Compiling with the Sequent Calculus

A Unifying Approach for Control Effects and Codata Types

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

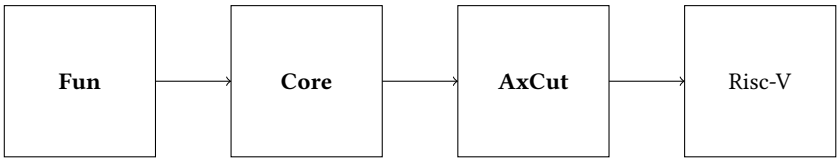
Compiling a high-level functional programming language to machine code that can be executed efficiently on a modern machine is a complicated task, since we have to traverse many different levels of abstraction. This is particularly challenging if the language contains some form of control effects and a mix of different calling conventions, such as call-by-value data types and call-by-name codata types. In this paper we tell the complete story, starting from a simple functional programming language with control effects and both data and codata types, and ending up with Risc-V and x86-64 machine code. The novelty of our approach lies in the fact that we use the sequent calculus, and sequent calculus inspired languages, in specifying the intermediate stages of our compiler. In that sense, we view this work as a continuation, and generalization, of Andrew Appel's landmark work on "Compiling with Continuations".

CCS Concepts: • **Theory of computation → Lambda calculus**; • **Software and its engineering → Compilers**; *Control structures.*

Additional Key Words and Phrases: Intermediate representations, continuations, codata types, control effects

1 Introduction

Compiling a modern functional programming language like Haskell or OCaml to efficient machine code is a complicated affair, since we have to cross many different levels of abstraction in order to bridge the gap between high-level conveniences and low-level concerns. In this paper we present such a complete compilation pipeline, starting with a surface language which supports different calling conventions, codata types and control effects, and ending up with machine code for Risc-V and X86-64. More concretely, we show how to implement all of the following stages in the compiler pipeline:



The rest of this article is structured as follows:

- In section 2 we introduce the surface language **Fun**.
- In section 3 we introduce the intermediate language **Core**.

Authors' Contact Information: David Binder, Department of Computer Science, University of Tübingen, Tübingen, Germany, david.binder@uni-tuebingen.de; Marco Tzschentke, Department of Computer Science, University of Tübingen, Tübingen, Germany, marco.tzschentke@uni-tuebingen.de; Marius Müller, Department of Computer Science, University of Tübingen, Tübingen, Germany, mari.mueller@uni-tuebingen.de; Philipp Schuster, Department of Computer Science, University of Tübingen, Tübingen, Germany, philipp.schuster@uni-tuebingen.de; Jonathan Immanuel Brachthäuser, Department of Computer Science, University of Tübingen, Tübingen, Germany, jonathan.brachthaeuser@uni-tuebingen.de; Klaus Ostermann, Department of Computer Science, University of Tübingen, Tübingen, Germany, klaus.ostermann@uni-tuebingen.de.

- In section 4 we show how to translate programs from the surface language **Fun** to the intermediate language **Core**.
- In section 5 we show a transformation on programs in **Core** which names all intermediate computations, similar to ANF or focusing transformations.
- In section 6 and section 7 we ...
- In section 8 we introduce the low-level intermediate language **AxCut**, and in section 9 we show how to compile **Core** to **AxCut**.

We discuss related work in section 10 and conclude in section 11.

2 The Surface Language Fun

In this section we are going to introduce the language **Fun** that we want to compile to machine code. This language is in many respects an ordinary expression-oriented, functional programming language. For example, we do not support mutable state or object-oriented mechanisms such as inheritance or dynamic dispatch. But the language is also more interesting than just the lambda calculus, from which it is distinguished by several features. These features were chosen to illustrate specific aspects of the compilation through the sequent calculus, and we are going to introduce all of them in this chapter. The syntax of the entire language **Fun** is given in definition 2.1.

*Definition 2.1 (Syntax of **Fun**).* We assume an infinite set of names $\mathcal{N}$ containing type names $T \in \mathcal{N}$, constructor names $K \in \mathcal{N}$ and destructor names $D \in \mathcal{N}$.

$$
\begin{array}{rcll}
\odot & \coloneqq & + \mid - \mid * & \textit{Arithmetic Operators} \\
t & \coloneqq & x \mid \ulcorner n \urcorner \mid t \odot t \mid \mathbf{ifz}(t,t,t) \mid \mathbf{let}\ x = t\ \mathbf{in}\ t \mid \mathbf{label}\ \alpha\ \{t\} \mid \mathbf{goto}(t;\alpha) & \textit{Terms} \\
 & \mid & K\,\sigma \mid \mathbf{case}\ t\ \mathbf{of}\ \{\overline{K\ \Gamma \Rightarrow t}\} \mid t.D\,\sigma \mid \mathbf{cocase}\ \{\overline{D\ \Gamma \Rightarrow t}\} \mid f\,\sigma & \\
c & \coloneqq & \alpha & \textit{Consumers} \\
\sigma & \coloneqq & \Diamond \mid \sigma, t \mid \sigma, c & \textit{Substitutions} \\
\tau & \coloneqq & \mathbf{Int} \mid T & \textit{Types} \\
\kappa & \coloneqq & \mathbf{prim} \mid \mathbf{data} \mid \mathbf{codata} & \textit{Kinds} \\
\Gamma & ::= & \Diamond \mid \Gamma, x : \tau \mid \Gamma, \alpha :^{\text{cns}} \tau & \textit{Typing Contexts} \\
\delta & \coloneqq & \mathbf{data}\,T\,\{\overline{K\ \Gamma}\} \mid \mathbf{codata}\,T\,\{\overline{D\ \Gamma : \tau}\} \mid \mathbf{def}\ f\ \Gamma : \tau \coloneqq t & \textit{Declarations} \\
\Theta & \coloneqq & \overline{\delta} & \textit{Programs}
\end{array}
$$

Terms t contain variables x and non-recursive let-expressions $\mathbf{let}\ x = t_1\ \mathbf{in}\ t_2$, both of which are completely standard.

Three constructs in the language of terms pertain to operations on integers. There are integer literals $\ulcorner n \urcorner$, binary arithmetic operations $t_1 + t_2$, $t_1 - t_2$ and $t_1 * t_2$ (abbreviated as $t_1 \odot t_2$) and the conditional $\mathbf{ifz}(t_1, t_2, t_3)$. This conditional expression evaluates to t_2 if t_1 evaluates to $\ulcorner 0 \urcorner$, and to t_3 otherwise. These arithmetic operations are contained in the language because there is a straight-forward correspondence to operations available on modern instruction sets.

The two terms $\mathbf{label}\ \alpha\ \{t\}$ and $\mathbf{goto}(t;\alpha)$ are control operators which respectively capture and instantiate a continuation α. The language **Fun** contains these control operators to illustrate that the sequent calculus is naturally suited as a compilation target for languages which have control operators.

Practically every statically typed functional programming language provides a mechanism to defined algebraic data types. They pose the problem, from a compiler implementors perspective, on how elements of these data types should

be layed out in memory, and how pattern matching on these elements should be implemented. In the term language we have constructors $K\,\sigma$ and pattern-matching expressions **case** t **of** $\{\overline{K\;\Gamma \Rightarrow t}\}$.

In both the definitions for (co-) cases and for declarations we use typing contexts Γ to encode the arguments of constructors and destructors. This is a slight abuse of notation, since in data declarations, types for arguments are mandatory, while for clauses in (co-) cases, they are omitted. Thus, the following is a valid program in **Fun**.

Example 2.2. Consider the example of lists of integers:

```
data ListInt { Nil, Cons(x:Int, xs:ListInt) }
def tl(ls) := case ls of { Nil => Nil, Cons(x,xs) => xs }
```

Here, the constructor `Cons` has arguments x and xs, both of which are type annotated in the declaration of `ListInt`, but not in the case expression.

2.1 Typing Rules

Typing terms in **Fun** is split into five different judgement forms. The forms $\Theta \mid \Gamma \vdash t : \tau$ and $\Theta \mid \Gamma \vdash \sigma : \Gamma'$ are both used to type terms. We use the subsititution rules to easily check arguments for constructors/destructors and if clauses in cases/cocases are well-formed. For these rules, the order of variables in Γ and terms in σ is important, as we always want to keep the order of arguments defined in a declaration.

Example 2.3. Take the example program in example 2.2. In this case, we have following program

$$\Theta = \textbf{data}\,\texttt{Listint}\,\{\texttt{Nil}\;\Gamma_1, \texttt{Cons}\;\Gamma_2\} \quad \Gamma_1 = \diamond, \Gamma_2 = x : \texttt{Int}, xs : \texttt{ListInt}.$$

When we then want to type check the body of `tl`, within the program Θ, we have to ensure the contexts for the constructors `Nil` and `Cons` in the case-expression match Γ_1 and Γ_2 in the data declaration (which in this case works). On the other hand, if we have a term $\mathrm{Cons}(1, \mathrm{Nil})$, we have the substitution $\sigma = 1, \texttt{Nil}$, and type checking this term amounts to checking $\Theta \mid \Gamma \vdash 1 : \texttt{Int}$ and $\Theta \mid \Gamma \vdash \texttt{Nil} : \texttt{ListInt}$.

Next, instead of having a single kind $*$ for the inhabited kind, we instead have this split into three distinct kinds. The **prim** kind should be self-explanatory, as it is often used for more efficient handling of primitive types such as integers or floats. On the other hand, **data** and **codata** are less standard, even in sequent calculus based languages. We have included them in the language, since, as we will later see, data and codata types are handled differently when being translated to machine code, and thus we always need to keep track of them.
Lastly, we have the judgements $\vdash \Theta\ \textsc{Ok}$ and $\Theta \vdash \Gamma\ \textsc{Ctx}$ used to ensure typing contexts and programs are well-formed, i.e. that all occurring terms can be typed and all occuring terms can be kinded. The one additional check that needs to happen during the well-formedness checks is name checking. That is, we need to ensure all types T, constructors K, destructors D and top-level definitions f are only defined once and there are no name clashes.

3 The Intermediate Language Core

The language **Core**, introduced in this section, is the first of the languages which is directly influenced by the sequent calculus. The skeleton of **Core** is formed by the $\lambda\mu\tilde{\mu}$-calculus-calculus of **?**. They, however, did not yet consider arbitrary data and codata types in their system, which were formalized by TODO. The syntax of **Core** is given in definition 3.1.

Typing Terms $\Theta \mid \Gamma \vdash t : \tau$

$$\frac{x : \tau \in \Gamma}{\Theta \mid \Gamma \vdash x : \tau}\ \textsc{Var} \qquad \frac{}{\Theta \mid \Gamma \vdash \ulcorner n \urcorner : \mathbf{Int}}\ \textsc{Lit} \qquad \frac{\Theta \mid \Gamma \vdash t_1 : \mathbf{Int} \quad \Theta \mid \Gamma \vdash t_2 : \mathbf{Int}}{\Theta \mid \Gamma \vdash t_1 \odot t_2 : \mathbf{Int}}\ \textsc{Op}$$

$$\frac{\Theta \mid \Gamma \vdash t : \mathbf{Int} \qquad \Theta \mid \Gamma \vdash t_1 : \tau \qquad \Theta \mid \Gamma \vdash t_2 : \tau}{\Theta \mid \Gamma \vdash \mathbf{ifz}(t, t_1, t_2) : \tau}\ \textsc{Ifz} \qquad \frac{\Theta \mid \Gamma \vdash t_1 : \tau_1 \qquad \Theta \mid \Gamma, x : \tau_1 \vdash t_2 : \tau_2}{\Theta \mid \Gamma \vdash \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2 : \tau_2}\ \textsc{Let}$$

$$\frac{\alpha :^{\text{cns}} \tau \in \Gamma \qquad \Theta \mid \Gamma \vdash t : \tau}{\Theta \mid \Gamma \vdash \mathbf{label}\ \alpha\ \{t\} : \tau}\ \textsc{Label} \qquad \frac{\Theta \mid \Gamma \vdash t : \tau \qquad \alpha :^{\text{cns}} \tau \in \Gamma}{\Theta \mid \Gamma \vdash \mathbf{goto}(t; \alpha) : \tau'}\ \textsc{Goto}$$

$$\frac{\mathbf{data}\ T\ \{K\ \Gamma', \ldots\} \in \Theta \qquad \Theta \mid \Gamma \vdash \sigma : \Gamma'}{\Theta \mid \Gamma \vdash K\ \sigma : T}\ \textsc{Constructor} \qquad \frac{\mathbf{data}\ T\ \{\overline{K_i\ \Gamma_i}\} \in \Theta \qquad \overline{\Theta \mid \Gamma, \Gamma_i \vdash t_i : \tau}}{\Theta \mid \Gamma \vdash \mathbf{case}\ t\ \mathbf{of}\ \{\overline{K_i\ \Gamma_i \Rightarrow t_i}\} : \tau}\ \textsc{Case}$$

$$\frac{\mathbf{codata}\ T\ \{D\ \Gamma' : \tau, \ldots\} \in \Theta \qquad \Theta \mid \Gamma \vdash \sigma : \Gamma' \qquad \Theta \mid \Gamma \vdash t : T}{\Theta \mid \Gamma \vdash t.D\ \sigma : \tau}\ \textsc{Destructor}$$

$$\frac{\mathbf{codata}\ T\ \{\overline{D_i\ \Gamma_i : \tau_i}\} \in \Theta \qquad \Theta \mid \Gamma, \Gamma_i \vdash t_i : \tau_i}{\Theta \mid \Gamma \vdash \mathbf{cocase}\ \{\overline{D_i\ \Gamma_i \Rightarrow t_i}\} : T}\ \textsc{Cocase} \qquad \frac{\mathbf{def}\ f\ \Gamma' : \tau \in \Theta \qquad \Theta \mid \Gamma \vdash \sigma : \Gamma'}{\Theta \mid \Gamma \vdash f\ \sigma : \tau}\ \textsc{Call}$$

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

$$\frac{\alpha :^{\text{cns}} \tau \in \Gamma}{\Theta \mid \Gamma \vdash \alpha :^{\text{cns}} \tau}\ \textsc{Var}_2$$

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

$$\frac{}{\Theta \mid \Gamma \vdash \Diamond : \Diamond}\ \textsc{Subst}_1 \qquad \frac{\Theta \mid \Gamma \vdash \sigma : \Gamma' \quad \Theta \mid \Gamma \vdash t : \tau}{\Theta \mid \Gamma \vdash \sigma, t : \Gamma', x : \tau}\ \textsc{Subst}_2 \qquad \frac{\Theta \mid \Gamma \vdash \sigma : \Gamma' \quad \Theta \mid \Gamma \vdash c :^{\text{cns}} \tau}{\Theta \mid \Gamma \vdash \sigma, c : \Gamma', \alpha :^{\text{cns}} \tau}\ \textsc{Subst}_3$$

Kinding Types $\Theta \vdash \tau : \kappa$

$$\frac{}{\Theta \vdash \mathbf{Int} : \mathbf{prim}}\ \textsc{PrimKind} \qquad \frac{\mathbf{data}\ T\ \{\ldots\} \in \Theta}{\Theta \vdash T : \mathbf{data}}\ \textsc{DataKind} \qquad \frac{\mathbf{codata}\ T\ \{\ldots\} \in \Theta}{\Theta \vdash T : \mathbf{codata}}\ \textsc{CodataKind}$$

Well-formed Programs $\vdash \Theta\ \textsc{Ok}$

$$\frac{}{\vdash \Diamond\ \textsc{Ok}}\ \textsc{Wf-Empty} \qquad \frac{\vdash \Theta\ \textsc{Ok} \qquad \overline{\Theta, \mathbf{codata}\ T\ \{\ldots\} \vdash \Gamma_i\ \textsc{Ctx}} \qquad \overline{\Gamma_i \vdash \tau_i : \kappa}}{\vdash \Theta, \mathbf{codata}\ T\ \{\overline{D_i\ \Gamma_i : \tau_i}\}\ \textsc{Ok}}\ \textsc{Wf-Codata}$$

$$\frac{\vdash \Theta\ \textsc{Ok} \qquad \Theta, \mathbf{def}\ f\ \Gamma : \tau := t \mid \Gamma \vdash t : \tau}{\vdash \Theta, \mathbf{def}\ f\ \Gamma : \tau := t\ \textsc{Ok}}\ \textsc{Wf-Def} \qquad \frac{\vdash \Theta\ \textsc{Ok} \qquad \overline{\Theta, \mathbf{data}\ T\ \{\ldots\} \vdash \Gamma_i\ \textsc{Ctx}}}{\vdash \Theta, \mathbf{data}\ T\ \{\overline{K_i\ \Gamma_i}\}\ \textsc{Ok}}\ \textsc{Wf-Data}$$

Well-formed Contexts $\Theta \vdash \Gamma\ \textsc{Ctx}$

$$\frac{}{\Theta \vdash \Diamond\ \textsc{Ctx}}\ \textsc{Ctx}_1 \qquad \frac{\Theta \vdash \Gamma\ \textsc{Ctx} \quad x \notin \Gamma \quad \Theta \vdash \tau : \kappa}{\Theta \vdash \Gamma, x : \tau\ \textsc{Ctx}}\ \textsc{Ctx}_2 \qquad \frac{\Theta \vdash \Gamma\ \textsc{Ctx} \quad \alpha \notin \Gamma \quad \Theta \vdash \tau : \kappa}{\Theta \vdash \Gamma, \alpha :^{\text{cns}} \tau\ \textsc{Ctx}}\ \textsc{Ctx}_3$$

Fig. 1. Typing rules for **Fun**

*Definition 3.1 (Syntax of **Core**).*

$$
\begin{array}{rcll}
p & \coloneqq & x \mid \ulcorner n \urcorner \mid \mu\alpha.s \mid K\ \sigma \mid \mathbf{cocase}\ \{\overline{D\ \Gamma \Rightarrow s}\} & \textit{Producers} \\
c & \coloneqq & \alpha \mid \tilde{\mu}x.s \mid D\ \sigma \mid \mathbf{case}\ \{\overline{K\ \Gamma \Rightarrow s}\} & \textit{Consumers} \\
s & \coloneqq & \langle p \mid c \rangle \mid \mathbf{ifz}(p, s, s) \mid \odot(p, p; c) \mid f\ \sigma \mid \mathbf{done} & \textit{Statements} \\
\sigma & \coloneqq & \Diamond \mid \sigma, p \mid \sigma, c & \textit{Substitutions} \\
\tau & \coloneqq & \mathbf{Int} \mid T & \textit{Types} \\
\kappa & \coloneqq & \mathbf{data} \mid \mathbf{codata} \mid \mathbf{prim} & \textit{Kinds} \\
\Gamma & \coloneqq & \emptyset \mid \Gamma, x : \tau \mid \Gamma, \alpha :^{\text{cns}} \tau & \textit{Typing Contexts} \\
\delta & \coloneqq & \mathbf{data}\ T\ \{\overline{K\ \Gamma}\} \mid \mathbf{codata}\ T\ \{\overline{D\ \Gamma}\} \mid \mathbf{def}\ f\ \Gamma \coloneqq s & \textit{Declarations} \\
\Theta & \coloneqq & \overline{\delta} & \textit{Programs}
\end{array}
$$

The most important difference between the language Fun and the language **Core** are the different syntactic categories. While Fun only had terms, **Core** has producers, consumers and statements. As a simple approximation, terms in Fun correspond to producers in **Core**, consumers correspond to continuations and statements correspond to computations. In section 4 we will explain this in more detail.

3.1 Typing Rules

As before, there are a number of judgements used to type expressions in **Core**. Most of them are analogous to Fun, but since we now have producers, consumers and statements instead of terms, the typing judgement for terms has been split into three different ones.

4 Translating Fun to Core

In this section we show how to translate terms of the surface language **Fun** to producers of the intermediate language **Core**. We first introduce a simpler translation $[\![\cdot]\!]$ in section 4.1 which is straightforward but introduces administrative redexes. We then introduce the optimized translation $[\![\cdot]\!]^*$ in section 4.2 whis is more complicated, but which does not generate any additional administrative redexes. We finish in section 4.3 by proving some core properties of these translation functions.

$$[\![\mathbf{codata}\ T\ \{\overline{D_i\ \Gamma_i : \tau}\}]\!] \quad \coloneqq \quad \mathbf{codata}\ T\ \{\overline{D_i\ \Gamma_i, \alpha :^{\text{cns}} \tau}\} \quad (\alpha \text{ fresh})$$

Translating Declarations and typing contexts are the same for both translations we will introduce. Here, the most important part of the translation is the fact that codata declarations no longer have a type τ used for destructor terms. Instead, each destructor gets a fresh covariable argument α with this type. As we will see in the translations for cocases, this will ensure types are preserved under translation.

4.1 Naive Translation

First, we will introduce the naive translation, turning terms in **Fun** into producers in **Core**. As we can see from the definitions of **Core** (section 3), certain terms, for example cases and destructors, are consumers in **Core**, and others, such as if zero and top-level calls are statements. In order to correctly translate such terms to **Core**, we thus introduce μ-abstractions, turning statements into producers. Where **Core**-expressions are consumers, they are cut with some other producer to turn them into statements which then make a producer with the μ-abstraction. We assume that each

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

$$\frac{\Theta \mid \Gamma, \alpha :^{\text{cns}} \tau \vdash s}{\Theta \mid \Gamma \vdash \mu\alpha.s : \tau}\ \mu \qquad \frac{\mathbf{codata}\ T\ \{\overline{D_i\ \Gamma_i}\} \in \Gamma \qquad \overline{\Theta \mid \Gamma, \Gamma_i \vdash s_i}}{\Theta \mid \Gamma \vdash \mathbf{cocase}\ \{\overline{D_i\ \Gamma_i \Rightarrow s_i}\} : T}\ \textsc{Cocase}$$

The rules Var, Constructor and Lit are identical to the ones in fig. 1.

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

$$\frac{\Theta \mid \Gamma, x : \tau \vdash s}{\Theta \mid \Gamma \vdash \tilde{\mu}x.s :^{\text{cns}} \tau}\ \tilde{\mu}$$

$$\frac{\mathbf{codata}\ T\ \{D\ \Gamma', \ldots\} \in \Gamma \qquad \Theta \mid \Gamma \vdash \sigma : \Gamma'}{\Theta \mid \Gamma \vdash D\ \sigma :^{\text{cns}} T}\ \textsc{Destructor} \qquad \frac{\mathbf{data}\ T\ \{\overline{K_i\ \Gamma_i}\} \in \Gamma \qquad \overline{\Theta \mid \Gamma, \Gamma_i \vdash s_i}}{\Theta \mid \Gamma \vdash \mathbf{case}\ \{\overline{K_i\ \Gamma_i \Rightarrow s_i}\} :^{\text{cns}} T}\ \textsc{Case}$$

The rule Var$_2$ is identical to the one in fig. 1

Well-Typed Statements $\Theta \mid \Gamma \vdash s$

$$\frac{\Theta \mid \Gamma \vdash p : \tau \qquad \Theta \mid \Gamma \vdash c :^{\text{cns}} \tau}{\Theta \mid \Gamma \vdash \langle p \mid c \rangle}\ \textsc{Cut} \qquad \frac{\Theta \mid \Gamma \vdash p : \mathbf{Int} \qquad \Theta \mid \Gamma \vdash s_1 \qquad \Theta \mid \Gamma \vdash s_2}{\Theta \mid \Gamma \vdash \mathbf{ifz}(p, s_1, s_2)}\ \textsc{IfZ} \qquad \frac{}{\Theta \mid \Gamma \vdash \mathbf{done}}\ \textsc{Done}$$

$$\frac{\Theta \mid \Gamma \vdash p_1 : \mathbf{Int} \qquad \Theta \mid \Gamma \vdash p_2 : \mathbf{Int} \qquad \Theta \mid \Gamma \vdash c :^{\text{cns}} \mathbf{Int}}{\Theta \mid \Gamma \vdash \odot(p_1, p_2; c)}\ \textsc{binop} \qquad \frac{\mathbf{def}\ f\ \Gamma' \in \Theta \qquad \Theta \mid \Gamma \vdash \sigma : \Gamma'}{\Theta \mid \Gamma \vdash f\ \sigma}\ \textsc{Call}$$

Well-formed Programs $\vdash \Theta$ Ok

$$\frac{\vdash \Theta\ \textsc{Ok} \qquad \overline{\Theta, \mathbf{codata}\ T\ \{\ldots\} \vdash \Gamma_i\ \textsc{Ctx}}}{\vdash \Theta, \mathbf{codata}\ T\ \{\overline{D_i\ \Gamma_i}\}\ \textsc{Ok}}\ \textsc{Wf-Codata} \qquad \frac{\vdash \Theta\ \textsc{Ok} \qquad \Theta, \mathbf{def}\ f\ \Gamma := s \mid \Gamma \vdash s}{\vdash \Theta, \mathbf{def}\ f\ \Gamma := s\ \textsc{Ok}}\ \textsc{Wf-Fun}$$

The rules Wf-Data and Wf-Empty are identical to the ones in fig. 1

$\Theta \mid \Gamma \vdash \sigma : \Gamma'$ identical to fig. 1 | $\Theta \mid \Gamma \vdash \tau : \kappa$ identical to fig. 1 | $\Theta \vdash \Gamma$ Ctx identical to fig. 1

Fig. 2. Typing Rules for **Core**

covariable α appearing in a translated term is fresh with respect to the term that is translated (except for those already present in the source term). For example, when translating a top-level call $f(\overline{t_i}; \overline{\alpha_i})$, the α_j used in the translation does not appear free in any of the t_i and is not equal to any of the α_j. Additionally, whenever a typing context Γ appears on the left-hand side, we also write Γ for the corresponding typing context in **Core**, since for translating contexts there is nothing to do. Any context Γ containing variables $x : \tau$ and covariables $\alpha :^{\text{cns}} \tau$ is either valid in both **Fun** and **Core**

or in neither, so we can keep the same contexts.

$$[\![\cdot]\!] : \mathit{Term} \to \mathit{Producer}$$

$$\begin{array}{rcl@{\qquad}rcl}
[\![x]\!] & \coloneqq & x & [\![\mathbf{ifz}(t_1,t_2,t_3)]\!] & \coloneqq & \mu\alpha.\mathbf{ifz}([\![t_1]\!],\langle[\![t_2]\!]\mid\alpha\rangle,\langle[\![t_3]\!]\mid\alpha\rangle)\\
[\![\ulcorner n \urcorner]\!] & \coloneqq & \ulcorner n \urcorner & [\![t_1 \odot t_2]\!] & \coloneqq & \mu\alpha.\odot([\![t_1]\!],[\![t_2]\!];\alpha)\\
[\![f\ \sigma]\!] & \coloneqq & \mu\alpha.f\ [\![\sigma,\alpha]\!] & [\![\mathbf{let}\ x = t_1\ \mathbf{in}\ t_2]\!] & \coloneqq & \mu\alpha.\langle[\![t_1]\!]\mid\tilde{\mu}x.\langle[\![t_2]\!]\mid\alpha\rangle\rangle\\
[\![K\ \sigma]\!] & \coloneqq & K\ [\![\sigma]\!] & [\![\mathbf{case}\ t\ \mathbf{of}\ \{\overline{K_i\ \Gamma_i \Rightarrow t_i}\}]\!] & \coloneqq & \mu\alpha.\langle[\![t]\!]\mid\mathbf{case}\ \{\overline{K_i\ \Gamma_i \Rightarrow \langle[\![t_i]\!]\mid\alpha\rangle}\}\rangle\\
[\![t.D\ \sigma]\!] & \coloneqq & \mu\alpha.\langle[\![t]\!]\mid D\ [\![\sigma,\alpha]\!]\rangle & [\![\mathbf{cocase}\ \{\overline{D_i\ \Gamma_i \Rightarrow t_i}\}]\!] & \coloneqq & \mathbf{cocase}\ \{\overline{D_i\ \Gamma_i,\alpha_i \Rightarrow \langle[\![t_i]\!]\mid\alpha_i\rangle}\}\\
[\![\mathbf{label}\ \alpha\ \{t\}]\!] & \coloneqq & \mu\alpha.\langle[\![t]\!]\mid\alpha\rangle & [\![\mathbf{goto}(t;\alpha)]\!] & \coloneqq & \mu\beta.\langle[\![t]\!]\mid\alpha\rangle
\end{array}$$

$$[\![\cdot]\!] : \mathit{Definition}_{\mathbf{Fun}} \to \mathit{Definition}_{\mathbf{Core}}$$

$$[\![\mathbf{def}\ f\ \Gamma : \tau \coloneqq t]\!] \coloneqq \mathbf{def}\ f\ \Gamma,\alpha :^{\mathrm{cns}} \tau \coloneqq \langle[\![t]\!]\mid\alpha\rangle$$

$$[\![\cdot]\!] : \mathit{Substitution}_{\mathbf{Fun}} \to \mathit{Substitution}_{\mathbf{Core}}$$

$$[\![\diamond]\!] \coloneqq \diamond \qquad [\![\sigma,t]\!] \coloneqq [\![\sigma]\!],[\![t]\!] \qquad [\![\sigma,\alpha]\!] \coloneqq [\![\sigma]\!],\alpha$$

In order to see why this translation is inefficient, consider the following term and its translation.

Example 4.1 (Naive Translation of Let Bindings).

$$[\![\mathbf{let}\ x = \ulcorner 2 \urcorner\ \mathbf{in}\ x * x]\!] = \mu\alpha.\langle\ulcorner 2 \urcorner \mid \tilde{\mu}x.\langle\mu\beta. * (x,x;\beta) \mid \alpha\rangle\rangle$$

Here, the statement bound by $\tilde{\mu}$ is the cut $\langle\mu\beta. * (x,x;\beta) \mid \alpha\rangle$, which after a single reduction step becomes $*(x,x;\alpha)$.

Thus, whenever we introduce a new μ-abstraction while translating a term in **Fun**, there is a potential administrative redex generated depending on the surrounding terms. With these redexes included, we have two options on how to continue compilation. Either we keep them in the expressions and finally translate them into machine code, or we add an additional simplification step before we translate **Core** to AxCut. In each case, we will incur a performance overhead which we would like to avoid. To solve this issue, we will introduce the optimized translation, which keeps track of introduced covariables and only generate new ones where necessary.

4.2 Optimized Translation

We now introduce the optimized version of the translation introduced above. This optimization works by splitting the function $[\![\cdot]\!]$ into the two functions $[\![\cdot]\!]^*$ and $[\![\cdot]\!]^*_k$. The intuition is that $[\![p]\!]^*_k$ should be the same as $\langle[\![p]\!] \mid k\rangle$, but whenever this results in a redex, for example if $[\![p]\!]$ has the form $\mu\alpha.s$, we reduce this administrative redex during the translation itself. As before, we will again assume that any covariable α in the translation of a term (except for those already present in the source term) is fresh with respect to the term being translated.

$$[\![\cdot]\!]^* : \mathit{Term} \to \mathit{Producer}$$

$$\begin{array}{rcl@{\qquad}rcl}
[\![x]\!]^* & \coloneqq & x & [\![\mathbf{cocase}\ \{\overline{D_i(\overline{x_{i,j}}) \Rightarrow t_i}\}]\!]^* & \coloneqq & \mathbf{cocase}\ \{\overline{D_i\ \Gamma_i, \alpha_i \Rightarrow [\![t_i]\!]^*_{\alpha_i}}\} \\
[\![\ulcorner n \urcorner]\!]^* & \coloneqq & \ulcorner n \urcorner & [\![\mathbf{label}\ \alpha\ \{t\}]\!]^* & \coloneqq & \mu\alpha.[\![t]\!]^*_\alpha \\
[\![K\ \sigma]\!]^* & \coloneqq & K\ [\![\sigma]\!]^* & [\![t]\!]^* & \coloneqq & \mu\alpha.[\![t]\!]^*_\alpha
\end{array}$$

$$[\![\cdot]\!]^* : \mathit{Term} \times \mathit{Consumer} \to \mathit{Statement}$$

$$\begin{array}{rcl@{\qquad}rcl}
[\![x]\!]^*_c & \coloneqq & \langle x \mid c \rangle & [\![t_1 \odot t_2]\!]^*_c & \coloneqq & \odot([\![t_1]\!]^*, [\![t_2]\!]^*; c) \\
[\![\ulcorner n \urcorner]\!]^*_c & \coloneqq & \langle \ulcorner n \urcorner \mid c \rangle & [\![\mathbf{ifz}(t_1, t_2, t_2)]\!]^*_c & \coloneqq & \mathbf{ifz}([\![t_1]\!]^*, [\![t_2]\!]^*_c, [\![t_3]\!]^*_c) \\
[\![\mathbf{let}\ x = t_1\ \mathbf{in}\ t_2]\!]^*_c & \coloneqq & \langle [\![t_1]\!]^* \mid \tilde{\mu} x.[\![t_2]\!]^*_c \rangle & [\![\mathbf{let}\ x = t_1\ \mathbf{in}\ t_2]\!]^*_c & \coloneqq & [\![t_1]\!]^*_{\tilde{\mu} x.[\![t_2]\!]^*_c} \\
\text{where} & & t_1 : T : \mathbf{codata} & [\![f\ \Gamma]\!]^*_c & \coloneqq & f\ \Gamma, c \\
[\![K\ \sigma]\!]^*_c & \coloneqq & \langle K\ [\![\sigma]\!]^* \mid c \rangle & [\![\mathbf{case}\ t\ \mathbf{of}\ \{\overline{K_i\ \Gamma_i \Rightarrow t_i}\}]\!]^*_c & \coloneqq & [\![t]\!]^*_{\mathbf{case}\ \{\overline{K_i\ \Gamma_i \Rightarrow [\![t_i]\!]^*_c}\}} \\
[\![t.D\ \sigma]\!]^*_c & \coloneqq & [\![t]\!]^*_{D\ [\![\sigma]\!]^*} & [\![\mathbf{cocase}\ \{\overline{D_i\ \Gamma_i \Rightarrow t_i}\}]\!]^*_c & \coloneqq & \langle \mathbf{cocase}\ \{\overline{D_i\ \Gamma_i, \alpha_i \Rightarrow [\![t_i]\!]^*_{\alpha_i}}\} \mid c \rangle \\
[\![\mathbf{label}\ \alpha\ \{t\}]\!]^*_c & \coloneqq & \langle \mu\alpha.[\![t]\!]^*_\alpha \mid c \rangle & [\![\mathbf{goto}(t; \alpha)]\!]^*_c & \coloneqq & [\![t]\!]^*_\alpha
\end{array}$$

$$[\![\cdot]\!] : \mathit{Definition}_{\mathbf{Fun}} \to \mathit{Definition}_{\mathbf{Core}}$$

$$[\![\mathbf{def}\ f\ \Gamma : \tau \coloneqq t]\!]^* \coloneqq \mathbf{def}\ f\ \Gamma, \alpha :^{\mathrm{cns}} \tau \coloneqq [\![t]\!]^*_\alpha$$

$$[\![\cdot]\!] : \mathit{Substitution}_{\mathbf{Fun}} \to \mathit{Substitution}_{\mathbf{Core}}$$

$$[\![\diamond]\!]^* \coloneqq \diamond \qquad [\![\sigma, t]\!]^* \coloneqq [\![\sigma]\!]^*, [\![t]\!]^* \qquad [\![\sigma, \alpha]\!]^* \coloneqq [\![\sigma]\!]^*, \alpha$$

To see how this translation removes administrative redexes, consider again the example example 4.1

Example 4.2 (Optimized Translation of Let Bindings).

$$[\![\mathbf{let}\ x = \ulcorner 2 \urcorner\ \mathbf{in}\ x * x]\!]^* = \mu\alpha.\langle \ulcorner 2 \urcorner \mid \tilde{\mu} x. * (x, x; \alpha) \rangle$$

The resulting producer in **Core** is exactly the same as the one generated from the naive translation after reducing the administrative redex.

As we will see below (theorem 4.5), this property holds in general.

4.3 Properties

Lemma 4.3. *If* $\Gamma \vdash t : \tau$ *and* $\Gamma' \vdash k :^{\mathrm{cns}} \tau$*, then* $\Gamma' \vdash [\![t]\!]^*_k$.

Proof. TODO: Figure out correct statement w.r.t. Γ and Γ'. □

Theorem 4.4 (Type Preservation). *Let* t *be a term in* ***Fun****, and let* τ *be a type. If* $\Gamma \vdash t : \tau$*, then both* $\Gamma \vdash [\![t]\!] : \tau$ *and* $\Gamma \vdash [\![t]\!]^* : \tau$.

Theorem 4.5 (Correctness). *Let* t *be a term in* ***Fun****. Then there are a finite number of reduction steps (where reduction here also means reduction under binders) such that* $\mathcal{F}([\![t]\!]) \triangleright^* \mathcal{F}([\![t]\!]^*)$*, where* $\mathcal{F}(\cdot)$ *is the focusing translation.*

5 Naming Transformation

We now start transforming terms of the language **Core** into a normal form suitable for compilation to machine code. The first step is to give names to all subterms, similar to A-normal form [**?**] or continuation-passing style [**?**]. This is a generalization of the static focusing transformation [**??**] which usually only names non-value producers of data types and non-value consumers of codata types. The transformation targets the following fragment **CoreV** of **Core**

where arguments of constructors, destructors and calls to top-level definitions as well as the producer arguments of conditionals and arithmetic operators are always (co-)variables.

*Definition 5.1 (Syntax of **CoreV**).*

$$\begin{array}{lcll} p & \coloneqq & x \mid \ulcorner n \urcorner \mid \mu\alpha.s \mid K\ \Gamma \mid \mathbf{cocase}\ \{\overline{D\ \Gamma \Rightarrow s}\} & \textit{Producers} \\ c & \coloneqq & \alpha \mid \tilde{\mu}x.s \mid D\ \Gamma \mid \mathbf{case}\ \{\overline{K\ \Gamma \Rightarrow s}\} & \textit{Consumers} \\ s & \coloneqq & \langle p \mid c \rangle \mid \mathbf{ifz}(x, s, s) \mid \odot(x, x; c) \mid f\ \Gamma \mid \mathbf{done} & \textit{Statements} \end{array}$$

This means that substitutions now always consist of (co-)variables which is indicated in the syntax by replacing σ by Γ. For the corresponding typing rules we thus obtain

$$\frac{\Theta \mid \Gamma \vdash \sigma : \Gamma' \qquad \dfrac{x : \tau \in \Gamma}{\Theta \mid \Gamma \vdash x : \tau}\ \textsc{Var}}{\Theta \mid \Gamma \vdash \sigma, x : \Gamma', x : \tau}\ \textsc{Subst}_2 \qquad\qquad \frac{\Theta \mid \Gamma \vdash \sigma : \Gamma' \qquad \dfrac{\alpha :^{\text{cns}} \tau \in \Gamma}{\Theta \mid \Gamma \vdash \alpha :^{\text{cns}} \tau}\ \textsc{Var}_2}{\Theta \mid \Gamma \vdash \sigma, \alpha : \Gamma', \alpha :^{\text{cns}} \tau}\ \textsc{Subst}_3$$

The typing judgment for substitutions hence always becomes $\Theta \mid \Gamma \vdash \Gamma' : \Gamma'$ which we can abbreviate to $\Theta \mid \Gamma \vdash \Gamma'$ with the meaning that for each $v :^{p} \tau \in \Gamma'$ we have $v :^{p} \tau \in \Gamma$.

To make the transformation uniform, compositional and to avoid administrative redexes, it is split into three mutually recursive functions $\mathcal{N}(\cdot)$, $\mathcal{B}(\cdot)[\cdot]$ and $\mathcal{M}(\cdot)[\cdot]$, where the latter two both take a continuation as an additional argument. The interesting part for $\mathcal{N}(\cdot)$ is its definition on statements for the cases where producers and consumers occur in argument positions Therefore, the cases for a cut including a constructor or destructor are special-cased. All other parts of $\mathcal{N}(\cdot)$ are simple congruences. Types do not need to be transformed. The idea is to lift all non-(co-)variable producers and consumers in argument positions by calling $\mathcal{B}(\cdot)[\cdot]$ — or $\mathcal{M}(\cdot)[\cdot]$ in the case where a whole list of arguments is to be lifted — and pass the statement itself in a continuation which binds fresh names for the arguments. The function $\mathcal{M}(\cdot)[\cdot]$ simply maps $\mathcal{B}(\cdot)[\cdot]$ over a list of arguments, but in continuation-passing style. The function $\mathcal{B}(\cdot)[\cdot]$ pattern-matches on a given producer or consumer, transforms it recursively, creates a fresh (co-)variable and uses a μ- or $\tilde{\mu}$-binding to bind the transformed term in a cut. The body of the μ- or $\tilde{\mu}$-binding consists of the given continuation applied to the freshly generated (co-)variable. In the case where the producer or consumer is a (co-)variable already, the continuation is simply applied to it. In the case where the producer or consumer is another constructor or destructor, the function $\mathcal{M}(\cdot)[\cdot]$ is applied recursively to again lift the corresponding arguments. Otherwise, the function $\mathcal{N}(\cdot)$ is recursively applied to all substatements.

$$
\begin{array}{rcl}
\mathcal{N}(\cdot) : \mathit{Statement}_{\mathbf{Core}} \to \mathit{Statement}_{\mathbf{CoreV}} & & \\
\mathcal{N}(\langle K\ \sigma \mid c\rangle) & := & \mathcal{M}(\sigma)[\lambda as.\langle K(as) \mid \mathcal{N}(c)\rangle] \\
\mathcal{N}(\langle p \mid D\ \sigma\rangle) & := & \mathcal{M}(\sigma)[\lambda as.\langle \mathcal{N}(p) \mid D(as)\rangle] \\
\mathcal{N}(\langle p \mid c\rangle) & := & \langle \mathcal{N}(p) \mid \mathcal{N}(c)\rangle \\
\mathcal{N}(\odot(p_1, p_2; c)) & := & \mathcal{B}(p_1)[\lambda a_1.\mathcal{B}(p_2)[\lambda a_2.\ \odot\ (a_1, a_2; c)]] \\
\mathcal{N}(\mathbf{ifz}(p, s_1, s_2)) & := & \mathcal{B}(p)[\lambda a.\mathbf{ifz}(a, \mathcal{N}(s_1), \mathcal{N}(s_2))] \\
\mathcal{N}(f\ \sigma) & := & \mathcal{M}(\sigma)[\lambda as.f(as)] \\
\mathcal{N}(\mathbf{done}) & := & \mathbf{done}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{N}(\cdot) : \mathit{Definition}_{\mathbf{Core}} \to \mathit{Definition}_{\mathbf{CoreV}} & & \\
\mathcal{N}(\mathbf{def}\ f\ \Gamma := s) & := & \mathbf{def}\ f\ \Gamma := \mathcal{N}(s)
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{N}(\cdot) : \mathit{Producer}_{\mathbf{Core}} \to \mathit{Producer}_{\mathbf{CoreV}} & & \\
\mathcal{N}(x) & := & x \\
\mathcal{N}(\ulcorner n \urcorner) & := & \ulcorner n \urcorner \\
\mathcal{N}(\mu\alpha.s) & := & \mu\alpha.\mathcal{N}(s) \\
\mathcal{N}(K\ \sigma) & := & \text{does not occur} \\
\mathcal{N}(\mathbf{cocase}\ \{\overline{D_i\ \Gamma_i \Rightarrow s_i}\}) & := & \mathbf{cocase}\ \{\overline{D_i\ \Gamma_i \Rightarrow \mathcal{N}(s_i)}\}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{N}(\cdot) : \mathit{Consumer}_{\mathbf{Core}} \to \mathit{Consumer}_{\mathbf{CoreV}} & & \\
\mathcal{N}(\alpha) & := & \alpha \\
\mathcal{N}(\tilde{\mu}x.s) & := & \tilde{\mu}x.\mathcal{N}(s) \\
\mathcal{N}(D\ \sigma) & := & \text{does not occur} \\
\mathcal{N}(\mathbf{case}\ \{\overline{K_i\ \Gamma_i \Rightarrow s_i}\}) & := & \mathbf{case}\ \{\overline{K_i\ \Gamma_i \Rightarrow \mathcal{N}(s_i)}\}
\end{array}
$$

$$
\begin{array}{rcll}
\mathcal{B}(\cdot)[\cdot] : \mathit{Producer}_{\mathbf{Core}} \times (\mathit{Name} \to \mathit{Statement}_{\mathbf{CoreV}}) \to \mathit{Statement}_{\mathbf{CoreV}} & & & \\
\mathcal{B}(x)[k] & := & k(x) & \\
\mathcal{B}(\ulcorner n \urcorner)[k] & := & \langle \ulcorner n \urcorner \mid \tilde{\mu}x.k(x)\rangle & (x \text{ fresh}) \\
\mathcal{B}(\mu\alpha.s)[k] & := & \langle \mu\alpha.\mathcal{N}(s) \mid \tilde{\mu}x.k(x)\rangle & (x \text{ fresh}) \\
\mathcal{B}(K\ \sigma)[k] & := & \mathcal{M}(\sigma)[\lambda as.\langle K(as) \mid \tilde{\mu}x.k(x)\rangle] & (x \text{ fresh}) \\
\mathcal{B}(\mathbf{cocase}\ \{\overline{D_i\ \Gamma_i \Rightarrow s_i}\})[k] & := & \langle \mathbf{cocase}\ \{\overline{D_i\ \Gamma_i \Rightarrow \mathcal{N}(s_i)}\} \mid \tilde{\mu}x.k(x)\rangle & (x \text{ fresh})
\end{array}
$$

$$
\begin{array}{rcll}
\mathcal{B}(\cdot)[\cdot] : \mathit{Consumer}_{\mathbf{Core}} \times (\mathit{Name} \to \mathit{Statement}_{\mathbf{CoreV}}) \to \mathit{Statement}_{\mathbf{CoreV}} & & & \\
\mathcal{B}(\alpha)[k] & := & k(\alpha) & \\
\mathcal{B}(\tilde{\mu}x.s)[k] & := & \langle \mu\alpha.k(\alpha) \mid \tilde{\mu}x.\mathcal{N}(s)\rangle & (\alpha \text{ fresh}) \\
\mathcal{B}(D\ \sigma)[k] & := & \mathcal{M}(\sigma)[\lambda as.\langle \mu\alpha.k(\alpha) \mid D(as)\rangle] & (\alpha \text{ fresh}) \\
\mathcal{B}(\mathbf{case}\ \{\overline{K_i\ \Gamma_i \Rightarrow s_i}\})[k] & := & \langle \mu\alpha.k(\alpha) \mid \mathbf{case}\ \{\overline{K_i\ \Gamma_i \Rightarrow \mathcal{N}(s_i)}\}\rangle & (\alpha \text{ fresh})
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{M}(\cdot)[\cdot] : \mathit{Substitution} \times (\mathit{List}\ \mathit{Name} \to \mathit{Statement}_{\mathbf{CoreV}}) \to \mathit{Statement}_{\mathbf{CoreV}} & & \\
\mathcal{M}(\diamond)[k] & := & k(\diamond) \\
\mathcal{M}(e :: \sigma)[k] & := & \mathcal{B}(e)[\lambda a.\mathcal{M}(\sigma)[\lambda as.k(a :: as)]]
\end{array}
$$

6 Redundancy Elimination

The next step towards our normal consists of removing redundant part of the language. More precisely, note that the fragment **CoreV** explained in the previous section could as well be described with only statements, by inlining the producers and consumers into the cut statement and the consumers into the arithmetic operators. This is because all producer and consumer arguments are (co-)variables in this fragment. For arithmetic operators, typing only allows the consumer to be a covariable or a $\tilde{\mu}$-binding. For cuts, we could in principle have 20 different forms of cuts since

we have five different kinds of producers and four different kinds of consumers. However, typing again precludes four of them: a constructor and a destructor as well as a pattern match and a copattern match can never meet in a cut, and moreover, an integer literal can never be cut with a destructor or a pattern match. Together with conditionals, calls of top-level definitions and the terminating statement, this results in 21 statement forms in total.

We will now transform away eight of them, resulting in the following fragment **CoreR** of **Core**.

*Definition 6.1 (Syntax of **CoreR**).*

$$
\begin{array}{rcll}
s & \coloneqq & \langle K\ \Gamma \mid \tilde{\mu}x.s\rangle \mid \langle x \mid \mathbf{case}\ \{\overline{K\ \Gamma \Rightarrow s}\}\rangle \mid \langle \mathbf{cocase}\ \{\overline{D\ \Gamma \Rightarrow s}\} \mid \tilde{\mu}x.s\rangle \mid \langle x \mid D\ \Gamma\rangle & \textit{Statements} \\
& \mid & \langle \mu\alpha.s \mid D\ \Gamma\rangle \mid \langle \mathbf{cocase}\ \{\overline{D\ \Gamma \Rightarrow s}\} \mid \alpha\rangle \mid \langle \mu\alpha.s \mid \mathbf{case}\ \{\overline{K\ \Gamma \Rightarrow s}\}\rangle \mid \langle K\ \Gamma \mid \alpha\rangle & \\
& \mid & \langle \ulcorner n \urcorner \mid \tilde{\mu}x.s\rangle \mid \mathbf{ifz}(x, s, s) \mid \odot(x, x; \alpha) \mid f\ \Gamma \mid \mathbf{done} &
\end{array}
$$

We consider the interesting cases in turn. Let us start with the cases where a variable or covariable meets a $\tilde{\mu}$- or μ-binding in a cut. This is a simple renaming which we can transform away by reducing the cut via substitution. Next, consider the cases of completely known cuts, i.e., cuts where one side is a constructor or destructor and the other side is a pattern match or copattern match. These cuts can also simply be reduced. As the arguments of constructors and destructors are all (co-)variables, this just amounts to picking the corresponding branch and renaming the bound (co-)variables. Hence, there is no risk of non-termination. Now, consider the case where a μ- and a $\tilde{\mu}$-binding are cut, the famous critial pairs [**?**]. To transform those away, we distinguish between the different kinds of types. For data and codata types, we have ensured that all η-laws hold by using call-by-value evaluation for data types and call-by-name evaluation for codata types. Therefore, we can η-expand the $\tilde{\mu}$-binding in critical pairs of data types and the μ-binding in critical pairs of data types. For the primitive type **Int**, we cannot use η-expansion, so we need another way to resolve critical pairs. The $\tilde{\mu}$-binding can be viewed as a continuation for integers. As we use call-by-value evaluation for integers, we should find a different possibility to model this continuation, in such a way that the μ-binding would be reduced first. To do so, we can define a new data type:

$$\mathbf{data}\ \texttt{Cont}\ \{\texttt{Ret}(\texttt{x} : \mathbf{Int})\}$$

A pattern match of this data type can also be viewed as a continuation for integers, the only difference from the $\tilde{\mu}$-binding being that the integer is wrapped into the constructor `Ret`. This means that in all places where an integer is cut with a covariable, which now stands for a consumer of type `Cont` instead of a $\tilde{\mu}$-binding of type **Int**, we have to wrap the integer into a `Ret`. There are three cases where this happens: in a cut of a variable and a covariable of type **Int**, in the cut of an integer literal with a covariable and in the case where the consumer of an arithmetic operator is a covariable. For the latter two cases, we further insert a $\tilde{\mu}$-binding to give the integer literal or the result of the arithmetic operator a name before wrapping this name into a `Ret` which is then cut with the covariable. It will become clear later, that no integer is ever actually wrapped into a constructor in the sense of storing it on the heap. This is because those wrapped integers only appear in cuts with covariables in which the role of the constructor is to pick a branch in the pattern match. As there is always only one branch, there is no overhead in modelling continuations for integers this way. Finally, we transform away completely unknown cuts, that is, cuts of a variable with a covariable. We have already seen above how to do this for primitive integers. For data types and codata types we can again use η-expansion as in the cases of critical pairs, i.e., we η-expand the covariable in cuts at data types and we η-expand the variable in cuts at codata types. All other cases are simple congruences.

$$\mathcal{R}(\cdot) : \mathit{Definition}_{\mathbf{CoreV}} \to \mathit{Definition}_{\mathbf{CoreR}}$$
$$\mathcal{R}(\mathbf{def}\ f\ \Gamma \coloneqq s) \quad \coloneqq \quad \mathbf{def}\ f\ \Gamma \coloneqq \mathcal{R}(s)$$

$$\mathcal{R}(\cdot) : \mathit{Statement}_{\mathbf{CoreV}} \to \mathit{Statement}_{\mathbf{CoreR}}$$

$$\begin{array}{rcl@{\qquad}rcl}
\mathcal{R}(\langle \mu\alpha.s \mid \beta \rangle) & \coloneqq & \mathcal{R}(s[\alpha \mapsto \beta]) & \mathcal{R}(\langle K_j\ \Gamma_0 \mid \mathbf{case}\ \{\overline{K_i\ \Gamma_i \Rightarrow s_i}\} \rangle) & \coloneqq & \mathcal{R}(s_j[\Gamma_j \mapsto \Gamma_0]) \\
\mathcal{R}(\langle y \mid \tilde{\mu}x.s \rangle) & \coloneqq & \mathcal{R}(s[x \mapsto y]) & \mathcal{R}(\langle \mathbf{cocase}\ \{\overline{D_i\ \Gamma_i \Rightarrow s_i}\} \mid D_j\ \Gamma_0 \rangle) & \coloneqq & \mathcal{R}(s_j[\Gamma_j \mapsto \Gamma_0])
\end{array}$$

$$\begin{array}{rcl}
\mathcal{R}(\langle \mu\alpha.s_1 \mid \tilde{\mu}x.s_2 \rangle_T) & \coloneqq & \langle \mu\alpha.\mathcal{R}(s_1) \mid \mathbf{case}\ \{\overline{K_i\ \Gamma_i \Rightarrow \langle K_i\ \Gamma_i \mid \tilde{\mu}x.\mathcal{R}(s_2) \rangle}\} \rangle \\
\text{where} & & \mathbf{data}\ T\ \{\overline{K_i\ \Gamma_i}\} \in \Theta \quad (\overline{\Gamma_i}\ \text{fresh}) \\
\mathcal{R}(\langle \mu\alpha.s_1 \mid \tilde{\mu}x.s_2 \rangle_T) & \coloneqq & \langle \mathbf{cocase}\ \{\overline{D_i\ \Gamma_i \Rightarrow \langle \mu\alpha.\mathcal{R}(s_1) \mid D_i\ \Gamma_i \rangle}\} \mid \tilde{\mu}x.s_2 \rangle \\
\text{where} & & \mathbf{codata}\ T\ \{\overline{D_i\ \Gamma_i}\} \in \Theta \quad (\overline{\Gamma_i}\ \text{fresh}) \\
\mathcal{R}(\langle \mu\alpha.s_1 \mid \tilde{\mu}x.s_2 \rangle_{\mathbf{Int}}) & \coloneqq & \langle \mu\alpha.s_1 \mid \mathbf{case}\ \{\mathtt{Ret}(x) \Rightarrow \mathcal{R}(s_2)\} \rangle \\
\text{where} & & \mathbf{data}\ \mathtt{Cont}\ \{\mathtt{Ret}(\mathtt{x} : \mathbf{Int})\} \in \Theta \\
\mathcal{R}(\langle x \mid \alpha \rangle_T) & \coloneqq & \langle x \mid \mathbf{case}\ \{\overline{K_i\ \Gamma_i \Rightarrow \langle K_i\ \Gamma_i \mid \alpha \rangle}\} \rangle \\
\text{where} & & \mathbf{data}\ T\ \{\overline{K_i\ \Gamma_i}\} \in \Theta \quad (\overline{\Gamma_i}\ \text{fresh}) \\
\mathcal{R}(\langle x \mid \alpha \rangle_T) & \coloneqq & \langle \mathbf{cocase}\ \{\overline{D_i\ \Gamma_i \Rightarrow \langle x \mid D_i\ \Gamma_i \rangle}\} \mid \alpha \rangle \\
\text{where} & & \mathbf{codata}\ T\ \{\overline{D_i\ \Gamma_i}\} \in \Theta \quad (\overline{\Gamma_i}\ \text{fresh}) \\
\mathcal{R}(\langle x \mid \alpha \rangle_{\mathbf{Int}}) & \coloneqq & \langle \mathtt{Ret}(x) \mid \alpha \rangle
\end{array}$$

$$\begin{array}{rcl@{\qquad}rcl}
\mathcal{R}(\langle \ulcorner n \urcorner \mid \alpha \rangle) & \coloneqq & \langle \ulcorner n \urcorner \mid \tilde{\mu}x.\langle \mathtt{Ret}(x) \mid \alpha \rangle \rangle & \mathcal{R}(\odot(x_1, x_2; \alpha)) & \coloneqq & \odot(x_1, x_2; \tilde{\mu}x.\langle \mathtt{Ret}(x) \mid \alpha \rangle) \\
\text{where} & & (x\ \text{fresh}) & \text{where} & & (x\ \text{fresh}) \\
\mathcal{R}(\langle \ulcorner n \urcorner \mid \tilde{\mu}x.s \rangle) & \coloneqq & \langle \ulcorner n \urcorner \mid \tilde{\mu}x.\mathcal{R}(s) \rangle & \mathcal{R}(\odot(x_1, x_2; \tilde{\mu}x.s)) & \coloneqq & \odot(x_1, x_2; \tilde{\mu}x.\mathcal{R}(s)) \\
\mathcal{R}(\langle K\ \Gamma_0 \mid \tilde{\mu}x.s \rangle) & \coloneqq & \langle K\ \Gamma_0 \mid \tilde{\mu}x.\mathcal{R}(s) \rangle & \mathcal{R}(\langle x \mid \mathbf{case}\ \{\overline{K_i\ \Gamma_i \Rightarrow s_i}\} \rangle) & \coloneqq & \langle x \mid \mathbf{case}\ \{\overline{K_i\ \Gamma_i \Rightarrow \mathcal{R}(s_i)}\} \rangle \\
\mathcal{R}(\langle \mu\alpha.s \mid D\ \Gamma_0 \rangle) & \coloneqq & \langle \mu\alpha.\mathcal{R}(s) \mid D\ \Gamma_0 \rangle & \mathcal{R}(\langle \mathbf{cocase}\ \{\overline{D_i\ \Gamma_i \Rightarrow s_i}\} \mid \alpha \rangle) & \coloneqq & \langle \mathbf{cocase}\ \{\overline{D_i\ \Gamma_i \Rightarrow \mathcal{R}(s_i)}\} \mid \alpha \rangle \\
\mathcal{R}(\langle x \mid D\ \Gamma_0 \rangle) & \coloneqq & \langle x \mid D\ \Gamma_0 \rangle & \mathcal{R}(\langle \mathbf{cocase}\ \{\overline{D_i\ \Gamma_i \Rightarrow s_i}\} \mid \tilde{\mu}x.s \rangle) & \coloneqq & \langle \mathbf{cocase}\ \{\overline{D_i\ \Gamma_i \Rightarrow \mathcal{R}(s_i)}\} \mid \tilde{\mu}x.\mathcal{R}(s) \rangle \\
\mathcal{R}(\langle K\ \Gamma_0 \mid \alpha \rangle) & \coloneqq & \langle K\ \Gamma_0 \mid \alpha \rangle & \mathcal{R}(\langle \mu\alpha.s \mid \mathbf{case}\ \{\overline{K_i\ \Gamma_i \Rightarrow s_i}\} \rangle) & \coloneqq & \langle \mu\alpha.\mathcal{R}(s) \mid \mathbf{case}\ \{\overline{K_i\ \Gamma_i \Rightarrow \mathcal{R}(s_i)}\} \rangle \\
\mathcal{R}(f\ \Gamma_0) & \coloneqq & f\ \Gamma_0 & \mathcal{R}(\mathbf{ifz}(x, s_1, s_2)) & \coloneqq & \mathbf{ifz}(x, \mathcal{R}(s_1), \mathcal{R}(s_2)) \\
\mathcal{R}(\mathbf{done}) & \coloneqq & \mathbf{done} & & &
\end{array}$$

7 Substructurality Anaysis

This is the translation from **CoreR** to **CoreS**. It inserts explicit subtitutions. The subtitutions aim to drop unneeded variables as early as possible and duplicate variables as late as possible to reduce the register pressure. (I'm not perfectly sure whether the insertions are fully correct (i.p., with respect to shadowing) and there may be a better solution.)

$$\mathcal{S}(\cdot) : \mathit{Definition}_{\mathbf{CoreR}} \to \mathit{Definition}_{\mathbf{CoreS}}$$

$$\begin{array}{rcl}
\mathcal{S}(\mathbf{def}\ f\ \Gamma \coloneqq s) & \coloneqq & \mathbf{def}\ f\ \Gamma \coloneqq [\Gamma' \mapsto \Gamma']; \mathcal{S}(s)_{\Gamma'} \\
\text{where} & & \Gamma' = \Gamma \cap \mathsf{freeVars}(s)
\end{array}$$

$$\mathcal{S}(\cdot). : \mathit{Statement}_{\mathbf{CoreR}} \times \mathit{Context}_{\mathbf{CoreS}} \to \mathit{Statement}_{\mathbf{CoreS}}$$

$$\begin{array}{rcl}
\mathcal{S}(\langle K\ \Gamma_0 \mid \tilde{\mu}x.s\rangle)_\Gamma & \coloneqq & [\Gamma' \mapsto \Gamma', \Gamma_0 \mapsto \Gamma_0]; \langle K\ \Gamma_0 \mid \tilde{\mu}x.\mathcal{S}(s)_{\Gamma',x}\rangle \\
\text{where} & & \Gamma' = (\Gamma - x) \cap \mathsf{freeVars}(s) \text{ (subtraction only needed if variables not unique)} \\
\mathcal{S}(\langle \mu\alpha.s \mid D\ \Gamma_0\rangle)_\Gamma & \coloneqq & [\Gamma' \mapsto \Gamma', \Gamma_0 \mapsto \Gamma_0]; \langle \mu\alpha.\mathcal{S}(s)_{\Gamma',\alpha} \mid D\ \Gamma_0\rangle \\
\text{where} & & \Gamma' = (\Gamma - \alpha) \cap \mathsf{freeVars}(s) \\
\mathcal{S}(\langle K\ \Gamma_0 \mid \alpha\rangle)_\Gamma & \coloneqq & [\Gamma_0 \mapsto \Gamma_0, \alpha \mapsto \alpha]; \langle K\ \Gamma_0 \mid \alpha\rangle \\
\mathcal{S}(\langle x \mid D\ \Gamma_0\rangle)_\Gamma & \coloneqq & [\Gamma_0 \mapsto \Gamma_0, x \mapsto x]; \langle x \mid D\ \Gamma_0\rangle \\
\mathcal{S}(\langle x \mid \mathbf{case}\ \{\overline{K_i\ \Gamma_i \Rightarrow s_i}\}\rangle)_\Gamma & \coloneqq & [\Gamma' \mapsto \Gamma', x \mapsto x]; \langle x \mid \mathbf{case}\ \{\overline{K_i\ \Gamma_i \Rightarrow \mathcal{S}(s_i)_{\Gamma',\Gamma_i}}\}\rangle \\
\text{where} & & \Gamma' = \Gamma \cap \bigcup_i \mathsf{freeVars}(s_i) \\
\mathcal{S}(\langle \mathbf{cocase}\ \{\overline{D_i\ \Gamma_i \Rightarrow s_i}\} \mid \alpha\rangle)_\Gamma & \coloneqq & [\Gamma' \mapsto \Gamma', \alpha \mapsto \alpha]; \langle \mathbf{cocase}\ \{\overline{D_i\ \Gamma_i \Rightarrow \mathcal{S}(s_i)_{\Gamma',\Gamma_i}}\} \mid \alpha\rangle \\
\text{where} & & \Gamma' = \Gamma \cap \bigcup_i \mathsf{freeVars}(s_i) \\
\mathcal{S}(\langle \mu\alpha.s \mid \mathbf{case}\ \{\overline{K_i\ \Gamma_i \Rightarrow s_i}\}\rangle)_\Gamma & \coloneqq & [\Gamma' \mapsto \Gamma', \Gamma_0 \mapsto \Gamma_0]; \langle \mu\alpha.\mathcal{S}(s)_{\Gamma',\alpha} \mid \mathbf{case}\ \{\overline{K_i\ \Gamma_i \Rightarrow \mathcal{S}(s_i)_{\Gamma_i,\Gamma_0}}\}_{\Gamma_0}\rangle \\
\text{where} & & \Gamma_0 = \Gamma \cap \bigcup_i \mathsf{freeVars}(s_i) \quad \Gamma' = (\Gamma - \alpha) \cap \mathsf{freeVars}(s) \\
\mathcal{S}(\langle \mathbf{cocase}\ \{\overline{D_i\ \Gamma_i \Rightarrow s_i}\} \mid \tilde{\mu}x.s\rangle)_\Gamma & \coloneqq & [\Gamma' \mapsto \Gamma', \Gamma_0 \mapsto \Gamma_0]; \langle \mathbf{cocase}\ \{\overline{D_i\ \Gamma_i \Rightarrow \mathcal{S}(s_i)_{\Gamma_i,\Gamma_0}}\}_{\Gamma_0} \mid \tilde{\mu}x.\mathcal{S}(s)_{\Gamma',x}\rangle \\
\text{where} & & \Gamma_0 = \Gamma \cap \bigcup_i \mathsf{freeVars}(s_i) \quad \Gamma' = (\Gamma - x) \cap \mathsf{freeVars}(s) \\
\mathcal{S}(\langle \ulcorner n \urcorner \mid \tilde{\mu}x.s\rangle)_\Gamma & \coloneqq & [\Gamma' \mapsto \Gamma']; \langle \ulcorner n \urcorner \mid \tilde{\mu}x.\mathcal{S}(s)_{\Gamma',x}\rangle \\
\text{where} & & \Gamma' = (\Gamma - x) \cap \mathsf{freeVars}(s) \\
\mathcal{S}(\odot(x_1, x_2; \tilde{\mu}x.s))_\Gamma & \coloneqq & [\Gamma' \mapsto \Gamma']; \odot(x_1, x_2; \tilde{\mu}x.\mathcal{S}(s)_{\Gamma',x}) \\
\text{where} & & \Gamma' = (\Gamma - x) \cap (x_1 \cup x_2 \cup \mathsf{freeVars}(s)) \\
\mathcal{S}(\mathbf{ifz}(x, s_1, s_2))_\Gamma & \coloneqq & \mathbf{ifz}(x, \mathcal{S}(s_1)_\Gamma, \mathcal{S}(s_2)_\Gamma) \\
\mathcal{S}(f\ \Gamma_0)_\Gamma & \coloneqq & [\Gamma_0 \mapsto \Gamma_0]; f\ \Gamma_0 \\
\mathcal{S}(\mathbf{done})_\Gamma & \coloneqq & \mathbf{done}
\end{array}$$

8 The Language AxCut

Definition 8.1 (Syntax of AxCut).

$$\begin{array}{rcll}
s & \coloneqq & \mathbf{done} \mid \mathbf{jump}\, f \mid [\Gamma \mapsto \Gamma]; s \mid \mathbf{lit}[\ulcorner n \urcorner]\{v \Rightarrow s\} \mid \odot(v, v)\{v \Rightarrow s\} \mid \mathbf{ifz}(v)\{() \Rightarrow s, () \Rightarrow s\} & \textit{Statements} \\
 & \mid & \mathbf{let}\, v = m\ \Gamma; s \mid \mathbf{switch}\, v\, b \mid \mathbf{new}\, v = \Gamma\ b; s \mid \mathbf{invoke}\, v\, m & \\
b & \coloneqq & \{\overline{m\ \Gamma}\} & \textit{Branches} \\
v & \coloneqq & x \mid \alpha \mid j \mid \ldots & \textit{Variables} \\
\tau & \coloneqq & \mathbf{Int} \mid T & \textit{Types} \\
\kappa & \coloneqq & \mathbf{sign} \mid \mathbf{prim} & \textit{Kinds} \\
\Gamma & \coloneqq & \emptyset \mid \Gamma, v : \tau \mid \Gamma, v :^{\mathrm{cns}} \tau \mid v :^{\mathrm{ext}} \tau & \textit{Typing Contexts} \\
\delta & \coloneqq & \mathbf{signature}\, T\, \{\overline{m\ \Gamma \Rightarrow s}\} \mid \mathbf{def}\ f : \Gamma \coloneqq s & \textit{Declarations} \\
\Theta & \coloneqq & \overline{\delta} & \textit{Programs}
\end{array}$$

8.1 Typing Rules

9 Translation to AxCut

This is the translation from **CoreS** to **AxCut**. It collapses data and codata types and renames constructs to a unified syntax.

$$
\begin{array}{rcll}
\multicolumn{4}{c}{C(\cdot) : \mathit{Type}_{\mathbf{CoreS}} \to \mathit{Type}_{\mathbf{AxCut}}} \\
C(\mathbf{data}\ T\ \{\overline{K_i\ \Gamma_i}\}) & := & \mathbf{signature}\ T\ \{\overline{K_i\ C(\Gamma_i)}\} & \\
C(\mathbf{codata}\ T\ \{\overline{D_i\ \Gamma_i}\}) & := & \mathbf{signature}\ T\ \{\overline{D_i\ C(\Gamma_i)}\} & \\
C(\Gamma, v : T) & := & C(\Gamma), v\ C(: T) & \\
C(: T) & := & : T & \text{where } T : \mathbf{data} \\
C(:^{\mathrm{cns}} T) & := & :^{\mathrm{cns}} T & \text{where } T : \mathbf{data} \\
C(: T) & := & :^{\mathrm{cns}} T & \text{where } T : \mathbf{codata} \\
C(:^{\mathrm{cns}} T) & := & : T & \text{where } T : \mathbf{codata} \\
C(: \mathbf{Int}) & := & :^{\mathrm{ext}} \mathbf{Int} &
\end{array}
$$

$$
\begin{array}{rcl}
\multicolumn{3}{c}{C(\cdot) : \mathit{Definition}_{\mathbf{CoreS}} \to \mathit{Definition}_{\mathbf{AxCut}}} \\
C(\mathbf{def}\ f\ \Gamma := s) & := & \mathbf{def}\ f : \Gamma := C(s)
\end{array}
$$

$$
\begin{array}{rcl}
\multicolumn{3}{c}{C(\cdot) : \mathit{Statement}_{\mathbf{CoreS}} \to \mathit{Statement}_{\mathbf{AxCut}}} \\
C(\langle K\ \Gamma_0 \mid \tilde{\mu} x.s \rangle) & := & \mathbf{let}\ x = K\ \Gamma_0; C(s) \\
C(\langle \mu\alpha.s \mid D\ \Gamma_0 \rangle) & := & \mathbf{let}\ \alpha = D\ \Gamma_0; C(s) \\
C(\langle K\ \Gamma_0 \mid \alpha \rangle) & := & \mathbf{invoke}\ \alpha\ K \\
C(\langle x \mid D\ \Gamma_0 \rangle) & := & \mathbf{invoke}\ x\ D \\
C(\langle x \mid \mathbf{case}\ \{\overline{K_i\ \Gamma_i \Rightarrow s_i}\} \rangle) & := & \mathbf{switch}\ x\ \{\overline{K_i\ \Gamma_i \Rightarrow C(s_i)}\} \\
C(\langle \mathbf{cocase}\ \{\overline{D_i\ \Gamma_i \Rightarrow s_i}\} \mid \alpha \rangle) & := & \mathbf{switch}\ \alpha\ \{\overline{D_i\ \Gamma_i \Rightarrow C(s_i)}\} \\
C(\langle \mu\alpha.s \mid \mathbf{case}\ \{\overline{K_i\ \Gamma_i \Rightarrow s_i}\}_{\Gamma_0} \rangle) & := & \mathbf{new}\ \alpha = (\Gamma_0)\{\overline{K_i\ \Gamma_i \Rightarrow C(s_i)}\}; C(s) \\
C(\langle \mathbf{cocase}\ \{\overline{D_i\ \Gamma_i \Rightarrow s_i}\}_{\Gamma_0} \mid \tilde{\mu} x.s \rangle) & := & \mathbf{new}\ x = (\Gamma_0)\{\overline{D_i\ \Gamma_i \Rightarrow C(s_i)}\}; C(s) \\
C(\langle \ulcorner n \urcorner \mid \tilde{\mu} x.s \rangle) & := & \mathbf{lit}[\ulcorner n \urcorner]\{x \Rightarrow C(s)\} \\
C(\odot(x_1, x_2; \tilde{\mu} x.s)) & := & \odot(x_1, x_2)\{x \Rightarrow C(s)\} \\
C(\mathbf{ifz}(x, s_1, s_2)) & := & \mathbf{ifz}(x)\{() \Rightarrow C(s_1), () \Rightarrow C(s_2)\} \\
C([\Gamma \mapsto \Gamma']; s) & := & \mathbf{substitute}[\Gamma \mapsto \Gamma']; C(s) \\
C(f\ \Gamma_0) & := & \mathbf{jump}\ f \\
C(\mathbf{done}) & := & \mathbf{done}
\end{array}
$$

10 Related Work

- [?]

11 Conclusion

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

$$\frac{\Theta \mid \Gamma, v :^{\text{ext}} \mathbf{Int} \vdash s}{\Theta \mid \Gamma \vdash \mathbf{lit}[\ulcorner n \urcorner]\{v \Rightarrow s\}}\ \textsc{Op} \qquad \frac{v_1 :^{\text{ext}} \mathbf{Int} \in \Gamma \qquad v_2 :^{\text{ext}} \mathbf{Int} \in \Gamma \qquad \Theta \mid \Gamma, v :^{\text{ext}} \mathbf{Int} \vdash s}{\Theta \mid \Gamma \vdash \odot(v_1, v_2)\{v \Rightarrow s\}}\ \textsc{Op}$$

$$\frac{v :^{\text{ext}} \mathbf{Int} \in \Gamma \qquad \Theta \mid \Gamma \vdash s_1 \qquad \Theta \mid \Gamma \vdash s_2}{\Theta \mid \Gamma \vdash \mathbf{ifz}(v)\{() \Rightarrow s_1, () \Rightarrow s_2\}}\ \textsc{Ifz}$$

$$\frac{\mathbf{signature}\ T\ \{\ldots, m\ \Gamma_0, \ldots\} \in \Theta \qquad \Theta \mid \Gamma, v : T \vdash s}{\Theta \mid \Gamma, \Gamma_0 \vdash \mathbf{let}\, v = m\ \Gamma_0; s}\ \textsc{Let}$$

$$\frac{\mathbf{signature}\ T\ \{\overline{m_i\ \Gamma_i}\} \in \Theta \qquad \overline{\Theta \mid \Gamma_i, \Gamma_0 \vdash s_i} \qquad \Theta \mid \Gamma, v :^{\text{cns}} T \vdash s}{\Theta \mid \Gamma, \Gamma_0 \vdash \mathbf{new}\, v = (\Gamma_0)\{\overline{m_i\ \Gamma_i \Rightarrow s_i}\}; s}\ \textsc{New}$$

$$\frac{\mathbf{signature}\ T\ \{\overline{m_i\ \Gamma_i}\} \in \Theta \qquad \overline{\Theta \mid \Gamma, \Gamma_i \vdash s_i}}{\Theta \mid \Gamma, v : T \vdash \mathbf{switch}\, v\, \{\overline{m_i \Gamma_i}\}}\ \textsc{Switch} \qquad \frac{\mathbf{signature}\ T\ \{\ldots, m\ \Gamma, \ldots\} \in \Theta}{\Theta \mid \Gamma, v :^{\text{cns}} T \vdash \mathbf{invoke}\, v\, m}\ \textsc{Invoke}$$

$$\frac{}{\Theta \mid \Gamma \vdash \mathbf{done}}\ \textsc{Done} \qquad \frac{\Theta \mid v_1' :^{p} \tau, \ldots \vdash s \qquad v_1 :^{p} \tau \in \Gamma \ldots}{\Theta \mid \Gamma \vdash [v_1' \mapsto v_1, \ldots]; s}\ \textsc{Substitute} \qquad \frac{\mathbf{def}\ f\ \Gamma \in \Theta}{\Theta \mid \Gamma \vdash \mathbf{jump}\, f}\ \textsc{Jump}$$

Kinding Types $\Theta \vdash \tau : \kappa$

$$\frac{}{\Theta \vdash \mathbf{Int} : \mathbf{prim}}\ \textsc{PrimKind} \qquad \frac{\mathbf{signature}\ T\ \{\ldots\} \in \Theta}{\Theta \vdash T : \mathbf{sign}}\ \textsc{SignatureKind}$$

Well-formed Programs $\vdash \Theta$ Ok

$$\frac{}{\vdash \Diamond\ \textsc{Ok}}\ \textsc{Wf-Empty} \qquad \frac{\vdash \Theta\ \textsc{Ok} \qquad \overline{\Theta, \mathbf{signature}\ T\ \{\ldots\} \vdash \Gamma_i\ \textsc{Ctx}} \qquad \overline{\Gamma_i \vdash \tau_i : \kappa}}{\vdash \Theta, \mathbf{signature}\ T\ \{\overline{m_i\ \Gamma_i}\}\ \textsc{Ok}}\ \textsc{Wf-Signature}$$

$$\frac{\vdash \Theta\ \textsc{Ok} \qquad \Theta, \mathbf{def}\ f : \Gamma := s \mid \Gamma \vdash s}{\vdash \Theta, \mathbf{def}\ f : \Gamma := s\ \textsc{Ok}}\ \textsc{Wf-Def}$$

Well-formed Contexts $\Theta \vdash \Gamma$ Ctx

$$\frac{}{\Theta \vdash \Diamond\ \textsc{Ctx}}\ \textsc{Ctx}_1 \qquad \frac{\Theta \vdash \Gamma\ \textsc{Ctx} \quad v \notin \Gamma \quad \Theta \vdash \tau : \kappa}{\Theta \vdash \Gamma, v : \tau\ \textsc{Ctx}}\ \textsc{Ctx}_2$$

$$\frac{\Theta \vdash \Gamma\ \textsc{Ctx} \quad v \notin \Gamma \quad \Theta \vdash \tau : \kappa}{\Theta \vdash \Gamma, v :^{\text{cns}} \tau\ \textsc{Ctx}}\ \textsc{Ctx}_3 \qquad \frac{\Theta \vdash \Gamma\ \textsc{Ctx} \quad v \notin \Gamma \quad \Theta \vdash \tau : \kappa}{\Theta \vdash \Gamma, v :^{\text{ext}} \tau\ \textsc{Ctx}}\ \textsc{Ctx}_4$$

Fig. 3. Typing rules for **AxCut**