefore need a way to interact with the outside world, for example for input, output, and termination. We thus introduce a general **extern** statement, which has a name m, zero or more arguments v, and zero or more clauses $\{\ (\Gamma_1) \Rightarrow s_1,\ \ldots\ \}$, each with zero or more parameters described by Γ.

$$
\text{Statements}\quad s\ ::=\ \ldots\ \mid\ \textbf{extern}\ m(v,\ \ldots)\{\ (\Gamma) \Rightarrow s,\ \ldots\ \}
$$

They are a general extension mechanism for primitive operations. For example, the addition statement is expressed as $\textbf{extern}\ \mathsf{add}(v_1,\ v_2)\ \{\ (v) \Rightarrow s\ \}$, which adds the machine integers given by v_1 and v_2, and makes the result available as v in the remaining statement s. While add has one clause to continue execution, in general, externs can have multiple clauses, *e.g.*, a conditional statement, or even no clauses, *e.g.*, a terminating statement.

Extern statements generally operate on external types, such as the type Int of machine integers in the above example of add. These external types can appear in user-defined signatures, and are distinct from logical types. We thus extend the typing environment with a third kind of binding for variables of external types. A type τ is thus either a producer **prd** T or a consumer **cns** T of a signature or an external type **ext** T. Logically speaking, external types can be thought of as atomic propositions and extern statements as daimons [22, 50].

Syntax of AxCut

Programs	P	::=	**define** $l\ :\ \Gamma\ =\ s,\ ...$	Branches	b	::=	$\{\ m(\Gamma) \Rightarrow s,\ ...\ \}$
Statements	s	::=	**jump** l	Labels	l	::=	$f\ \mid g\ \mid ...$
		|	**substitute** $[v \mapsto v,\ ...];\ s$	Variables	v	::=	$x\ \mid y\ \mid j\ \mid k\ \mid ...$
		|	**extern** $m(v,\ ...)\{\ (\Gamma) \Rightarrow s,\ ...\ \}$	Term symbols	m	::=	nil | true | apply ...
		|	**let** $v\ =\ m(\Gamma);\ s$	Type symbols	T	::=	List | Bool | Func ...
		|	**new** $v\ =\ (\Gamma)b;\ s$				
		|	**switch** $v\ b$				
		|	**invoke** $v\ m$				

Signatures	Σ	::=	**signature** $T\ =\ \{\ m(\Gamma),\ ...\ \},\ ...$	Types	τ	::=	**prd** T
Type Env.	Γ	::=	$v\ :\ \tau,\ ...$			|	**cns** T
Label Env.	Θ	::=	$l\ :\ \Gamma,\ ...$			|	**ext** T

Figure 3 Syntax of terms and types in AxCut.

3 AxCut

In this section, we formally introduce AxCut. We briefly summarize its syntax and explain the typing rules. Moreover, we give an operational semantics in terms of an abstract machine.

3.1 Syntax

Figure 3 defines the syntax of AxCut. We explain a few syntactic elements not mentioned so far. In **new** statements, which bind (co)pattern matches, we explicitly annotate the closure environment Γ, because it is operationally relevant. In contrast, **invoke** does not mention arguments since we require the environment to match at the call site. We define a separate syntactic category of branches b that contain one clause $m(\Gamma) \Rightarrow s$ for each symbol in a signature and symmetrically occur in **switch** and **new** statements. As we have unified data and codata types, we collect all names for constructors and destructors in the syntactic category m of term-level symbols, which also contains the names of extern statements. Similarly, the syntactic category T of type-level symbols contains signature names as well as names of external types. All signatures are collected in a global map Σ which maps signature names to a list of symbols m, and each symbol to a list of parameters Γ. Finally, a label environment Θ maps labels to the type environments they assume.

3.2 Typing

Figure 4 defines the typing rules for AxCut. Programs are typed with signatures Σ and with labels Θ. The statement for each label must be well-typed in the annotated type environment Γ. The only typing rule that refers to the label environment is Jump. It ensures that the current type environment exactly matches the one assumed by the label. We omit the signatures Σ and the label environment Θ from the rules, since they do not change.

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. All other typing rules, except for rule Extern, follow the rules of ordered logic. Extern statements, such as add, do not consume the variables they use and it does not matter

Program Typing $\boxed{\Sigma \mid \Theta \vdash P}$

$$\frac{\Theta(l) = \Gamma \quad \Sigma \mid \Theta \mid \Gamma \vdash s \quad \ldots}{\Sigma \mid \Theta \vdash \textbf{define}\ l : \Gamma = s, \ldots}\ [\textsc{Program}]$$

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

$$\frac{\Theta(l) = \Gamma}{\Gamma \vdash \textbf{jump}\ l}\ [\textsc{Jump}] \qquad \frac{v_1' : \tau_1, \ldots \vdash s \quad \Gamma(v_1) = \tau_1 \quad \ldots}{\Gamma \vdash \textbf{substitute}\ [v_1' \mapsto v_1, \ldots];\ s}\ [\textsc{Substitute}]$$

$$\frac{v_1 : \tau_1, \ldots \vdash m : (\Gamma_1), \ldots \qquad \Gamma(v_1) = \tau_1 \quad \ldots \qquad \Gamma, \Gamma_1 \vdash s_1 \quad \ldots}{\Gamma \vdash \textbf{extern}\ m(v_1, \ldots)\{\ (\Gamma_1) \Rightarrow s_1, \ \ldots\ \}}\ [\textsc{Extern}]$$

$$\frac{\Sigma(T) = \{\ \ldots, m(\Gamma_0), \ \ldots\ \} \qquad \Gamma, v : \textbf{prd}\ T \vdash s}{\Gamma, \Gamma_0 \vdash \textbf{let}\ v = m(\Gamma_0);\ s}\ [\textsc{Let}]$$

$$\frac{\Sigma(T) = \{\ m_1(\Gamma_1), \ \ldots\} \qquad \Gamma, v : \textbf{cns}\ T \vdash s \qquad \Gamma_1, \Gamma_0 \vdash s_1 \quad \ldots}{\Gamma, \Gamma_0 \vdash \textbf{new}\ v = (\Gamma_0)\{\ m_1(\Gamma_1) \Rightarrow s_1, \ \ldots\ \};\ s}\ [\textsc{New}]$$

$$\frac{\Sigma(T) = \{\ m_1(\Gamma_1), \ \ldots\} \qquad \Gamma, \Gamma_1 \vdash s_1 \quad \ldots}{\Gamma, v : \textbf{prd}\ T \vdash \textbf{switch}\ v\ \{\ m_1(\Gamma_1) \Rightarrow s_1, \ \ldots\ \}}\ [\textsc{Switch}] \qquad \frac{\Sigma(T) = \{\ \ldots, m(\Gamma), \ \ldots\}}{\Gamma, v : \textbf{cns}\ T \vdash \textbf{invoke}\ v\ m}\ [\textsc{Invoke}]$$

Figure 4 Typing rules of AxCut.

at which positions in the environment these variables occur. Rule Extern merely checks that the variables v_i are of the correct type and adds the bound variables (Γ_i) to the type environment in each statement s_i, but otherwise does not change the environment. Which assumptions and consequences an external symbol m has is defined outside the system. They usually only involve external types.

The remaining rules are concerned with producers and consumers, as described in Section 2.2. In rules Let and New, an introduction rule meets an activation rule in a cut. The introduced term is bound to a variable v in the remaining statement s. In rule Let the only subderivation is for the remaining statement s. There are no subderivations for the arguments of symbol m, as all of them would be axioms. We omit them and instead simply require the type environment Γ_0 to exactly match the signature of m. This environment is consumed before the new binding is added. Similarly, in rule New, 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 symbol. Dually, in rules Switch and Invoke, an introduction meets an axiom in a cut. The variable v is consumed in both rules. In rule Switch, the rest of the environment Γ must be used in each clause in addition to the parameters Γ_i of the corresponding symbol. In rule Invoke, the rest of the environment Γ must exactly match the signature of the symbol m, again because the variable restriction is baked into the rules.

Example To get acquainted with the system, let us look again at the example from Section 1.1 of multiplication of a list of numbers with early return, this time in AxCut. Corresponding parts are again highlighted in the same color.

```
define mult : (k : cns Cont, l : prd List) = substitute [h ↦ k, k ↦ k, l ↦ l]; jump loop

define loop : (h : cns Cont, k : cns Cont, l : prd List) = switch l {
  nil() ⇒ extern lit 1 { (r) ⇒ substitute [r ↦ r, k ↦ k]; invoke k ret }
  cons(x, xs) ⇒ extern ifz(x) {
    () ⇒ extern lit 0 { (r) ⇒ substitute [r ↦ r, h ↦ h]; invoke h ret }
    () ⇒
      substitute [h ↦ h, xs ↦ xs, k ↦ k, x ↦ x];
      new j = (k, x){
        ret(z) ⇒ extern mul(x, z) { (r) ⇒ substitute [r ↦ r, k ↦ k]; invoke k ret }
      };
      substitute [h ↦ h, j ↦ j, xs ↦ xs]; jump loop } }
```

It is visible how explicit substitutions are used to duplicate the context k in mult, by using it in two right-hand sides, and to drop one of the contexts in the nil case and the else branch, by not mentioning them on any of the right-hand sides. Moreover, explicit substitutions rearrange the context appropriately before direct or indirect jumps but also for other non-external constructs, *e.g.*, for the closure environment consumed by the context j created by **new**.

3.3 Semantics

We define the semantics of AxCut in terms of an abstract machine, whose syntax is defined in Figure 5. 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 P. The latter is required for jumping to labels and does not change during execution. We hence omit it from the definition of the individual steps. Values are either producers $\{\ m;\ E\ \}$ consisting of a symbol and a field environment, consumers $\{\ E;\ b\ \}$ consisting of a closure environment and branches, or machine integers.

Figure 5 defines the evaluation steps of the abstract machine. There is one rule for each kind of statement, which mirrors its typing rule. Rule *(let)* constructs a producer by removing the part E_0, corresponding to the variable environment Γ_0 of the symbol m, from the value environment, and then adding a binding of producer $\{m;\ E_0\}$ to E. Dually, rule *(new)* constructs a consumer. 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)* destructs a producer by looking up the symbol m in the value environment, and selecting the corresponding statement $b(m)$ to execute. It then removes the binding for the producer and adds the value environment E_0. Rule *(invoke)* destructs a consumer by looking up the statement $b(m)$ corresponding to m and extending the value environment with the stored closure environment E_0. Rule *(jump)* transfers execution to the statement $P(l)$ of label l defined in program P. Rule *(subst)* creates a new value environment by looking up each variable in the bindings of the old value environment. We list rules *(lit)*, *(add)*, and *(ifz)* as examples of the semantics of extern statements. Generally, the representation of external types and the semantics of extern statements are dependent on the platform the program runs on.

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 symbol is unknown but the clauses are known, and rule *(invoke)* where the symbol is known but the clauses are unknown. Finally, extern statements have control over if and how to continue

Syntax of Machine States

$$
\begin{array}{llll}
\text{Values} & V ::= \{\ m;\ E\ \} & \text{Environments} & E ::= x \mapsto V,\ \ldots \\
& \mid\ \{\ E;\ b\ \} & \text{Configurations} & M ::= \langle\ s \parallel E \parallel P\ \rangle \\
& \mid\ 0 \mid 1 \mid \ldots &&
\end{array}
$$

Stepping Relation

$$
\begin{array}{lllll}
\textit{(let)} & \langle\ \textbf{let}\ v\ =\ m(\Gamma_0);\ s \parallel E,\ E_0\ \rangle & \to & \langle\ s \parallel E,\ v \mapsto \{\ m;\ E_0\ \}\ \rangle & \text{where } E_0 : \Gamma_0 \\
\textit{(new)} & \langle\ \textbf{new}\ v\ =\ (\Gamma_0)b;\ s \parallel E,\ E_0\ \rangle & \to & \langle\ s \parallel E,\ v \mapsto \{\ E_0;\ b\ \}\ \rangle & \text{where } E_0 : \Gamma_0 \\
\textit{(switch)} & \langle\ \textbf{switch}\ v\ b \parallel E,\ v \mapsto \{\ m;\ E_0\ \}\ \rangle & \to & \langle\ b(m) \parallel E,\ E_0\ \rangle & \\
\textit{(invoke)} & \langle\ \textbf{invoke}\ v\ m \parallel E,\ v \mapsto \{\ E_0;\ b\ \}\ \rangle & \to & \langle\ b(m) \parallel E,\ E_0\ \rangle & \\
\textit{(jump)} & \langle\ \textbf{jump}\ l \parallel E\ \rangle & \to & \langle\ P(l) \parallel E\ \rangle & \\
\textit{(subst)} & \langle\ \textbf{substitute}\ [v_1' \mapsto v_1,\ \ldots];\ s \parallel E\ \rangle & \to & \langle\ s \parallel v_1' \mapsto E(v_1),\ \ldots\ \rangle & \\
\textit{(lit)} & \langle\ \textbf{extern}\ \mathsf{lit}\ n\ \{\ (v) \Rightarrow s\ \}\ \parallel E\ \rangle & \to & \langle\ s \parallel E,\ v \mapsto n\ \rangle & \\
\textit{(add)} & \langle\ \textbf{extern}\ \mathsf{add}(v_1,\ v_2)\ \{\ (v) \Rightarrow s\ \}\ \parallel E\ \rangle & \to & \langle\ s \parallel E,\ v \mapsto E(v_1)\ +\ E(v_2)\ \rangle & \\
\textit{(ifz)} & \langle\ \textbf{extern}\ \mathsf{ifz}(v)\ \{\ ()\Rightarrow s_1,\ ()\Rightarrow s_2\ \}\ \parallel E\ \rangle & \to & \langle\ s_1 \parallel E\ \rangle & \text{if } E(v) = 0 \\
& & & \langle\ s_2 \parallel E\ \rangle & \text{otherwise}
\end{array}
$$

Figure 5 Small-step operational machine semantics of AxCut.

execution.

Type safety The typing rules for machine states are described in Appendix A. AxCut satisfies the usual theorems of progress (Theorem 1) and preservation (Theorem 2), under the assumption that the semantics of the extern statements do so. In particular, this is true with the rules for lit, add and ifz above.

▶ **Theorem 1** (Progress)**.**
If $\Sigma \mid \Theta \vdash M$ *then there exists a unique machine state* M' *such that* $M \to M'$*.*

▶ **Theorem 2** (Preservation)**.**
If $\Sigma \mid \Theta \vdash M$ *and* $M \to M'$ *then* $\Sigma \mid \Theta \vdash M'$*.*

These theorems have been shown in Idris 2 by the intrinsic typing and the totality of the `step`-function on machines when restricted to the above extern statements. Programs, as defined here, never terminate unless we include an extern exit statement and a final state for the abstract machine. This is in analogy to how the only way for cut elimination of a closed derivation in sequent calculus to terminate is a daimon. As we will see, it also directly corresponds to how a program executed on a real machine typically ends with some communication with an external system, for example, an exit syscall to the operating system.

4 From AxCut to Machine Code

On the one hand, AxCut is a normal form of classical sequent calculus presented in Section 2, 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 code generation to a subset of RISC-V [45]. There are other implementations targeting x86-64 and AArch64, which we briefly discuss in Section 5.1.

Syntax of RISC-V

Instructions I	::=	l :	label
	|	`add` $r\ r\ r$	add registers
	|	`addi` $r\ r\ i$	add immediate
	|	`mv` $r\ r$	move register
	|	`li` $r\ i$	load immediate
	|	`la` $r\ l$	load address
	|	`j` o	jump
	|	`jr` $r\ o$	jump register
	|	`beq` $r\ r\ o$	branch if equal
	|	`lw` $r\ r\ i$	load word
	|	`sw` $r\ r\ i$	store word

Registers	r	::=	`x0`, ..., `x31` registers
Offsets	o	::=	l | i
Labels	l	::=	`l0` | `l1` | ... label
Immediates	i	::=	`0` | `1` | ... integer

Figure 6 Syntax of RISC-V.

4.1 RISC-V

Figure 6 defines the syntax of the subset of RISC-V we use as the target. Programs 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 immediates can often only be 12-bit integers. Moreover, we freely use pseudo instructions which have to be desugared into actual instructions. In RISC-V there are typically 32 registers, named `x0`,...,`x31`, each holding a machine integer. Register `x0` is special in that it always contains the value 0.

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`. The `mv` instruction copies the value of the second register into the first register. The load instruction `li` loads an immediate into its register operand and the load instruction `la` loads an address specified by a label into its register operand. Direct jump instructions `j` jump to an address that 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.

4.2 Overview

Typing judgments in AxCut are of the form $\Sigma \mid \Theta \mid \Gamma \vdash\ s$, 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. Each variable in Γ corresponds to two adjacent registers, corresponding to the two components of values in the operational semantics in Figure 5. They are assigned from left to right [4], starting with the first register not reserved for other purposes, which is `x4`.

The translation of extern statements depends on the target platform and consists of corresponding hardware instructions or system calls. Top-level labels and jumps in AxCut

are translated to assembly labels and direct jumps. A top-level label is typed with an environment Γ which communicates what variables the label 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 [40] of registers. The second purpose of explicit substitutions is to drop and duplicate variables. Since we use reference counting for memory management, the code for explicit substitutions also decrements and increments reference counts of memory blocks associated with the dropped and duplicated variables.

The code for constructing and destructing producers and consumers is dual, in congruence with the abstract machine steps. Both **let** and **new** store values onto the heap. But for **let**, the block of memory is paired with a tag and contains the corresponding arguments, and for **new**, it is paired with a jump table and contains the closure environment. In both cases, the newly bound variable is thus represented by two registers. **let** and **new** are the only places where memory blocks are acquired. Similarly, both **switch** and **invoke** perform indirect jumps with a tag being used as 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 **new**. In both cases, at the beginning of each branch, the values stored in the memory block are loaded back into registers and the block of memory is released if its reference count is 0. The exact way values are stored to memory blocks and loaded back into registers is determined by the type signatures.

In the next subsections, we illustrate code generation on example programs. The full formal definition is given in Appendix B.

4.3 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 the **jump**, **substitute**, and **extern** statements.

4.3.1 Definitions are Labels, Jumps are Direct Jumps

In AxCut, a program is a list of label definitions. The type of a label is the type environment Γ it assumes. To jump to a label, we use a **jump** statement. The type environment Γ must exactly match what is assumed by the definition, which we track in the label environment Θ. In the following example, we have two label definitions, main and loop, and from main, we jump to loop.

AxCut

define main : Γ_0 =
  ...
  jump loop
define loop : Γ =
  ...

$$\frac{\Theta(\mathsf{loop}) = \Gamma}{\Sigma \mid \Theta \mid \Gamma \vdash \textbf{jump}\ \mathsf{loop}}\ [\textsc{Jump}]$$

Machine Code

```
main:
  ...
  j loop
loop:
  ...
```

We translate each label definition to the labeled translation of the statement and we translate jumps to direct jumps to the label. A crucial property, which stands in contrast to most other intermediate representations, is that in AxCut there is no a priori fixed concept of functions. Instead, signatures specify a protocol of interaction between a producer and a consumer, which includes synchronous communication, *i.e.*, function calls, as a special case. In Section 1 we have seen throwing an exception to and coroutining with the caller as other examples.

4.3.2 Explicit Substitutions are Parallel Moves

Before a jump, 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.

AxCut

$$\frac{y : \textbf{ext}\ \mathsf{Int},\ x : \textbf{ext}\ \mathsf{Int} \vdash \ldots}{x : \textbf{ext}\ \mathsf{Int},\ y : \textbf{ext}\ \mathsf{Int} \vdash \textbf{substitute}\ [y \mapsto y,\ x \mapsto x]; \ldots}\ [\textsc{Substitute}]$$

Machine Code

```
mv temp x5
mv x5 x7
mv x7 temp
...
```

We translate substitutions that change the order of variables to parallel moves [40], using the temporary register `temp`. 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 4.5.

4.3.3 Externs are System Calls and Hardware Instructions

Every program runs in an external environment and eventually needs to terminate, output a result, or interact with the outside world in another way. For this purpose, AxCut includes extern statements like exit, add, and ifz, which generally operate on external types, like machine integers. In this example, we halt execution and exit with code 0.

AxCut

$$\frac{}{\Gamma \vdash \textbf{extern}\ \mathsf{exit}()\{\}}\ [\textsc{Extern}]$$

Machine Code

```
li x17 93
li x10 0
ecall
```

We translate these extern statements either to system calls, as in this example, or to hardware instructions, like addition and branching. Here, we load the syscall number to `x17`, its argument to `x10`, and call the environment with `ecall`.

4.3.4 Summary

We have seen that it is possible to write programs in AxCut which just act on external types, without considering logical types. This makes it 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.

4.4 Compiling Producers and Consumers

In AxCut, we specify types by their signatures, which can be arbitrarily mutually recursive. In the following example, we specify the signature of a Maybe type with two constructors, which contain either some integer or none. It is intuitive to view it as a data type. Moreover, we specify the signature Cont of continuations with a destructor that returns an integer to them. It is intuitive to view this as a codata type. We emphasize, however, that formally there is no difference between these viewpoints and their opposite.

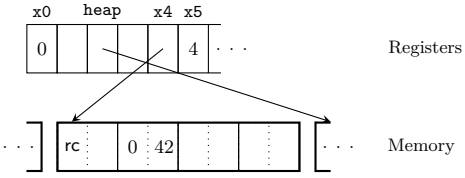
AxCut

```
signature Maybe = {
  none(), some(ext Int)
}
signature Cont = {
  ret(ext Int)
}
```

Machine Code

—

Illustration



In machine code, signatures 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 $\Gamma = v : \textbf{prd}\ \mathsf{Maybe}, \ldots$ is bound to $\mathsf{some}(42)$. 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, which we will come back to in Section 4.5. 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. Again, we only do so to help intuition, and everything we say is valid from the dual viewpoint as well.

4.4.1 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 $v_0 : \textbf{ext}\ \mathsf{Int}$ becomes the field environment of the constructor $\mathsf{some}(v_0)$. 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 **prd** Maybe. When we match on v, there are two subderivations, one for each of the two clauses, in accordance with the signature Maybe. Variable v is removed from the type environment, and, depending on the clause, the constructor fields are added. The above picture corresponds to the situation after the **let** but before the **switch**.

AxCut

$$\dfrac{\dfrac{\vdash \ldots \qquad x : \textbf{ext}\ \mathsf{Int} \vdash \ldots}{v : \textbf{prd}\ \mathsf{Maybe} \vdash \textbf{switch}\ v\ \{\ \mathsf{none}() \Rightarrow \ldots,\ \mathsf{some}(x) \Rightarrow \ldots\ \}}\,[\textsc{Switch}]}{v_0 : \textbf{ext}\ \mathsf{Int} \vdash \textbf{let}\ v = \mathsf{some}(v_0);\ \textbf{switch}\ v\ \{\ \mathsf{none}() \Rightarrow \ldots,\ \mathsf{some}(x) \Rightarrow \ldots\ \}}\,[\textsc{Let}]$$

Machine Code

```
sw x5 heap 12     jr x5 table     none:     some:
ACQUIRE x4        table:            ...       RELEASE x4
li x5 4             j none                    lw x5 x4 12
                    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 4.5. 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 clause, we first load the field environment, if there is any, and then execute the rest of the statement. In the clause 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 4.5. The fields are moved into registers which overwrites the matched variable.

4.4.2 Objects are Virtual Tables, Closures are Memory Blocks, Destructor Invocations are Indirect Jumps

We now consider the signature Cont as a codata type of continuations. Intuitively, a codata type is defined by what it can do rather than what it is. The only thing we can do with a continuation is invoke it with a value in order to continue. Hence, Cont has a single destructor ret which accepts an integer argument. In the following example, we create a continuation k with **new**, which defines the closure environment $(v : \textbf{ext}\ \mathsf{Int})$ and a statement that implements the return destructor. The statement must use the variable v, as well as the parameter x. Variable v is moved into the closure and not available in the rest of the statement. The type of k is **cns** Cont, it is a consumer with signature Cont. We can use k in the remaining statement with **invoke** on a destructor. In the example, we return to the continuation k with an argument r, that was added to the environment after **new** but before **invoke**. The environment must consist of exactly r and k, in this order.

AxCut

$$\dfrac{\dfrac{\dfrac{}{r : \textbf{ext}\ \mathsf{Int},\ k : \textbf{cns}\ \mathsf{Cont} \vdash \textbf{invoke}\ k\ \mathsf{ret}(r)}\,[\textsc{Invoke}]}{k : \textbf{cns}\ \mathsf{Cont} \vdash \ldots\ \textbf{invoke}\ k\ \mathsf{ret}(r)}\,[\ldots] \qquad x : \textbf{ext}\ \mathsf{Int},\ v : \textbf{ext}\ \mathsf{Int} \vdash \ldots}{v : \textbf{ext}\ \mathsf{Int} \vdash \textbf{new}\ k = (v : \textbf{ext}\ \mathsf{Int})\{\ \mathsf{ret}(x) \Rightarrow \ldots\ \};\ \ldots\ \textbf{invoke}\ k\ \mathsf{ret}(r)}\,[\textsc{New}]$$

Machine Code

```
sw x5 heap 12     table:      ret:
ACQUIRE x4          j ret       RELEASE x6
la x5 table                     lw x7 x6 12
...                             ...
jr x7 0
```

We translate the **new** 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 in analogy 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 clause 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 `x6`, occupied by the second variable in the

environment. In general, it must come after all destructor arguments, which are determined by the signature of Cont.

We translate the **invoke** statement to an indirect jump to the virtual table, which must be in register `x7`. This means that the instructions between **new** and **invoke** must have rearranged the registers, putting the argument for the destructor ret into registers `x4` and `x5`. The offset into the virtual table is given by the tag of destructor ret, which in this case is `0`. If there was more than one destructor, it could be different. This implements dynamic dispatch.

4.4.3 Exploiting the Non-Difference Between Viewpoints

Finally, we demonstrate some additional flexibility we gain from collapsing data and codata types. Consider the signature of a Result that is either ok or an error, viewed as a data type. 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

```
signature Result = {
  ok(ext Int), error(ext Int)
}
```

Machine Code

—

When compiling a chain of matches naively, each function in such a chain will upon return store the result in memory, which the caller then immediately investigates. In AxCut, however, we can choose a different viewpoint and represent pattern-matching contexts directly. While the type **prd** Result describes expressions, the type **cns** Result describes their contexts. We create such contexts with **new**, as in the following example.

AxCut

$$\frac{k : \textbf{cns}\ \mathsf{Result} \vdash \textbf{jump}\ f \qquad x : \textbf{ext}\ \mathsf{Int} \vdash \ldots \qquad y : \textbf{ext}\ \mathsf{Int} \vdash \ldots}{\textbf{new}\ k = (\,)\{\ \mathsf{ok}(x) \Rightarrow \ldots,\ \mathsf{error}(y) \Rightarrow \ldots\ \};\ \textbf{jump}\ f}\ [\textsc{New}]$$

Machine Code

```
li x4 null      table:      ok:     error:
la x5 table       j ok        ...       ...
j f               j error
```

We avoid storing intermediate results in memory, as we directly pass this context k to the label f. Intuitively, the context is a stack frame with two return addresses, known as vectored returns [36, 31]. The statement in f then invokes k with the corresponding destructor to select the control path, for example, with **invoke** k error(e). The values are returned in registers. This is another example of a definable protocol of interaction in AxCut.

4.4.4 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. So far, we have assumed that all variables, except in extern statements, are used linearly. Next, we describe how we translate the non-linear use of variables.

4.5 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.

4.5.1 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 **new**, corresponding to cuts with an activation rule, 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, not shown here. 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 heap 0
```

```
RELEASE x4 = sw heap x4 0
             mv heap x4
```

4.5.2 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 4.4, 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 [47, 48]. In combination with the allocation of equal-sized blocks that we use, we have an implementation of constant-time reference counting [27]. 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 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, pointed to by register `heap`, and one where the children of blocks must still be traversed and erased, which we call the todo list, pointed to by register `todo`. We acquire blocks from the free list as explained above, only falling back to the todo list if the free list is empty. Releasing a block whose reference count is `0` puts it onto the free list. There is no need to process the

children since they are moved into registers. Blocks are only put onto the todo list if the last reference to them is erased by weakening. The above definitions of `ACQUIRE` and `RELEASE` have to be adapted accordingly.

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 k into variables k_1 and k_2. It is precisely in these cases that we have to increment the reference count of the block of memory of k. The substitutions tell us exactly when, where, and how much these increments happen. The block of memory of k is in register `x4`. We use the temporary register `temp` to do the increment.

AxCut

$$\frac{k_1 \ : \ \textbf{cns}\ \mathsf{Cont},\ k_2 \ : \ \textbf{cns}\ \mathsf{Cont} \vdash \ \ldots}{k \ : \ \textbf{cns}\ \mathsf{Cont} \vdash \ \textbf{substitute}\ [k_1 \mapsto k,\ k_2 \mapsto k];\ \ldots}\ [\textsc{Substitute}]$$

Machine Code

```
lw temp x4 0
addi temp temp 1
sw temp x4 0
...
```

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

$$\frac{x \ : \ \textbf{prd}\ \mathsf{Result} \vdash \ \ldots}{x \ : \ \textbf{prd}\ \mathsf{Result},\ y \ : \ \textbf{prd}\ \mathsf{Maybe} \vdash \ \textbf{substitute}\ [x \mapsto x];\ \ldots}\ [\textsc{Substitute}]$$

Machine Code

```
lw temp x6 0
beq temp x0 then
  addi temp temp -1
  sw temp x6 0
  j done
then:
  sw todo x6 0
  mv todo x6
done:
...
```

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 todo list in register `todo`. Otherwise, we decrement the reference count.

4.5.3 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. All operations take constant time, independent of the number of live or dead blocks.

4.6 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, 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.

5 Evaluation

The design of AxCut closely follows the underlying logic, yielding an intermediate representation that actively utilizes the symmetries of classical sequent calculus. At the same time, we designed AxCut to admit a direct translation to machine code. Here, we briefly describe our implementation and report early benchmarking results.

5.1 Implementation

AxCut is implemented in Idris 2 [7], a dependently-typed functional programming language. We parse a textual representation of programs in AxCut and elaborate them to an intrinsically-typed representation [34]. The operational semantics defined in Section 3.3 is implemented as a stepping function on this representation. The translation itself encompasses a few hundred lines of code. Our implementation in Idris 2 not only compiles AxCut to RISC-V, as discussed in Section 4, but also supports compilation to x86-64 and AArch64. Both of these work in a very similar way. To keep the implementation simple and to not obscure the underlying logic, our implementation currently neither performs register allocation nor does it perform any optimizations. Moreover, register and memory layout could be improved significantly.

RISC-V

The implementation of the translation to RISC-V closely follows the presentation in Section 4. There are, however, a few differences. We desugar pseudo instructions (such as `j`, `mv`, `li`, etc.) into actual instructions. Moreover, we do not use labels but instead resolve all labels as addresses within the code. Of course, we could instead generate code for an existing assembler for RISC-V, which allows for the use of labels and pseudo instructions.

x86-64

The code we generate for x86-64 is again very similar to the translation in Section 4. However, since we want to create an executable to run on a Linux system, we emit code for the Netwide Assembler (NASM)[1]. This allows us to use labels just as presented in the paper. We then use the object code generated for this assembler routine by NASM 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. In general, the instructions for x86-64 are different from those for RISC-V of course. However, we only use a small subset of basic instructions (like, `mov`, `add`, `jmp`, `cmp`, etc.), so that the differences for the translation are minor. Still, one practically important difference is that in x86-64 there are only 16 registers compared to 32 registers in RISC-V. To allow 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.

[1] https://nasm.us

Table 1 Benchmark results comparing the different systems for the last N reported in Figure 11.

	Mean time in ms (standard deviation in ms)						
Benchmark (N)	**AxCut**	**MLton**	**OCaml**	**Koka**	**Koka (Opt)**	**Rust**	**Rust (Opt)**
Factorial Acc. (10M)	52 (±2)	53 (±3)	47 (±3)	52 (±2)	54 (±3)	55 (±2)	39 (±5)
Fibonacci (40)	812 (±6)	477 (±3)	434 (±4)	414 (±7)	169 (±3)	576 (±5)	211 (±2)
Sum Range (10M)	401 (±4)	427 (±6)	4057 (±86)	1135 (±17)	344 (±10)	1265 (±33)	733 (±9)
Iterate Incr. (100M)	200 (±3)	40 (±4)	133 (±2)	406 (±13)	1 (±0)	240 (±17)	1 (±0)
Match Options (10M)	258 (±6)	95 (±3)	2024 (±132)	299 (±17)	268 (±3)	260 (±7)	1 (±0)
Lookup Tree (10M)	324 (±2)	291 (±8)	3551 (±174)	1054 (±16)	292 (±3)	1388 (±50)	700 (±23)
Erase Unused (10k)	120 (±22)	16 (±4)	81 (±13)	4165 (±44)	198 (±4)	2417 (±94)	1104 (±17)

AArch64

Thanks to a community effort, we can also 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[2]. Using the same C driver as for x86-64, the resulting executables can be run on a Linux system and have also been tested on an Apple M1.

5.2 Benchmarks

To get a first impression of the performance of the generated code, we conducted microbenchmarks and compare the running time with a selection of industrial and research languages. Some of the compilers we compare against produce highly optimized code, while our implementation is rather naive in comparison and does not perform any compile-time optimizations, like inlining or constant propagation. The goal of these benchmarks is thus not to show raw performance, but to demonstrate that the compilation of sequent calculus to machine code actually works and is comparable to existing compilers when it comes to performance.

For the AxCut benchmarks we used the surface language **Fun** presented in [6] and manually applied a translation to sequent calculus which avoids administrative redexes. We then manually applied the transformation to AxCut presented in Section 2. Finally, we ran our implementation of code generation from AxCut to machine code presented in Section 4. We did not apply any optimizations.

The detailed results and a discussion are presented in Appendix C. Table 1 lists the results for the largest input N in tabular form. In general, we can see that even though the implementation of code generation for AxCut itself is fairly simple, running times are comparable with state-of-the-art compilers. Executables generated by AxCut hence provide a decent value-to-weight ratio. However, the results also show that compile-time optimizations make a significant difference. Of course, standard optimizations could be performed, both on AxCut as well as on the machine code it generates. We leave exploring the optimization of AxCut programs to future work.

6 Related Work

We discuss term assignments for classical sequent calculus, which AxCut is explicitly based on, and compare AxCut with other kinds of intermediate representations.

[2] https://www.gnu.org/software/binutils/

6.1 Term Assignments for Sequent Calculus

Curien and Herbelin [10] present the $\overline{\lambda}\mu\tilde{\mu}$-calculus as a term assignment for Gentzen'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 consider the logical connective of implication only and globally fix the evaluation order to be either call-by-value or call-by-name.

Wadler [44] 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.

Zeilberger [50] 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 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.

Downen [12] 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 [13], a tutorial explaining a computational interpretation of classical logic via focusing [14], and how to mix call-by-value and call-by-name for arbitrary data and codata types within the same program [15].

Ostermann et al. [33] 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 focusing.

6.2 Intermediate Representations

Compiling programming languages via continuation-passing style is a long and ongoing tradition [43, 4, 26]. In continuation-passing style, functions explicitly call their continuation instead of returning to their implicitly available calling context. Our approach is more general, in that we allow for multiple arbitrarily-structured explicit contexts. This subsumes, for example, double-barreled continuation-passing style for implementing exceptions. We can mix different such calling and returning conventions within the same program. Finally, our contexts are fully first-class, following classical logic.

An alternative to continuation-passing style is A-normal form [41, 19]. 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 variable. Our normal form fixes the evaluation order, a property it shares with continuation-passing style and A-normal form.

Leißa et al. [29] present an intermediate representation where, unlike in lambda-lifted representations, there are no scopes and top-level definitions do not receive parameters. Rather, the nesting of scopes follows from the use of variables. We took this as inspiration to make our top-level labels not take parameters. Instead, they are annotated with the free

variables they assume.

Maurer et al. [32] and Cong et al. [9] each present an intermediate representation with some form of control operator that reifies the current context and gives a name to it. In both of these works, the use of this name is restricted and it is not first-class. In contrast, we can give names to arbitrary contexts and use them in unrestricted ways. Moreover, our notion of context is more general and goes beyond returning to it with an argument.

Downen et al. [17] present an intermediate representation that is explicitly based on sequent calculus. However, being in the context of the GHC Haskell compiler forced some design decisions in order to maintain a direct correspondence to the existing intermediate representation of Haskell. In particular, consumers are restricted in order to keep the system pure. The authors write:

> However, it still remains to be seen how an unrestrained, and thus more cohesive and simpler, classic sequent calculus would fare as the intermediate language of a compiler.

Our work can be seen as the first part of answering this question.

Binder et al. [6] aim to introduce sequent calculus to "compiler hackers and programming-language enthusiasts". They do so by translating a direct-style functional programming language with data and codata types to a sequent-calculus-based intermediate representation. We show how such an intermediate representation can be further compiled to machine code.

7 Conclusion

In this paper, we have presented a compiler intermediate representation AxCut, which serves as a bridge between fully symmetric classical sequent calculus and bare-metal machine code. We have identified AxCut as a fragment of a term assignment for sequent calculus proofs together with a normalization into this fragment, and we have shown how to generate code for RISC-V, x86-64 and AArch64 from AxCut. The symmetric nature of classical sequent calculus naturally facilitates expressing control effects 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 compilation of high-level features for structuring interactive programs, such as async/await, coroutines, or effect handlers, to AxCut. Furthermore, we believe the uniform and principled representation of programs, expressions, and contexts in AxCut 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.

8 Data-Availability Statement

The benchmarks presented in Section 5.2 and the implementations in Idris 2 of the normalization procedure described in Section 2.2 and of AxCut described in Section 5.1 are available to be run in a Docker container in the accompanying artifact [42].

Acknowledgments

The work on this project was supported by the Deutsche Forschungsgemeinschaft (DFG – German Research Foundation) – project number DFG-448316946.

References

1 M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991. doi: 10.1017/S0956796800000186.

2 A. Abel, B. Pientka, D. Thibodeau, and A. Setzer. Copatterns: Programming infinite structures by observations. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 27–38, New York, NY, USA, 2013. ACM. doi: 10.1145/2429069.2429075.

3 J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992. doi: 10.1093/logcom/2.3.297.

4 A. W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, NY, USA, 1992. ISBN 0-521-41695-7. doi: 10.1017/CBO9780511609619.

5 G. Bierman, C. Russo, G. Mainland, E. Meijer, and M. Torgersen. Pause'n'play: Formalizing asynchronous C#. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 233–257, Berlin, Heidelberg, 2012. Springer. doi: 10.1007/978-3-642-31057-7_12.

6 D. Binder, M. Tzschentke, M. Müller, and K. Ostermann. Grokking the sequent calculus (functional pearl). *Proc. ACM Program. Lang.*, 8(ICFP), aug 2024. doi: 10.1145/3674639. URL `https://doi.org/10.1145/3674639`.

7 E. Brady. Idris 2: Quantitative Type Theory in Practice. In A. Møller and M. Sridharan, editors, *35th European Conference on Object-Oriented Programming (ECOOP 2021)*, volume 194 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:26, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-190-0. doi: 10.4230/LIPIcs.ECOOP.2021.9. URL `https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2021.9`.

8 R. M. Burstall, D. B. MacQueen, and D. T. Sannella. Hope: An experimental applicative language. In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, LFP '80, page 136–143, New York, NY, USA, 1980. Association for Computing Machinery. ISBN 9781450373968. doi: 10.1145/800087.802799.

9 Y. Cong, L. Osvald, G. M. Essertel, and T. Rompf. Compiling with continuations, or without? whatever. *Proc. ACM Program. Lang.*, 3(ICFP):79:1–79:28, July 2019. ISSN 2475-1421. doi: 10.1145/3341643.

10 P.-L. Curien and H. Herbelin. The duality of computation. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, page 233–243, New York, NY, USA, 2000. Association for Computing Machinery. ISBN 1581132026. doi: 10.1145/351240.351262.

11 P.-L. Curien and G. Munch-Maccagnoni. The duality of computation under focus. In *IFIP international conference on theoretical computer science*, pages 165–181. Springer, 2010. doi: 10.1007/978-3-642-15240-5_13.

12 P. Downen. *Sequent Calculus: A Logic and a Language for Computation and Duality*. PhD thesis, University of Oregon, 2017.

13 P. Downen and Z. M. Ariola. The duality of construction. In *Programming Languages and Systems: 23rd European Symposium on Programming, ESOP 2014*, pages 249–269. Springer, 2014. doi: 10.1007/978-3-642-54833-8_14.

14 P. Downen and Z. M. Ariola. A tutorial on computational classical logic and the sequent calculus. *Journal of Functional Programming*, 28:e3, 2018. doi: 10.1017/S0956796818000023.

15 P. Downen and Z. M. Ariola. Compiling with classical connectives. *Logical Methods in Computer Science*, 16, 2020. doi: 10.23638/LMCS-16(3:13)2020.

16 P. Downen and Z. M. Ariola. Duality in Action. In N. Kobayashi, editor, *6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021)*, volume 195 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:32, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi: 10.4230/LIPIcs.FSCD.2021.1.

17 P. Downen, L. Maurer, Z. M. Ariola, and S. Peyton Jones. Sequent calculus as a compiler intermediate language. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, page 74–88, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342193. doi: 10.1145/2951913.2951931. URL `https://doi.org/10.1145/2951913.2951931`.

18 P. Downen, Z. Sullivan, Z. M. Ariola, and S. Peyton Jones. Codata in action. In L. Caires, editor, *Programming Languages and Systems*, pages 119–146, Cham, 2019. Springer International Publishing. doi: 10.1007/978-3-030-17184-1_5.

19 C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, page 237–247, New York, NY, USA, 1993. Association for Computing Machinery. ISBN 0897915984. doi: 10.1145/155090.155113.

20 G. Gentzen. Untersuchungen über das logische schließen. i. *Mathematische Zeitschrift*, 39, 1935. doi: 10.1007/BF01201353.

21 J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–101, 1987. doi: 10.1016/0304-3975(87)90045-4.

22 J.-Y. Girard. Locus solum: From the rules of logic to the logic of rules. *Mathematical Structures in Computer Science*, 11(3):301–506, 2001. doi: 10.1017/S096012950100336X.

23 T. G. Griffin. A formulae-as-type notion of control. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, page 47–58, New York, NY, USA, 1989. Association for Computing Machinery. ISBN 0897913434. doi: 10.1145/96709.96714. URL `https://doi.org/10.1145/96709.96714`.

24 T. Hagino. Codatatypes in ml. *Journal of Symbolic Computation*, 8(6):629–650, 1989. doi: 10.1016/S0747-7171(89)80065-3.

25 M. Hofmann. *Syntax and Semantics of Dependent Types*, page 79–130. Publications of the Newton Institute. Cambridge University Press, 1997. doi: 10.1017/CBO9780511526619.004.

26 A. Kennedy. Compiling with continuations, continued. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, page 177–190, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595938152. doi: 10.1145/1291151.1291179.

27 C. K. Lam and L. Parreaux. 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*, page 188–216, Berlin, Heidelberg, 2024. Springer-Verlag. ISBN 978-981-97-2299-0. doi: 10.1007/978-981-97-2300-3_11. URL `https://doi.org/10.1007/978-981-97-2300-3_11`.

28 D. Leijen. Koka: Programming with row polymorphic effect types. In *Proceedings of the Workshop on Mathematically Structured Functional Programming*, 2014. doi: 10.4204/eptcs.153.8.

29 R. Leißa, M. Köster, and S. Hack. A graph-based higher-order intermediate representation. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code*

Generation and Optimization, CGO '15, page 202–212, USA, 2015. IEEE Computer Society. ISBN 9781479981618. doi: 10.5555/2738600.2738626.

30 A. Lorenzen, D. Leijen, and W. Swierstra. Fp²: Fully in-place functional programming. *Proc. ACM Program. Lang.*, 7(ICFP), aug 2023. doi: 10.1145/3607840.

31 S. Marlow, A. R. Yakushev, and S. Peyton Jones. Faster laziness using dynamic pointer tagging. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, page 277–288, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595938152. doi: 10.1145/1291151.1291194. URL `https://doi.org/10.1145/1291151.1291194`.

32 L. Maurer, P. Downen, Z. M. Ariola, and S. L. Peyton Jones. Compiling without continuations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 482–494, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4988-8. doi: 10.1145/3062341.3062380.

33 K. Ostermann, D. Binder, I. Skupin, T. Süberkrüb, and P. Downen. Introduction and elimination, left and right. *Proceedings of the ACM on Programming Languages*, 6(ICFP): 438–465, 2022. doi: 10.1145/3547637.

34 A. Pardo, E. Gunther, M. Pagano, and M. Viera. An internalist approach to correct-by-construction compilers. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*, PPDP '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450364416. doi: 10.1145/3236950.3236965.

35 D. Peter. hyperfine. A command-line benchmarking tool, 2024. URL `https://github.com/sharkdp/hyperfine`. [Last access: 28-02-2024].

36 S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless g-machine. *Journal of functional programming*, 2(2):127–202, 1992. doi: 10.1017/S0956796800000319.

37 R. Pressler. Loom project: Fibers and continuations for the Java virtual machine. Openjdk project, HotSpot Group, September 2017. URL `https://cr.openjdk.org/~rpressler/loom/Loom-Proposal.html`.

38 A. Reinking, N. Xie, L. de Moura, and D. Leijen. Perceus: Garbage free reference counting with reuse. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 96–111, New York, NY, USA, 2021. Association for Computing Machinery. doi: 10.1145/3453483.3454032.

39 T. Rendel, J. Trieflinger, and K. Ostermann. Automatic refunctionalization to a language with copattern matching: with applications to the expression problem. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, page 269–279, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336697. doi: 10.1145/2784731.2784763. URL `https://doi.org/10.1145/2784731.2784763`.

40 L. Rideau, B. P. Serpette, and X. Leroy. Tilting at windmills with coq: Formal verification of a compilation algorithm for parallel moves. *Journal of Automated Reasoning*, 40:307–326, 2008. doi: 10.1007/s10817-007-9096-8.

41 A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, LFP '92, page 288–298, New York, NY, USA, 1992. Association for Computing Machinery. ISBN 0897914813. doi: 10.1145/141471.141563. URL `https://doi.org/10.1145/141471.141563`.

42 P. Schuster, M. Müller, K. Ostermann, and J. I. Brachthäuser. Artifact of the paper 'Compiling Classical Sequent Calculus to Stock Hardware: The Duality of Compilation', Jan. 2025.

43 G. L. Steele. Rabbit: A compiler for scheme. Technical report, USA, 1978. URL `https://hdl.handle.net/1721.1/6913`.

44 P. Wadler. Call-by-value is dual to call-by-name. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, pages 189–201, New York, NY, USA, 2003. ACM. ISBN 1-58113-756-7. doi: 10.1145/944705.944723.

45 A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović. The risc-v instruction set manual, volume i: User-level isa, version 2.0. Technical Report UCB/EECS-2014-54, May 2014. URL `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html`.

46 A. Weiss, O. Gierczak, D. Patterson, and A. Ahmed. Oxide: The essence of rust. 2021. doi: 10.48550/arXiv.1903.00982.

47 J. Weizenbaum. Symmetric list processor. *Commun. ACM*, 6(9):524–536, sep 1963. ISSN 0001-0782. doi: 10.1145/367593.367617. URL `https://doi.org/10.1145/367593.367617`.

48 J. Weizenbaum. Recovery of reentrant list structures in slip. *Commun. ACM*, 12 (7):370–372, jul 1969. ISSN 0001-0782. doi: 10.1145/363156.363159. URL `https://doi.org/10.1145/363156.363159`.

49 N. Xie and D. Leijen. Generalized evidence passing for effect handlers: Efficient compilation of effect handlers to c. *Proc. ACM Program. Lang.*, 5(ICFP), aug 2021. doi: 10.1145/3473576.

50 N. Zeilberger. On the unity of duality. *Annals of Pure and Applied Logic*, 153(1):66–96, 2008. doi: 10.1016/j.apal.2008.01.001.

Value Typing $\boxed{\Sigma \mid \Theta \vdash\ V\ :\ \tau}$

$$\frac{\Sigma(T)\ =\ \{\ m_1(\Gamma_1),\ ...\}\qquad \Sigma \mid \Theta \vdash\ E\ :\ \Gamma\qquad \Sigma \mid \Theta \mid \Gamma_1,\ \Gamma \vdash\ s_1\qquad ...}{\Sigma \mid \Theta \vdash\ \{\ E;\ m_1(\Gamma_1) \Rightarrow s_1,\ ...\ \}\ :\ \textsf{cns}\ T}\ [\textsc{Consumer}]$$

$$\frac{\Sigma(T)\ =\ \{\ ...,\ m(\Gamma),\ ...\ \}\qquad \Sigma \mid \Theta \vdash\ E\ :\ \Gamma}{\Sigma \mid \Theta \vdash\ \{\ m;\ E\ \}\ :\ \textsf{prd}\ T}\ [\textsc{Producer}]\qquad \frac{}{\Sigma \mid \Theta \vdash\ n\ :\ \textsf{ext Int}}\ [\textsc{Integer}]$$

Environment Typing $\boxed{\Sigma \mid \Theta \vdash\ E\ :\ \Gamma}$

$$\frac{\Sigma \mid \Theta \vdash\ V\ :\ \tau\qquad ...}{\Sigma \mid \Theta \vdash\ v \mapsto V,\ ...\ :\ v\ :\ \tau,\ ...}\ [\textsc{Bindings}]$$

Configuration Typing $\boxed{\Sigma \mid \Theta \vdash\ M}$

$$\frac{\Sigma \mid \Theta \mid \Gamma \vdash\ s\qquad \Sigma \mid \Theta \vdash\ E\ :\ \Gamma\qquad \Sigma \mid \Theta \vdash\ P}{\Sigma \mid \Theta \vdash\ \langle\ s \parallel E \parallel P\ \rangle}\ [\textsc{Execution}]$$

Figure 7 Typing rules for machine states.

A Operational Semantics of AxCut

Figure 7 defines the typing rules for values, environments and machine configurations. Machine configurations are typed with signatures Σ and labels Θ in rule Execution. The program P must be well-typed in Σ and Θ, the statement s must be well-typed in Σ, Θ and a type environment Γ, and the value environment E must be well-typed as Γ in Σ and Θ. Well-typedness of a value environment E as 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. Values are also typed with signatures Σ and labels Θ. Typing for producers in rule Producer requires its value environment E to cover the parameters of the symbol m. For consumers, the typing rule Consumer requires its closure environment E to cover the common part Γ of the type environment for each clause and that the statement of each clause is well-typed in the context of its parameter list extended with Γ.

B Translation to Machine Code

Figure 8 defines the translation from AxCut to RISC-V. It uses auxiliary definitions from Figures 9 and 10. The translation is defined over typing derivations, as it uses the type environment Γ, which directly corresponds to the assignment of registers. Each variable occupies two registers, corresponding to the two components of values in the operational semantics in Figure 5. Since register `x0` is always 0 and we have three reserved registers for `temp`, `todo`, and `heap`, the first variable occupies registers `x4` and `x5`, the second one registers `x6` and `x7`, and so on. We use functions $\textsc{Reg}_1$ and $\textsc{Reg}_2$ to map variables to the registers they occupy. Consequently, there can be at most 14 variables in Γ at the same time. This restriction can be lifted, of course, by spilling any further variables to memory, as briefly described in Section 5.1.

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

Translation of Proofs $\boxed{(\Sigma \mid \Theta \vdash P) \longrightarrow I^*}$

$$\begin{array}{lcl}
\mathcal{P}[\![\ \textsf{define}\ l\ :\ \Gamma\ =\ s,\ ...\ ]\!] & = & l : \mathcal{S}[\![\ s\ ]\!] \\
 & & ...
\end{array}$$

Translation of Statements $\boxed{(\Sigma \mid \Theta \mid \Gamma \vdash s) \longrightarrow I^*}$

$$\begin{array}{lcll}
\mathcal{S}[\![\ \textsf{jump}\ l\ ]\!] & = & \texttt{j}\ l & \\
\mathcal{S}[\![\ \textsf{substitute}\ [v_1' \mapsto v_1,\ ...];\ s\ ]\!] & = & \textsc{Share}\ [v_1' \mapsto v_1,\ ...]\ \Gamma & \\
 & & \textsc{Erase}\ [v_1' \mapsto v_1,\ ...]\ \Gamma & \\
 & & \textsc{Move}\ [v_1' \mapsto v_1,\ ...] & \\
 & & \mathcal{S}[\![\ s\ ]\!] & \\
\mathcal{S}[\![\ \textsf{extern}\ \mathsf{lit}\ n\ \{\ (v) \Rightarrow s\ \}\ ]\!] & = & \texttt{li}\ (\textsc{Reg}_2\ v)\ n & \\
 & & \mathcal{S}[\![\ s\ ]\!] & \\
\mathcal{S}[\![\ \textsf{extern}\ \mathsf{add}(v_1,\ v_2)\{\ (v) \Rightarrow s\ \}\ ]\!] & = & \texttt{add}\ (\textsc{Reg}_2\ v)\ (\textsc{Reg}_2\ v_1)\ (\textsc{Reg}_2\ v_2) & \\
 & & \mathcal{S}[\![\ s\ ]\!] & \\
\mathcal{S}[\![\ \textsf{extern}\ \mathsf{ifz}(v)\{\ () \Rightarrow s_1,\ () \Rightarrow s_2\ \}\ ]\!] & = & \texttt{beq}\ (\textsc{Reg}_2\ v)\ \texttt{x0}\ l & \\
 & & \quad \mathcal{S}[\![\ s_2\ ]\!] & \\
 & & l :\ \mathcal{S}[\![\ s_1\ ]\!] & \text{where } l \text{ fresh} \\
\mathcal{S}[\![\ \textsf{let}\ v\ =\ m(\Gamma_0);\ s\ ]\!] & = & \textsc{Store}\ (\textsc{Reg}_1\ v)\ \Gamma_0 & \\
 & & \texttt{li}\ (\textsc{Reg}_2\ v)\ (\textsc{Index}\ m) & \\
 & & \mathcal{S}[\![\ s\ ]\!] & \\
\mathcal{S}[\![\ \textsf{new}\ v\ =\ (\Gamma_0)b;\ s\ ]\!] & = & \textsc{Store}\ (\textsc{Reg}_1\ v)\ \Gamma_0 & \\
 & & \texttt{la}\ (\textsc{Reg}_2\ v)\ l & \\
 & & \mathcal{S}[\![\ s\ ]\!] & \\
 & & l :\ \textsc{VTable}\ b & \text{where } l \text{ fresh} \\
\mathcal{S}[\![\ \textsf{switch}\ v\ b\ ]\!] & = & \texttt{jr}\ (\textsc{Reg}_2\ v)\ l & \\
 & & l :\ \textsc{JTable}\ b & \text{where } l \text{ fresh} \\
\mathcal{S}[\![\ \textsf{invoke}\ v\ m\ ]\!] & = & \texttt{jr}\ (\textsc{Reg}_2\ v)\ (\textsc{Index}\ m) &
\end{array}$$

Figure 8 Translation to RISC-V.

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. [40], 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 how many times a variable from the old environment 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 integer, which has no associated memory block in its first register. Therefore, ShareBlock first checks whether the register indeed points to a memory block. If so, it loads the reference count from the first field of the block, increases it, and then stores it back to memory. 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 then simply decremented. If the reference count is

Auxiliary Definitions

$\textsc{Share}\ [...]\ (..., v : \tau) = \textbf{if}\ \textsc{NumRefs}\ v\ [...] > 1$
$\quad \textsc{ShareBlock}\ (\textsc{Reg}_1\ v)\ (\textsc{NumRefs}\ v\ [...] - 1)$
$\quad ...$

$\textsc{Erase}\ [...]\ (..., v : \tau) = \textbf{if}\ \textsc{NumRefs}\ v\ [...] = 0$
$\quad \textsc{EraseBlock}\ (\textsc{Reg}_1\ v)$
$\quad ...$

$\textsc{ShareBlock}\ r\ n =$ where l fresh

```
beq r x0 l
   lw temp r 0
   addi temp temp n
   sw temp r 0
l :
```

$\textsc{EraseBlock}\ r =$ where $l_1,\ l_2$ fresh

```
beq r x0 l1
   lw temp r 0
   beq temp x0 l2
      addi temp temp -1
      sw temp r 0
      j l1
   l2 : sw todo r 0
      mv todo r
l1 :
```

$\textsc{Reg}_1\ v$	first register of variable v
$\textsc{Reg}_2\ v$	second register of variable v
$\textsc{Index}\ m$	four times index of symbol m 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
$\textsc{Move}\ [v'_1 \mapsto v_1,\ ...]$	parallel moves for assignments $[v'_1 \mapsto v_1,\ ...]$
$\textsc{NumRefs}\ v\ [v'_1 \mapsto v_1,\ ...]$	number of variables v is assigned to in $[v'_1 \mapsto v_1,\ ...]$
$\textsc{ShareFields}\ r$	$\textsc{ShareBlock}\ f\ 1$ 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 todo list
`heap`	reserved register for free list

Figure 9 Auxiliary definitions for the translation.

zero, the block is prepended to the lazy free list pointed to by `todo`. The link to the next block in the list is stored in the reference count field of the block. The other fields, however, are not erased recursively.

We translate extern statements depending on the target platform. For example, integer literals are loaded with an `li` instruction and addition becomes an `add` instruction with the appropriate registers. The conditional ifz is translated to a conditional branching instruction `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. In congruence with the abstract machine steps, the duality of producers and consumers leads to dual translations.

Auxiliary Definitions

JTable { $m_1(\Gamma_1) \Rightarrow s_1, \ ...\}$	=	`j` l_1 `j` l_2 ... l_1 : Load (Reg$_1$ x) Γ_1 $\mathcal{S}[\![s_1]\!]$ l_2 : Load (Reg$_1$ x) Γ_2 ...	where $l_1, l_2, \ldots$ fresh where x is fresh after Γ where x is fresh after Γ
VTable { $m_1(\Gamma_1) \Rightarrow s_1, \ ...\}$	=	`j` l_1 `j` l_2 ... l_1 : Load (Reg$_1$ x) Γ_0 $\mathcal{S}[\![s_1]\!]$ l_2 : Load (Reg$_1$ x) Γ_0 ...	where $l_1, l_2, \ldots$ fresh where x is fresh after Γ_1 where x is fresh after Γ_2
Load r (..., v : τ)	=	Release r `lw` (Reg$_2$ v) r (Offset$_2$ v) `lw` (Reg$_1$ v) r (Offset$_1$ v) ...	
Store r (..., v : τ)	=	`sw` (Reg$_2$ v) `heap` (Offset$_2$ v) `sw` (Reg$_1$ v) `heap` (Offset$_1$ v) ... Acquire r	

Acquire r	=	`mv` r `heap`
		`lw heap heap 0`
		`beq heap x0` l_1
		`sw x0` r `0`
		`j` l_2
		l_1 : `mv heap todo`
		`lw todo todo 0`
		`beq todo x0` l_3
		`sw x0 heap 0`
		EraseFields `heap`
		`j` l_2
		l_3 : `addi todo heap 32`
		l_2 :
			where l_1, l_2, l_3 fresh

Release r	=	`lw temp` r `0`
		`beq temp x0` l_1
		`addi temp temp -1`
		`sw temp` r `0`
		ShareFields r
		`j` l_2
		l_1 : `sw heap` r `0`
		`mv heap` r
		l_2 :
			where l_1, l_2 fresh

Figure 10 Auxiliary definitions for the translation (continued).

We translate the construction of a producer by first storing the registers corresponding to the environment Γ_0 into the free block of memory in register `heap`. Doing so moves the address in `heap` into the first register of the variable v. Then, the second register of its variable v 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, a new free block of memory has to be acquired into register `heap`, as the previously free block is now in use. For this, Acquire distinguishes three cases. If the 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 not contain a block either, which is indicated by a `0` at this memory location, we fall back to bump-allocating from the remaining big block of memory. If the lazy free list 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. In our actual implementation, we use equal-size memory blocks to obtain constant-time memory management. The necessary plumbing for zeroing unused fields and linking together multiple blocks in case more than one block is needed, is not shown here.

Dually to the construction of a producer, we translate the construction of a consumer to a virtual table l, which is placed right after the code for the remaining statement. We translate each clause to first load the closure environment Γ_0 and then execute the translated statement. 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 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 now is an additional reference to them in the registers they are loaded into. 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 v. We then load the address of the virtual table into the second register of v.

We translate the destruction of a producer bound to variable v 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 clause, and each entry is a jump to a fresh label for the corresponding clause. We translate each clause first to the corresponding label, then to loading the environment Γ_i, and finally the translation of the clause statement.

We translate the destruction of a consumer v to an indirect jump to the address in the second register of v, adding the index of the tag. Restoring the closure environment is taken care of at the destination. The arguments of the observation are already in the corresponding registers, as ensured by our typing rules.

The code for constructing and destructing producers and consumers is indeed dual. Both **switch** and **invoke** perform indirect jumps. But in one of them, the offset is unknown, and in the other one, the jump table is unknown, following the operational semantics in Section 3.3. Similarly, both **let** and **new** store values onto the heap, but in one of them, we pair the block of memory with a tag, and in the other one, with a jump table. This leads to dual optimizations; for example, when a data type has a single constructor, or a codata type has a single destructor, the jump table has a single entry, and we could avoid generating it at all.

C Benchmarks

We conducted microbenchmarks and compare the running time with a selection of industrial and research languages. All benchmarks were conducted on a 12th Gen Intel(R) Core(TM)

i7-1255U running Ubuntu 22.04. We measured the running times using the command-line benchmarking tool `hyperfine` [35], which we configured to perform 10 warmup runs before taking the measurements.

C.0.1 Systems

We compare the running time of a range of different systems: AxCut, MLton, OCaml, Koka, 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 to the respective language. For AxCut we used the surface language **Fun** presented in [6] and manually applied a translation to sequent calculus which avoids administrative redexes. We then manually applied the transformation to AxCut presented in Section 2.2. Finally, we ran our implementation of code generation from AxCut to machine code presented in Section 4. We did not apply any optimizations.

MLton[3] (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[4] (5.0.0) directly generates machine code from its intermediate representations, but does not perform many optimizations. Like MLton, it uses generational garbage collection, with a mix of copying and mark-and-sweep.

Koka [28] (3.1.1) generates C code [49] and invokes a C compiler to generate machine code. It uses garbage-free reference counting [38] and optimizes programs to use in-place mutation and tail-recursion modulo cons [30]. Since it performs significant optimizations and we aim to measure the runtime costs and not how well optimizations are performed, we report results for both `koka` (*i.e.*, `-O0`) and `koka_opt` (*i.e.*, `-O2`).

Rust[5] (1.76.0) generates LLVM code[6] and invokes the LLVM toolchain to optimize programs and generate machine code. It uses a mix of linearity and borrowing [46] 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, LLVM performs significant optimizations and again we report results for both `rust` (*i.e.*, `-O0`) and `rust_opt` (*i.e.*, `-O3`).

Since many of our benchmarks use the stack and we run them on large inputs, we have to increase the default stack size for all languages other than AxCut, which does not use the C stack.

C.0.2 Programs

We measure the running time of various programs. They are simple and do not make use of advanced control flow features, since support for those wildly varies across the languages we compare with. Rather, each of them is designed to measure a certain aspect of a functional programming language, parametrized by a number N.

`factorial_accumulator` Computes the factorial of N modulo a magic number (to avoid integer overflows) with a loop. Assesses loops and operations on integer values.

[3] http://mlton.org
[4] https://ocaml.org
[5] https://www.rust-lang.org
[6] https://llvm.org

`fibonacci_recursive` Computes the Nth Fibonacci number with double recursion. Assesses pushing and popping of stack frames.

`sum_range` Computes the sum of numbers up to N by first building a linked list of them and then recursively summing them up. Assesses allocation and linear use of data types.

`iterate_increment` Computes the number N with a higher-order function that iterates its function argument. Assesses higher-order functions and indirect calls.

`match_options` Computes the number N with a chain of pattern matches on the call stack. Assesses returns to a pattern matching context.

`lookup_tree` Computes the number N by building a binary tree with shared nodes and then looking up the left-most leaf. Assesses allocation and non-linear use of data types.

`erase_unused` Computes the number N by unnecessarily allocating linked lists of increasing size and immediately erasing them. Assesses deallocation of data types.

While each of these programs is relatively small, we believe that together they provide a good first overview of the performance of various language features.

C.0.3 Results

Figure 11 shows the detailed results of our benchmark programs for the different systems for linearly increasing values of the input N.

In general, we can see that the fairly simplistic compilation of AxCut is comparable with state-of-the-art compilers. However, we can also see that compile-time optimizations make a significant difference. This, for instance, can be observed in the results for `iterate_increment`, where `koka_opt` and `rust_opt` both inline the function argument and are constantly under 1ms, defeating the purpose of measuring the cost of an indirect jump. Their unoptimized counterparts, however, perform worse than AxCut on this specific benchmark. While AxCut does not perform optimizations, this comparison might not be fair, since the other implementations might rely on trivial optimizations to remove administrative redexes introduced by the compiler. The performance of OCaml in `sum_range`, `match_options`, and `lookup_tree` is visibly worse than the other languages. This is likely due to their use of a garbage collector. Moreover, it should be noted that in `lookup_tree` and `erase_unused`, 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. For `fibonacci_recursive`, the performance of AxCut is worse than all others. The reason for this likely is that, due to our simplistic layout of memory blocks, we zero out all unused fields in each memory object, which leads to many unnecessary stores.

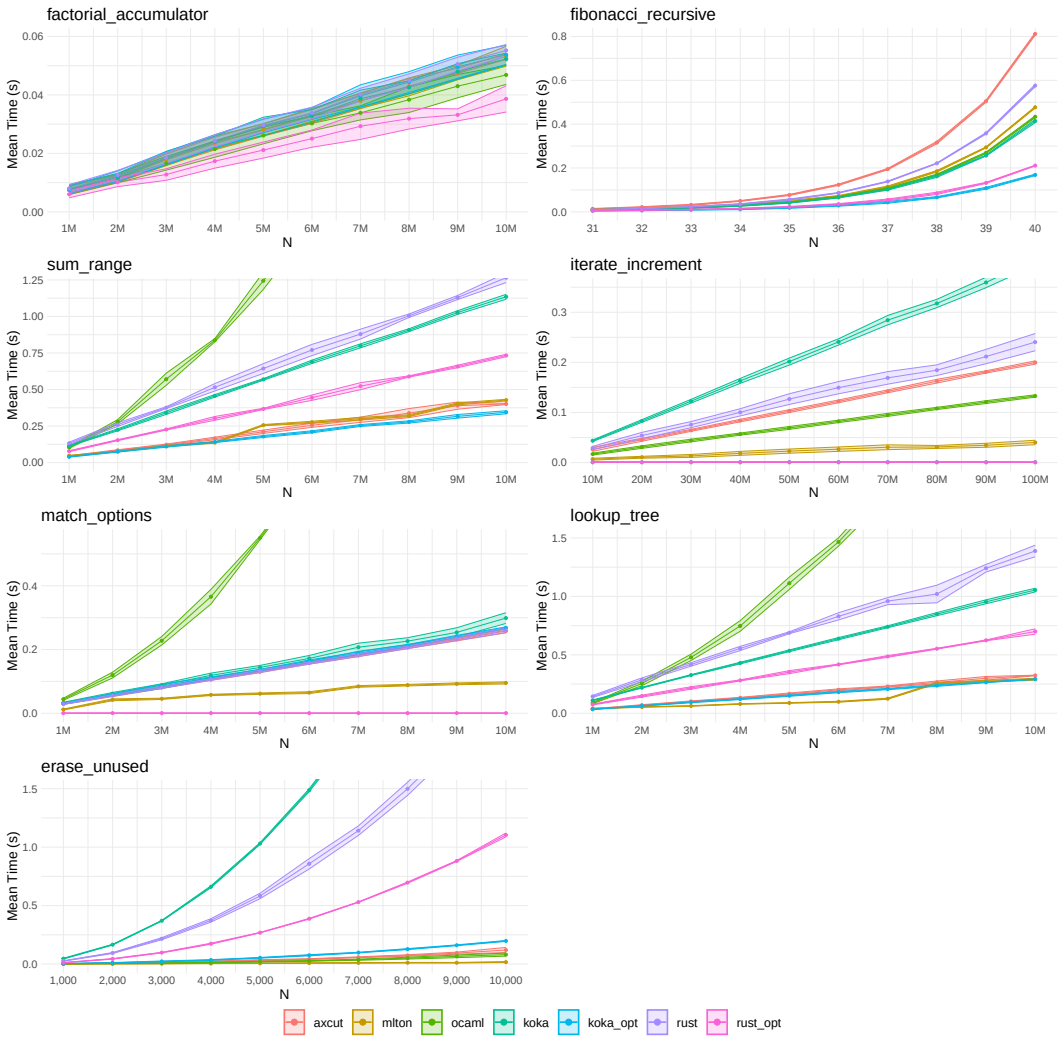


Figure 11 Performance results for the various systems in a linearly increasing value for N.