Sequent Calculus as a Compiler Intermediate Language

Paul Downen Luke Maurer
Zena M. Ariola
University of Oregon, USA
{pdownen,maurerl,ariola}@cs.uoregon.edu

Simon Peyton Jones
Microsoft Research Cambridge, UK
simonpj@microsoft.com

Abstract

The λ-calculus is popular as an intermediate language for practical compilers. But in the world of logic it has a lesser-known twin, born at the same time, called the *sequent calculus*. Perhaps that would make for a good intermediate language, too? To explore this question we designed Sequent Core, a practically-oriented core calculus based on the sequent calculus, and used it to re-implement a substantial chunk of the Glasgow Haskell Compiler.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers

Keywords Intermediate representations; Natural deduction; Sequent calculus; Compiler optimizations; Continuations; Haskell

1. Introduction

Steele and Sussman's "Lambda the ultimate" papers [41, 42] persuasively argued that the λ-calculus is far more than a theoretical model of computation: it is an incredibly expressive and practical intermediate language for a compiler. The Rabbit compiler [40], its successors (e.g. Orbit [21]), and Appel's book "Compiling with continuations" [1] all demonstrate the power and utility of the λ-calculus as a compiler's intermediate language.

The typed λ-calculus arises canonically as the term language for a logic called *natural deduction* [14], using the *Curry-Howard isomorphism* [45]: the pervasive connection between logic and programming languages asserting that propositions are types and proofs are programs. Indeed, for many people, the λ-calculus is the living embodiment of Curry-Howard.

But natural deduction is not the only logic! Conspicuously, natural deduction has a twin, born in the very same paper [14], called the *sequent calculus*. Thanks to the Curry-Howard isomorphism, terms of the sequent calculus can also be seen as a programming language [9, 15, 44] with an emphasis on *control flow*.

This raises an obvious question: *does the language of the sequent calculus have merit as a practical compiler intermediate language, in the same way that the λ-calculus does?* What advantages and disadvantages might it have, compared to the existing λ-based technology? Curiously, we seem to be the first to address these questions, and surprisingly the task was not as routine as we had expected. Specifically, our contributions are these:

- We describe a typed sequent calculus called Sequent Core with the same expressiveness as System Fω, including **let**, algebraic data types, and **case** (Section 2).

  The broad outline of the language is determined by the logic, but we made numerous choices driven by its role as a compiler intermediate representation (Section 2.2).
- Our language comes equipped with an operational semantics (Section 2.3), a type system (Section 2.4), and standard meta-theoretical properties. We also give direct-style translations to and from System Fω (Section 3).[1]
- The proof of the pudding is in the eating. We have implemented our intermediate language as a plugin[2] for GHC, a state-of-the-art optimizing compiler for Haskell (Section 4). GHC's intermediate language, called Core, is essentially System Fω; our new plugin translates Core programs into Sequent Core, optimizes them, and translates them back. Moreover, we have re-implemented some of GHC's Core-to-Core optimization passes, notably the simplifier, to instead use Sequent Core.
- From the implementation, we found a way that Sequent Core was qualitatively better than Core for optimization: the treatment of *join points*. Specifically, join points in Sequent Core are preserved during simplifications such as the ubiquitous case-of-case transformation (Sections 4.2 and 4.3). Further, we show how to recover the join points of Sequent Core programs, after they are lost in translation, using a lightweight version of a process known as *contification* [20] (Section 5).

So what kind of intermediate language do we get out of the sequent calculus? It turns out that the language resembles *continuation-passing style*, a common technique in the λ-calculus for representing control flow inside a program. The division between assumptions and conclusions in the logic gives us a divide between programs that yield results and continuations that observe those results in the language. Yet despite the surface similarity, Sequent Core is still quite different from continuation-passing style (Section 6). Perhaps most importantly, Sequent Core brings control flow and continuations to a compiler like GHC without stepping on its toes, allowing its extensive direct-style optimizations to still shine through. In the end, we get an intermediate language that lies somewhere in between direct and continuation-passing styles (Section 7), sharing some advantages of both.

In a sense, many of the basic ideas we present here have been re-discovered over the years as the tradition of Curry-Howard

[1] More details of the meta-theory can be found in the appendix: `http://ix.cs.uoregon.edu/~pdownen/publications/scfp_ext.pdf`

[2] Available at: `http://github.com/lukemaurer/sequent-core`

Shared syntax of kinds and types

$$a, b, c \in \mathit{TyVar} ::= \dots$$
$$\kappa \in \mathit{Kind} ::= \star \mid \kappa \to \kappa$$
$$\tau, \sigma \in \mathit{Type} ::= a \mid T \mid \sigma\ \tau \mid \sigma \to \tau \mid \forall a{:}\kappa.\tau \mid \exists \overrightarrow{a{:}\kappa}.(\vec{\tau})$$

Syntax of Sequent Core

$$x, y, z \in \mathit{Var} ::= \dots \qquad j \in \mathit{Label} ::= \dots$$
$$\mathit{pgm} \in \mathit{Program} ::= \overrightarrow{\mathit{bind}}$$
$$\mathit{bind} \in \mathit{Bind} ::= \mathit{bp} \mid \mathbf{rec}\left\{\overrightarrow{\mathit{bp}}\right\}$$
$$\mathit{bp} \in \mathit{BindPair} ::= x{:}\tau = v \mid j{:}\tau = \tilde{\mu}[\overrightarrow{a{:}\kappa}, \overrightarrow{x{:}\sigma}].c$$
$$v \in \mathit{Term} ::= \lambda x{:}\tau.v \mid \Lambda a{:}\kappa.v \mid x \mid K(\vec{\sigma}, \vec{v}) \mid \mu\mathsf{ret}.c$$
$$c \in \mathit{Command} ::= \mathbf{let}\ \mathit{bind}\ \mathbf{in}\ c \mid \langle v \parallel k \rangle \mid \mathsf{jump}\ j\ \vec{\sigma}\ \vec{v}$$
$$k \in \mathit{Kont} ::= v \cdot k \mid \tau \cdot k \mid \mathbf{case\ of}\ \overrightarrow{\mathit{alt}} \mid \mathsf{ret}$$
$$\mathit{alt} \in \mathit{Alternative} ::= x{:}\tau \to c \mid K(\overrightarrow{a{:}\kappa}, \overrightarrow{x{:}\tau}) \to c$$

Syntax of Core

$$x, y, z \in \mathit{Var} ::= \dots$$
$$\mathit{pgm} \in \mathit{Program} ::= \overrightarrow{\mathit{bind}}$$
$$\mathit{bind} \in \mathit{Bind} ::= \mathit{bp} \mid \mathbf{rec}\left\{\overrightarrow{\mathit{bp}}\right\}$$
$$\mathit{bp} \in \mathit{BindPair} ::= x{:}\tau = v$$
$$e \in \mathit{Expression} ::= \mathbf{let}\ \mathit{bind}\ \mathbf{in}\ e \mid \lambda x{:}\tau.e \mid \Lambda a{:}\kappa.e \mid x \mid K\ \vec{\sigma}\ \vec{e} \mid e\ e \mid e\ \tau \mid \mathbf{case}\ e\ \mathbf{of}\ \overrightarrow{\mathit{alt}}$$
$$\mathit{alt} \in \mathit{Alternative} ::= x{:}\tau \to e \mid K\ \overrightarrow{a{:}\kappa}\ \overrightarrow{x{:}\tau} \to e$$

Figure 1. Syntax

dictates [45]: first by a logician and later by computer scientists. Our goal is to put them together in a way that is useful for compilers. Our implementation demonstrates that Sequent Core is certainly up to the job: in short order, we achieved performance competitive with a highly mature optimizing compiler. While we are not ready to recommend that GHC change its intermediate language, we instead see Sequent Core as an illuminating new design point in the space of functional programming languages and laboratory for experiments on intermediate representation techniques. We hope that our work will bring the sequent calculus forth, Cinderella-like, out of the theory kitchen and into the arms of compiler writers.

2. Sequent Core

In this section we present the specifics of our new sequent-style intermediate language for functional programs, along with its type system and operational semantics. The language that comes out of the logic "for free" is more expressive [11] than the pure λ-calculus, since it naturally speaks of control flow as a first-class entity. Thus, our task is to find the sweet spot between the permission to express interesting control flow and the restriction to pure functions.

2.1 Overview

Figure 1 gives the syntax of Sequent Core. For comparison purposes, we also give the syntax of Core, GHC's current intermediate language [38]. Both languages share the same syntax of types and kinds, also given in Figure 1. We omit two important features of Core, namely casts and unboxed types; both are readily accommodated in Sequent Core, and our implementation does so, but they distract from our main point.

Here is a small program written in both representations:

Core
$$\mathit{plusOne} : \mathit{Int} \to \mathit{Int} = \lambda x{:}\mathit{Int}.(+)\ x\ 1$$
Sequent Core
$$\mathit{plusOne} : \mathit{Int} \to \mathit{Int} = \lambda x{:}\mathit{Int}.\mu\mathsf{ret}.\ \langle (+) \parallel x \cdot 1 \cdot \mathsf{ret} \rangle$$

Referring to Figure 1, we see that

- Just as in Core, a Sequent Core *program* is a set of top-level *bindings*; bindings can be non-recursive or (mutually) recursive.
- A binding in Sequent Core is either a *value binding* $x{:}\tau = v$, or a *continuation binding* $j{:}\tau = \tilde{\mu}[\overrightarrow{a{:}\kappa}, \overrightarrow{x{:}\tau}].c$.[3] We discuss the latter in Section 2.2.2.
- The right-hand side of a value binding is a *term* v, which begins with zero or more lambdas followed by a variable, a constructor application, or a *computation* $\mu\mathsf{ret}.c$.
- The computation term $\mu\mathsf{ret}.c$, where c is a *command*, means "run command c and return whatever is passed to ret as the result of this term."
- All the interesting work gets done by commands. A command c is a collection of local bindings, wrapping either a *cut pair* $\langle v \parallel k \rangle$ or a *jump*. We discuss jumps in Section 2.2.2.
- Finally, cut pairs do some real computation. They are pairs $\langle v \parallel k \rangle$ of a term v and a *continuation* k. A continuation k is a *call stack* containing a sequence of applications (to types or terms) ending with either a **case** analysis or a return to ret.

In $\mathit{plusOne}$ as written in Sequent Core above, the calculation of the function is carried out by the command in its body: the term is the function $(+)$, while the continuation is $x \cdot 1 \cdot \mathsf{ret}$, meaning "apply to x, then apply to 1, then return." With this reading, Sequent Core cut pairs closely resemble the *states* of many abstract machines (e.g., the CEK machine [12]), with a term v in the focus and a continuation or call stack k that describes how it is consumed.

Here is another example program, written in both representations, that drops the last element of a list:

Core
$$\begin{aligned}&\mathit{init} : \forall a.[a] \to [a]\\ &= \Lambda a.\lambda x{:}[a].\ \mathbf{case}\ \mathit{reverse}\ a\ \mathit{xs}\ \mathbf{of}\\ &\qquad [\,] \qquad\quad \to [\,]\\ &\qquad (y : \mathit{ys}) \to \mathit{reverse}\ a\ \mathit{xs}\end{aligned}$$
Sequent Core
$$\begin{aligned}&\mathit{init} : \forall a.[a] \to [a]\\ &= \Lambda a.\lambda x{:}[a].\mu\mathsf{ret}.\ \langle \mathit{reverse} \| a \cdot \mathit{xs} \cdot \mathbf{case\ of}\\ &\qquad [\,] \qquad\quad \to [\,]\\ &\qquad (y : \mathit{ys}) \to \langle \mathit{reverse} \parallel a \cdot \mathit{ys} \cdot \mathsf{ret} \rangle\end{aligned}$$

As before, the outer structure is the same, but the **case**, which is so prominent in Core, appears in Sequent Core as the continuation of the call to *reverse*. Indeed, this highlights a key difference: in Sequent Core, the focus of evaluation is always "at the top", whereas in Core it may be deeply buried [4]. In this example, the call to *reverse* is the first thing to happen, and it is visibly at the top of the body of the lambda. In this way, Sequent Core's operational reading is somewhat more direct, a useful property for a compiler intermediate language.

[3] We occasionally omit type annotations in examples for clarity.

2.2 The Language

Having seen how Sequent Core is a language resembling an abstract machine, let's look more closely at the new linguistic concepts that it introduces and how Sequent Core compares to Core. On closer inspection, Sequent Core can be seen as a nuanced variation on Core, separating the roles of distinct concepts of Core syntactically as part of the effort to split calculations across the two sides of a cut pair. More specifically, each construct of Core has an exact analogue in Sequent Core, but the single grammar of Core expressions e is divided among terms v, continuations k, and commands c in Sequent Core. Additionally, Sequent Core has special support for labels and direct jumps, which are not found in Core.

2.2.1 Terms and Continuations

Core expressions e, as shown in Figure 1, include a variety of values (more specifically *weak-head normal forms*) which require no further evaluation: lambdas (both small λ and big Λ) and applied constructors. Along with variables, these are all terms in Sequent Core, as they do not involve any work to be done and they immediately produce themselves as their result.

On the other hand, Core also includes expressions which *do* require evaluation: function applications $e\ e'$, polymorphic instantiations $e\ \tau$, and **case** expressions. Each of these expressions *uses* something to create the next result, and thus these are reflected as continuations k in Sequent Core. As usual, Sequent Core continuations represent evaluation contexts that receive an input which will be used immediately. For example, the application context $\square\ 1$, where "$\square$" is the hole where the input is placed, corresponds to the call stack $1 \cdot \mathsf{ret}$. Furthermore, we can apply the curried function $\lambda x.\lambda y.x$ to the arguments 1 and 2 by running it in concert with the stack $1 \cdot 2 \cdot \mathsf{ret}$, as in:

$$\langle \lambda x.\lambda y.x \parallel 1 \cdot 2 \cdot \mathsf{ret}\rangle = \langle \lambda y.1 \parallel 2 \cdot \mathsf{ret}\rangle = \langle 1 \parallel \mathsf{ret}\rangle$$

where ret signals a stop, so that the result 1 can be returned.

Since we are interested in modeling *lazy* functional languages, we also need to include the results of arbitrary deferred computations as terms in themselves. For example, when we perform the lazy function composition $f\ (g\ x)$ in Core, $g\ x$ is only computed when f demands it. This means we need the ability to inject computations into terms, which we achieve with μ-abstractions. A μ-abstraction extracts a result from a command by *binding* the occurrences of ret in that command, so that anything passed to ret is returned from the μ-abstraction. However, because we are only modeling purely functional programs, there is only ever one ret available at a time, making it a rather limited namespace. Thus, $\mu\mathsf{ret}.\ \langle g \parallel x \cdot \mathsf{ret}\rangle$ runs the underlying command, calling the function g with the argument x, so that whatever is returned by g pops out as the result of the term. So lazy function composition can be written in Sequent Core as $\langle f \parallel (\mu\mathsf{ret}.\ \langle g \parallel x \cdot \mathsf{ret}\rangle) \cdot \mathsf{ret}\rangle$.

Notice that every closed command must give a result to ret if it ever stops at all. Another way of looking at this fact is that every (finite) continuation has ret "in the tail"; it plays the role of "nil" in a linked list. However, the return structure of continuations is more complex than a plain linked list, since the terminating ret of a continuation may occur in several places. By inspection, a continuation is a sequence of zero or more type or term applications, followed by either ret itself or by a **case** continuation. But in the latter case, each alternative has a command whose continuation must in turn have ret in the tail. Unfortunately, this analogy breaks down in the presence of local bindings, as we will see. Luckily, however, viewing ret as a static variable bound by μ-abstractions tells us exactly how to "chase the tail" of a continuation by following the normal rules of static scope. So we may still say that every closed computation $\langle v \parallel k\rangle$ eventually returns if it does not diverge.

2.2.2 Bindings and Jumps

There is one remaining Core expression to be sorted into the Sequent Core grammar: **let** bindings. In Sequent Core, **let** bindings are commands, as they set up an enclosing environment for another command to run in, forming an executable code block. In both representations, **let** bindings serve two purposes: to give a shared name to the result of some computation, and to express (mutual) recursion. Thus in the Sequent Core command c, we can share the results of terms through $\mathbf{let}\, x = v\, \mathbf{in}\, c$ and we can share a continuation through $\langle \mu\mathsf{ret}.c \parallel k\rangle$. But something is missing. How can we give a shared label to a command (i.e., to a block of code) that we can go to during the execution of another command? This facility is critical for maintaining small code size, so that we are not forced to repeat the same command verbatim in a program.

For example, suppose we have the command

$$\langle z \parallel \mathbf{case\,of}\ \mathit{Left}(x) \to c, \mathit{Right}(x) \to c\rangle$$

wherein the same c is repeated twice due to the **case** continuation. Now, how do we lift out and give a name to c, given that it contains the free variable x? We would rather not use a lambda, as in $\lambda x.\mu\mathsf{ret}.c$, since that introduces additional overhead compared to the original command. Instead, we would rather think of c as a sort of continuation whose input is named x during execution of c. In the syntax of $\overline{\lambda}\mu\tilde{\mu}$ [9], this would be written as $\tilde{\mu}x.c$, the *dual* of μ-abstractions. However, this is not like the other continuations we have seen so far! There is no guarantee that $\tilde{\mu}x.c$ uses its input immediately, or even at all. Thus, we are not dealing with an evaluation context, but rather an arbitrary context. Furthermore, we might (reasonably) want to name commands with multiple free variables, or even free type variables. So in actuality, we are looking for a representation of continuations taking *multiple values* as inputs of *polymorphic* types, corresponding to general contexts with multiple holes.

This need leads us to multiple-input continuations, which we write as $\tilde{\mu}[a_1, \ldots, a_n, x_1, \ldots, x_m].c$ in the style of $\overline{\lambda}\mu\tilde{\mu}$. These continuations accept several inputs (named $x_1 \ldots x_m$), whose types are polymorphic over the choice of types for $a_1 \ldots a_n$, in order to run a command c. Intuitively, we may also think of these multiple-input continuations as a sequence of lambdas $\Lambda a_1 \ldots a_n.\lambda x_1 \ldots x_m.c$, except that the body is a command because it does not return. The purpose of introducing multiple-input continuations was to lift out and name arbitrary commands, and so they appear as a Sequent Core binding. Specifically, all multiple-input continuations in Sequent Core are given a label j, as in the continuation binding $j = \tilde{\mu}[x, y].\ \langle (+) \parallel x \cdot y \cdot \mathsf{ret}\rangle$. These labeled continuations serve as *join points*: places where the control flow of several diverging branches of a program joins back up again.

In order to invoke a bound continuation, we can *jump* to it by providing the correct number of terms for the inputs, as well as explicitly specifying the instantiation of any polymorphic type in System Fω style. For example, the command

$$\begin{array}{l}\mathbf{let}\, j = \tilde{\mu}[a{:}\star, x{:}a, f{:}a \to \mathit{Bool}].\ \langle f \parallel x \cdot \mathsf{ret}\rangle \\ \mathbf{in}\, \mathsf{jump}\ j\ \mathit{Bool}\ \mathit{True}\ \mathit{not}\end{array}$$

will jump to the label j with the inputs Bool, True, and not, which results in $\langle \mathit{not} \parallel \mathit{True} \cdot \mathsf{ret}\rangle$. So when viewing Sequent Core from the perspective of an abstract machine, its command language provides three instructions: (1) set a binding with **let**, (2) evaluate an expression with a cut pair, or (3) perform a direct jump.

Take note that a labeled continuation does not introduce a μ-binder. As a consequence, the ret found in $j = \tilde{\mu}[x, y].\ \langle (+) \parallel x \cdot y \cdot \mathsf{ret}\rangle$ refers to the nearest surrounding μ, unlike the ret found in $f = \lambda x.\lambda y.\mu\mathsf{ret}.\ \langle (+) \parallel x \cdot y \cdot \mathsf{ret}\rangle$. Viewing ret as a statically bound variable means that labeled continuations participate in the "tail chasing" discussed previously in Section 2.2.1. Thus, the ret

structure of commands and continuations treats labeled continuations quite the same as $\mathbf{case}$ alternatives for free.

2.2.3 The Scope of Labels

There is one major restriction that we enforce to ensure that each term must have a *unique* exit point by which it returns its result, and so evaluating a term cannot cause an observable jump to some surrounding continuation binding. The intuition is:

> Terms contain no free references to continuation variables,

where continuation variables can be labels j as well as ret. This restriction, similar to a restriction on CPS [20], makes sure that lambdas cannot close over labels available from their contexts, so that labels do not escape through returned lambdas. Thus, all jumps within the body of a lambda must be local. Likewise, in all computations $\mu\mathsf{ret}.c$, the underlying command c has precisely one unique exit point from which the computation can return a result, denoted by ret. Therefore, all jumps made during the execution of c are internal to c, and unobservable during evaluation of $\mu\mathsf{ret}.c$.

Notice that this restriction on the scope of continuation variables, while not very complex, still manages to tell us something about the expressive capabilities of Sequent Core. For example, we syntactically permit value and continuation bindings within the same recursive block, but can they mutually call one another? It turns out that the scoping restriction disallows any sort of interesting mutual recursion between terms and continuations, because terms are *prevented* from referencing labels within their surrounding (or same) binding environment. Specifically, there is some additional structure implicit to $\mathbf{let}$ bindings:

- Continuation bindings can reference value bindings and other continuation bindings, but value bindings can only reference other value bindings.
- In any sequence of bindings, all value bindings can always be placed before all continuation bindings.
- Value and continuation bindings cannot be mutually recursive. Any minimal, mutually recursive $\mathbf{rec}\left\{\overrightarrow{bp}\right\}$ block will consist of only value bindings or only continuation bindings.

For example, consider the recursive bindings:

$$\mathbf{rec}\left\{f = \lambda x.v, j = \tilde{\mu}[y].c\right\}$$

By the scoping rules, j may call f through c, but f cannot jump back to j in v because $\lambda x.v$ cannot contain a free reference to j. Therefore, since there is no true mutual recursion between f and j, we can break the recursive bindings into two separate blocks with the correct scope, placing the binding for f first:

$$\mathbf{rec}\left\{f = \lambda x.v\right\}, \mathbf{rec}\left\{j = \tilde{\mu}[y].c\right\}$$

While we do not syntactically enforce this separation, doing so would not cause any loss of expressiveness. Indeed, we could normalize all commands by gathering and partitioning all bindings into (1) first, the list of value bindings, Γ, and (2) second, the list of continuation bindings, Δ, so that commands have the form $\mathbf{let}\,\Gamma\,\mathbf{in}\,\mathbf{let}\,\Delta\,\mathbf{in}\,\langle v \,\|\, k\rangle$. However, we do not enforce this normal form in Sequent Core.

2.3 Operational Semantics

A useful way to understand Sequent Core is through its operational semantics, given in Figure 2, which provides a high-level specification for reasoning about the correctness of program transformations. The rules for lambda (both small λ and big Λ) are self-explanatory. The rules for $\mathbf{case}$ are disambiguated by selecting the first match, so the order of alternatives matters. For a non-recursive $\mathbf{let}$, we simply substitute, thus implementing call-by-name; implementing recursive

$$W \in WHNF ::= \lambda x{:}\tau.v \mid \Lambda a{:}\kappa.v \mid x \mid K(\vec{\sigma}, \vec{v})$$

$$\begin{aligned}
\langle \lambda x{:}\tau.v \,\|\, v' \cdot k\rangle &\mapsto \langle v\,\{v'/x\} \,\|\, k\rangle \\
\langle \Lambda a{:}\kappa.v \,\|\, \tau \cdot k\rangle &\mapsto \langle v\,\{\tau/a\} \,\|\, k\rangle \\
\left\langle K(\vec{\sigma}, \vec{v}) \,\middle\|\, \mathbf{case\,of}\,\overrightarrow{alt}\right\rangle &\mapsto c\overrightarrow{\{\sigma/b\}}\overrightarrow{\{v/x\}} && K(\overrightarrow{b{:}\kappa}, \overrightarrow{x{:}\tau}) \to c \in \overrightarrow{alt} \\
\left\langle W \,\middle\|\, \mathbf{case\,of}\,\overrightarrow{alt}\right\rangle &\mapsto c\,\{W/x\} && x{:}\tau \to c \in \overrightarrow{alt} \\
\langle \mu\mathsf{ret}.c \,\|\, k\rangle &\mapsto c\,\{k/\mathsf{ret}\} \\
\mathbf{let}\,x{:}\tau = v\,\mathbf{in}\,c &\mapsto c\,\{v/x\} \\
\mathbf{let}\,j{:}\tau' = \tilde{\mu}[\overrightarrow{a{:}\kappa}, \overrightarrow{x{:}\tau}].c'\,\mathbf{in}\,c &\mapsto c\left\{c'\overrightarrow{\{\sigma/a\}}\overrightarrow{\{v/x\}}/\mathsf{jump}\,j\,\vec{\sigma}\,\vec{v}\right\}
\end{aligned}$$

Figure 2. Call-by-name operational semantics

$\mathbf{let}$ is only slightly harder, but we omit it here for simplicity. Note that the rule for continuation $\mathbf{let}$s uses *structural substitution* [2, 28], which replaces every command matching the pattern $\mathsf{jump}\,j\,\vec{\sigma}\,\vec{v}$ with the given command. Intuitively, we can think of this substitution as inlining the right-hand side for j everywhere, and then β-reducing the jump at each inline site.

Note that Figure 2 serves equally well as an abstract machine, since every rule applies to the top of a command without having to search for a redex. Figure 2 can also be extended to a reduction theory for Sequent Core by permitting the rules to apply in any context, and further to an equational theory by symmetry, thereby providing a specification for valid transformations that a compiler might apply. Thus, a call-by-name operational semantics and an abstract machine are the same for Sequent Core, and the difference between a reduction and an equational theory is the difference between reducing anywhere or reducing only at the top of a command.

The most interesting rule is the one for computations:

$$\langle \mu\mathsf{ret}.c \,\|\, k\rangle \quad \mapsto \quad c\,\{k/\mathsf{ret}\}$$

If the computation $\mu\mathsf{ret}.c$ is consumed by continuation k, then we can just substitute k for all the occurrences of ret in c. From the point of view of a control calculus or continuation-passing style, ret can be seen as a static variable bound by μ-abstractions, providing the correct notion of substitution. Another way to think about it is that the substitution $c\,\{k/\mathsf{ret}\}$ appends k to the continuation(s) of c, including those in labeled continuations but not under any intervening μ-abstractions. For example:

$$\langle f \,\|\, x \cdot (\mu\mathsf{ret}.c) \cdot \mathsf{ret}\rangle\,\{y \cdot \mathsf{ret}/\mathsf{ret}\} = \langle f \,\|\, x \cdot (\mu\mathsf{ret}.c) \cdot y \cdot \mathsf{ret}\rangle$$

Expressing call-by-need simply requires the addition of a Launchbury-style [22] heap, as shown in Figure 3, which gives a lower-level operational reading of the different language constructs and shows how Sequent Core can be efficiently implemented. Note that unless otherwise specified in the rules, all additions to the heap $\mathcal{H}$ and jump environment $\mathcal{J}$ are assumed to be fresh, using α-renaming as necessary to pick a fresh variable or label. Specifically, the force and update rules modify an existing binding in the heap, whereas all the other rules allocate new heap bindings. Also note that for simplicity of the $\mathbf{let}$ rules, we assume that value bindings, $bind_v$, are kept separate from continuation bindings, $bind_k$, which can always be done as described in Section 2.2.3.

The main thing to notice about this semantics is how the different components of the state are separated—the heap $\mathcal{H}$, the jump environment $\mathcal{J}$, and the linear continuation $\mathcal{R}$—which is only possible because of our scope restrictions described in Section 2.2.3. Specifically, every rule that allocates in the heap uses the fact that terms cannot access the labels in the jump environment, and the μ and force rules use the fact that a μ-abstraction starts a fresh scope of labels. The scoping rules further allow these different components

$$V \in \mathit{Value} ::= \lambda x{:}\tau.v \mid \Lambda a{:}\kappa.v \mid x \mid K(\vec{\sigma}, \vec{V}) \qquad W \in \mathit{WHNF} ::= V \mid K(\vec{\sigma}, \vec{v})$$

$$\mathcal{H} \in \mathit{Heap} ::= \varepsilon \mid \Gamma, x = v \mid \Gamma, x = \bullet \qquad \mathcal{J} \in \mathit{JumpEnv} ::= \varepsilon \mid \mathcal{J}, j = \tilde{\mu}[\overrightarrow{a{:}\kappa}\ \overrightarrow{x{:}\tau}].c \qquad \mathcal{R} \in \mathit{LinKont} ::= \varepsilon \mid (k, \mathcal{J}) : \mathcal{R} \mid \mathbf{upd}\, x : \mathcal{R}$$

$$\boxed{\langle \Gamma;\ \mathcal{J}, \mathcal{R};\ c \rangle \rightsquigarrow \langle \Gamma;\ \mathcal{J}, \mathcal{R};\ c \rangle}$$

$$\begin{array}{rll}
(\beta^{\to}) & \langle \mathcal{H};\ \mathcal{J}, \mathcal{R};\ \langle \lambda x{:}\tau.v_1 \,\|\, v_2 \cdot k \rangle \rangle \rightsquigarrow \langle \mathcal{H}, x = v_2;\ \mathcal{J}, \mathcal{R};\ \langle v_1 \,\|\, k \rangle \rangle & \\
(\beta^{\forall}) & \langle \mathcal{H};\ \mathcal{J}, \mathcal{R};\ \langle \Lambda a{:}\kappa.v \,\|\, \tau \cdot k \rangle \rangle \rightsquigarrow \langle \mathcal{H};\ \mathcal{J}, \mathcal{R};\ \langle v\,\{\tau/a\} \,\|\, k \rangle \rangle & \\
(\mathbf{case}_{cons}) & \langle \mathcal{H};\ \mathcal{J}, \mathcal{R};\ \left\langle K(\vec{\sigma}, \vec{v}) \,\middle\|\, \mathbf{case\,of}\ \overrightarrow{alt} \right\rangle \rangle \rightsquigarrow \langle \mathcal{H}, \overrightarrow{x = v};\ \mathcal{J}, \mathcal{R};\ c\overrightarrow{\{\sigma/a\}} \rangle & K(\overrightarrow{a{:}\kappa}, \overrightarrow{x{:}\tau}) \to c \in \overrightarrow{alt} \\
(\mathbf{case}_{def}) & \langle \mathcal{H};\ \mathcal{J}, \mathcal{R};\ \left\langle W \,\middle\|\, \mathbf{case\,of}\ \overrightarrow{alt} \right\rangle \rangle \rightsquigarrow \langle \mathcal{H}, x = W;\ \mathcal{J}, \mathcal{R};\ c \rangle & x \to c \in \overrightarrow{alt} \\
(\mu) & \langle \mathcal{H};\ \mathcal{J}, \mathcal{R};\ \langle \mu \mathsf{ret}.c \,\|\, k \rangle \rangle \rightsquigarrow \langle \mathcal{H};\ \varepsilon, (k, \mathcal{J}) : \mathcal{R};\ c \rangle & \\
(\mathsf{jump}) & \langle \mathcal{H};\ \mathcal{J}, \mathcal{R};\ \mathsf{jump}\ j\ \vec{\sigma}\ \vec{v} \rangle \rightsquigarrow \langle \mathcal{H}, \overrightarrow{x = v};\ \mathcal{J}, \mathcal{R};\ c\overrightarrow{\{\sigma/a\}} \rangle & j = \tilde{\mu}[\overrightarrow{a{:}\kappa}, \overrightarrow{x{:}\tau}].c \in \mathcal{J} \\
(\mathsf{lookup}) & \langle \mathcal{H};\ \mathcal{J}, \mathcal{R};\ \langle x \,\|\, k \rangle \rangle \rightsquigarrow \langle \mathcal{H};\ \mathcal{J}, \mathcal{R};\ \langle V \,\|\, k \rangle \rangle & x = V \in \mathcal{H} \\
(\mathsf{lazysubst}) & \langle \mathcal{H};\ \mathcal{J}, \mathcal{R};\ \langle x \,\|\, k \rangle \rangle \rightsquigarrow \langle \mathcal{H}, \overrightarrow{y = v}, x = K(\vec{\sigma}, \vec{y});\ \mathcal{J}, \mathcal{R};\ \langle K(\vec{\sigma}, \vec{y}) \,\|\, k \rangle \rangle & x = K(\vec{\sigma}, \vec{v}) \in \mathcal{H} \\
(\mathsf{force}) & \langle \mathcal{H};\ \mathcal{J}, \mathcal{R};\ \langle x \,\|\, k \rangle \rangle \rightsquigarrow \langle \mathcal{H}, x = \bullet;\ \varepsilon, \mathbf{upd}\, x : (k, \mathcal{J}) : \mathcal{R};\ c \rangle & x = \mu \mathsf{ret}.c \in \mathcal{H} \\
(\mathsf{update}) & \langle \mathcal{H};\ \mathcal{J}, \mathbf{upd}\, x : \mathcal{R};\ \langle W \,\|\, \mathsf{ret} \rangle \rangle \rightsquigarrow \langle \mathcal{H}, x = W;\ \mathcal{J}, \mathcal{R};\ \langle W \,\|\, \mathsf{ret} \rangle \rangle & x = \bullet \in \mathcal{H} \\
(\mathsf{ret}) & \langle \mathcal{H};\ \mathcal{J}, (k', \mathcal{J}') : \mathcal{R};\ \langle W \,\|\, \mathsf{ret} \rangle \rangle \rightsquigarrow \langle \mathcal{H};\ \mathcal{J}', \mathcal{R};\ \langle W \,\|\, k' \rangle \rangle & \\
(\mathbf{let}_{val}) & \langle \mathcal{H};\ \mathcal{J}, \mathcal{R};\ \mathbf{let}\ \mathit{bind}_v\ \mathbf{in}\, c \rangle \rightsquigarrow \langle \mathcal{H}, \mathit{bind}_v;\ \mathcal{J}, \mathcal{R};\ c \rangle & \mathit{Dom}(\mathcal{H}) \cap \mathit{Dom}(\mathit{bind}_v) = \emptyset \\
(\mathbf{let}_{cont}) & \langle \mathcal{H};\ \mathcal{J}, \mathcal{R};\ \mathbf{let}\ \mathit{bind}_k\ \mathbf{in}\, c \rangle \rightsquigarrow \langle \mathcal{H};\ \mathcal{J}, \mathit{bind}_k, \mathcal{R};\ c \rangle & \mathit{Dom}(\mathcal{J}) \cap \mathit{Dom}(\mathit{bind}_k) = \emptyset
\end{array}$$

Figure 3. Call-by-need operational semantics

to embody different commonplace run-time entities. The heap $\mathcal{H}$ is of course implemented with a random-access mutable heap as usual. The linear continuation $\mathcal{R}$ is a mutable stack, since each element is accessed exactly once before disappearing. Contrastingly, the jump environment $\mathcal{J}$ serves only as syntactic bookkeeping for statically allocated code. Because of the scope restrictions, the binding for each label can be determined before execution, which is evident from the fact the $\mathbf{let}_{cont}$ side condition is guaranteed to hold whenever the initial program has distinct **let**-bound labels, making dynamic allocation unnecessary. So during execution the jump rule is a direct jump and the $\mathbf{let}_{cont}$ rule is nothing at all!

Even though the operational semantics of Figure 2 and Figure 3 implement different evaluation strategies—call-by-name and call-by-need, respectively—the two still produce the same answers. In particular, the abstract machine terminates if and only if the operational semantics does, which is enough to guarantee that the two semantics agree [24, 33].

Proposition 1 (Termination equivalence)**.** *For any closed command c, $c \mapsto^\star c_1 \not\mapsto$ if and only if $\langle \varepsilon;\ \varepsilon, \varepsilon;\ c \rangle \rightsquigarrow^\star \langle \mathcal{H};\ \mathcal{J}, \mathcal{R};\ c_2 \rangle \not\rightsquigarrow$.*

This should not be a surprise, as Sequent Core is intended for representing pure functional programs, and for the pure λ-calculus the two evaluation strategies agree [3].

2.4 Type System

Another way to understand Sequent Core is through its type system, which is given in Figure 4. Unsurprisingly, the type system for Sequent Core is based on the sequent calculus by reflecting programs as right rules and continuations as left rules. In particular, it is an extension (as well as a restriction) of the type system for the $\overline{\lambda}\mu\tilde{\mu}$-calculus [9, 16], and likewise we share the same unconventional notation for typing judgements to maintain a visible connection with the sequent calculus. More specifically, the typing judgements for the different syntactic categories of Sequent Core are written with the following sequents:

- The type of a command is described by $c : (\Gamma \vdash \Delta)$, which says that c is a well-typed command and may contain free variables described by Γ and Δ. Note that the command itself does not have a type directly, as terms and continuations do, because it does not take input or produce output directly. Rather, the "return" type of a command is in Δ, as in $\langle 1 \,\|\, \mathsf{ret} \rangle : (\varepsilon \vdash \mathsf{ret} : \mathit{Int})$.
- The type of a term is described by $\Gamma \vdash v : \tau$, which has the usual reading for typing terms of System Fω. In particular, it says that v returns a result of type τ and may contain free variables (for either values or types) with the types described by Γ.
- The type of a continuation is described by $\Gamma \mid k : \tau \vdash \Delta$, which says that k consumes an input of type τ and may contain free variables with the types described by Γ and Δ.
- The type of a binding is described by $\mathit{bind} : (\Gamma \mid \Delta' \vdash \Gamma' \mid \Delta)$, which is the most complex form of sequent. In essence, it says that bind binds the variables in Γ' and Δ' and may contain references to free variables from Γ and Δ. For example, we have $(x{:}\mathit{Int} = z + 1) : (z : \mathit{Int} \mid \varepsilon \vdash x : \mathit{Int} \mid \mathsf{ret} : \mathit{Bool})$ and $(j{:}\mathit{Int} = \tilde{\mu}[x{:}\mathit{Int}].\ \langle x \,\|\, \mathsf{ret} \rangle) : (\varepsilon \mid j : \mathit{Int} \vdash \varepsilon \mid \mathsf{ret} : \mathit{Int})$.

One important detail to note is the careful treatment of the continuation environment Δ in the rules of Figure 4. In particular, the term-typing judgement is missing Δ, which enforces the scoping restriction of continuation variables discussed in Section 2.2.3. A consequence of this fact is that Δ is treated linearly in the type system; it is only duplicated across the multiple alternatives of a **case** or in continuation bindings. Type variables in the environment $(\Gamma, a : \kappa, \Gamma')$ additionally scope over the remainder of the environment (Γ') as well as the entire conclusion (either Δ or $v : \tau$) and obey static scoping rules. Thus, the $\forall$R, TL, and Label rules only apply if they do not violate the static scopes of type variables.

The unusual notation we used for the type system makes it easy to bridge the gap between Sequent Core and the sequent calculus. In particular, if we drop all expressions (commands, *etc.*) and vertical bars from Figure 4, we end up with a corresponding logic of the sequent calculus, as shown in Figure 5 (where the type kinding is identical to Figure 4). All the similarly named rules come directly from erasing extra information from Figure 4, and additionally Ax represents both Var and Ret, MultiCut represents Let, WR represents Name, and the rest of the typing rules do nothing in

$$\Gamma \in \mathit{Environment} ::= \varepsilon \mid \Gamma, x:\tau \mid \Gamma, a:\kappa \mid \Gamma, K:\tau \mid \Gamma, T:\kappa \qquad \Delta \in \mathit{CoEnvironment} ::= \varepsilon \mid \mathsf{ret}:\tau \mid \Delta, j:\tau$$

Type kinding: $\boxed{\Gamma \vdash \tau : \kappa}$

$$\frac{}{\Gamma, a:\kappa \vdash a:\kappa}\ \mathsf{TyVar} \qquad \frac{}{\Gamma, T:\kappa \vdash T:\kappa}\ \mathsf{TyCon} \qquad \frac{\Gamma \vdash \sigma:\kappa' \to \kappa \quad \Gamma \vdash \tau:\kappa'}{\Gamma \vdash \sigma\,\tau:\kappa}\ \mathsf{TyApp} \qquad \frac{\Gamma, a:\kappa \vdash \tau:\star}{\Gamma \vdash \forall a{:}\kappa.\tau:\star}\ \forall \qquad \frac{\overrightarrow{\Gamma, \overrightarrow{a:\kappa} \vdash \tau:\star}}{\Gamma \vdash \exists\overrightarrow{a{:}\kappa}.(\vec{\tau}):\star}\ \exists\times$$

Command typing: $\boxed{c : (\Gamma \vdash \Delta)}$

$$\frac{bind : (\Gamma \mid \Delta' \vdash \Gamma' \mid \Delta) \quad c : (\Gamma, \Gamma' \vdash \Delta', \Delta)}{\mathbf{let}\ bind\ \mathbf{in}\ c : (\Gamma \vdash \Delta)}\ \mathsf{Let} \qquad \frac{\Gamma \vdash v:\tau \quad \Gamma \mid k:\tau \vdash \Delta}{\langle v \parallel k \rangle : (\Gamma \vdash \Delta)}\ \mathsf{Cut} \qquad \frac{\overrightarrow{\Gamma \vdash \sigma:\kappa} \quad \overrightarrow{\Gamma \vdash v:\tau\{\overrightarrow{\sigma/a}\}}}{\mathsf{jump}\ j\ \vec{\sigma}\ \vec{v} : (\Gamma \vdash j:\exists\overrightarrow{a{:}\kappa}.(\vec{\tau}), \Delta)}\ \mathsf{Jump}$$

Term typing: $\boxed{\Gamma \vdash v : \tau}$

$$\frac{}{\Gamma, x:\tau \vdash x:\tau}\ \mathsf{Var} \qquad \frac{c : (\Gamma \vdash \mathsf{ret}:\tau)}{\Gamma \vdash \mu\mathsf{ret}.c:\tau}\ \mathsf{Act} \qquad \frac{\Gamma, x:\sigma \vdash v:\tau}{\Gamma \vdash \lambda x{:}\sigma.v:\sigma \to \tau}\ {\to}\mathsf{R} \qquad \frac{\Gamma, a:\kappa \vdash v:\tau}{\Gamma \vdash \Lambda a{:}\kappa.v:\forall a{:}\kappa.\tau}\ \forall\mathsf{R}$$

$$\frac{K:\forall\overrightarrow{a{:}\kappa}.\forall\overrightarrow{b{:}\kappa'}.\overrightarrow{\tau'} \to T\ \vec{a} \in \Gamma \quad \overrightarrow{\Gamma \vdash \sigma:\kappa'} \quad \overrightarrow{\Gamma \vdash v:\tau'\{\overrightarrow{\tau/a}\}\{\overrightarrow{\sigma/b}\}}}{\Gamma \vdash K(\vec{\sigma}, \vec{v}):T\ \vec{\tau}}\ \mathsf{TRK}$$

Continuation typing: $\boxed{\Gamma \mid k:\tau \vdash \Delta}$ and Alternative typing: $\boxed{\Gamma \mid alt:\tau \vdash \Delta}$

$$\frac{}{\Gamma \mid \mathsf{ret}:\tau \vdash \mathsf{ret}:\tau, \Delta}\ \mathsf{Ret} \qquad \frac{\Gamma \vdash v:\sigma \quad \Gamma \mid k:\tau \vdash \Delta}{\Gamma \mid v \cdot k:\sigma \to \tau \vdash \Delta}\ {\to}\mathsf{L} \qquad \frac{\Gamma \vdash \sigma:\kappa \quad \Gamma \mid k:\tau\{\sigma/a\} \vdash \Delta}{\Gamma \mid \sigma \cdot k:\forall a{:}\kappa.\tau \vdash \Delta}\ \forall\mathsf{L} \qquad \frac{\overrightarrow{\Gamma \mid alt:\tau \vdash \Delta}}{\Gamma \mid \mathbf{case\ of}\ \overrightarrow{alt}:\tau \vdash \Delta}\ \mathsf{Case}$$

$$\frac{c : (\Gamma, x:\tau \vdash \Delta)}{\Gamma \mid x{:}\tau \to c:\tau \vdash \Delta}\ \mathsf{Deflt} \qquad \frac{K:\forall\overrightarrow{a{:}\kappa'}.\forall\overrightarrow{b{:}\kappa}.\vec{\sigma} \to T\ \vec{a} \in \Gamma \quad c : (\Gamma, \overrightarrow{b:\kappa}, \overrightarrow{x:\sigma\{\overrightarrow{\tau/a}\}} \vdash \Delta)}{\Gamma \mid K(\overrightarrow{b{:}\kappa}, \overrightarrow{x{:}\sigma}) \to c:T\ \vec{\tau} \vdash \Delta}\ \mathsf{TLK}$$

Binding typing: $\boxed{bind : (\Gamma \mid \Delta' \vdash \Gamma' \mid \Delta)}$ and $\boxed{bp : (\Gamma \mid \Delta' \vdash \Gamma' \mid \Delta)}$

$$\frac{\Gamma \vdash v:\tau}{(x{:}\tau = v) : (\Gamma \mid \varepsilon \vdash x:\tau \mid \Delta)}\ \mathsf{Name} \qquad \frac{c : (\Gamma, \overrightarrow{a:\kappa}, \overrightarrow{x:\tau} \vdash \Delta)}{(j{:}\exists\overrightarrow{a{:}\kappa}.(\vec{\tau}) = \tilde{\mu}[\overrightarrow{a{:}\kappa}, \overrightarrow{x{:}\tau}].c) : (\Gamma \mid j:\exists\overrightarrow{a{:}\kappa}.(\vec{\tau}) \vdash \varepsilon \mid \Delta)}\ \mathsf{Label}$$

$$\frac{\Gamma' = \overrightarrow{\Gamma''} \quad \Delta' = \overrightarrow{\Delta''} \quad \overrightarrow{bp : (\Gamma, \Gamma' \mid \Delta'' \vdash \Gamma'' \mid \Delta', \Delta)}}{\mathbf{rec}\left\{\overrightarrow{bp}\right\} : (\Gamma \mid \Delta' \vdash \Gamma' \mid \Delta)}\ \mathsf{Rec}$$

Figure 4. Type System

$$\Gamma \in \mathit{Assumption} ::= \varepsilon \mid \Gamma, \tau \mid \Gamma, K:\tau \mid \Gamma, a:\kappa \mid \Gamma, T:\kappa \qquad \Delta \in \mathit{Conclusion} ::= \varepsilon \mid \Delta, \tau$$

Structural rules:

$$\frac{}{\Gamma, \tau \vdash \tau, \Delta}\ \mathsf{Ax} \qquad \frac{\Gamma \vdash \tau \quad \Gamma, \tau \vdash \Delta}{\Gamma \vdash \Delta}\ \mathsf{Cut} \qquad \frac{\Gamma, \Delta' \vdash \Gamma', \Delta \quad \Gamma, \Gamma' \vdash \Delta', \Delta}{\Gamma \vdash \Delta}\ \mathsf{MultiCut} \qquad \frac{\Gamma \vdash \tau}{\Gamma \vdash \tau, \Delta}\ \mathsf{WR}$$

$$\frac{\overrightarrow{\Gamma, \tau \vdash \Delta}}{\Gamma, \tau \vdash \Delta}\ \mathsf{Case} \qquad \frac{\Gamma' = \overrightarrow{\Gamma''} \quad \Delta' = \overrightarrow{\Delta''} \quad \overrightarrow{\Gamma, \Gamma', \Delta'' \vdash \Gamma'', \Delta', \Delta}}{\Gamma, \Delta' \vdash \Gamma', \Delta}\ \mathsf{Rec}$$

Logical rules:

$$\frac{\Gamma, \sigma \vdash \tau}{\Gamma \vdash \sigma \to \tau}\ {\to}\mathsf{R} \qquad \frac{\Gamma \vdash \sigma \quad \Gamma, \tau \vdash \Delta}{\Gamma, \sigma \to \tau \vdash \Delta}\ {\to}\mathsf{L} \qquad \frac{\Gamma, a:\kappa \vdash \tau}{\Gamma \vdash \forall a{:}\kappa.\tau}\ \forall\mathsf{R} \qquad \frac{\Gamma \vdash \sigma:\kappa \quad \Gamma, \tau\{\sigma/a\} \vdash \Delta}{\Gamma, \forall a{:}\kappa.\tau \vdash \Delta}\ \forall\mathsf{L}$$

$$\frac{\overrightarrow{\Gamma \vdash \sigma:\kappa} \quad \overrightarrow{\Gamma \vdash \tau\{\overrightarrow{\sigma/a}\}}}{\Gamma \vdash \exists\overrightarrow{a{:}\kappa}.(\vec{\tau}), \Delta}\ \mathsf{Jump} \qquad \frac{\Gamma, \overrightarrow{a:\kappa}, \vec{\tau} \vdash \Delta}{\Gamma, \exists\overrightarrow{a{:}\kappa}.(\vec{\tau}) \vdash \Delta}\ \mathsf{Label}$$

$$\frac{K:\forall\overrightarrow{a{:}\kappa}.\forall\overrightarrow{b{:}\kappa'}.\overrightarrow{\tau'} \to T\ \vec{a} \in \Gamma \quad \overrightarrow{\Gamma \vdash \sigma:\kappa'} \quad \overrightarrow{\Gamma \vdash \tau'\{\overrightarrow{\tau/a}\}\{\overrightarrow{\sigma/b}\}}}{\Gamma \vdash T\ \vec{\tau}}\ \mathsf{TRK} \qquad \frac{K:\forall\overrightarrow{a{:}\kappa}.\forall\overrightarrow{b{:}\kappa'}.\vec{\sigma} \to T\ \vec{a} \in \Gamma \quad \Gamma, \overrightarrow{b:\kappa'}, \overrightarrow{\sigma\{\overrightarrow{\tau/a}\}} \vdash \Delta}{\Gamma, T\ \vec{\tau} \vdash \Delta}\ \mathsf{TLK}$$

Figure 5. Logical interpretation

the logic. This logic resembles Gentzen's original sequent calculus LK [14], but there are still some differences—besides just choice of connectives for types and kinds—that come from its application as an intermediate language for functional programs:

- LK has explicit inference rules for the structural properties of sequents: weakening (adding extraneous assumptions or conclusions), contraction (merging duplicate assumptions or conclusions), and exchange (swapping two assumptions or conclusions). Instead, the logic of Sequent Core makes these properties (except for weakening on the right, WR) implicit by generalizing the initial Ax rule to allow for extraneous assumptions and conclusions, and by duplicating the assumptions and conclusions in rules like Cut, MultiCut, and $\to$L. This comes from the interpretation of static variables in the language as side assumptions and side conclusions. However, implicit and explicit presentations of structural properties are logically equivalent to one another.
- LK is a *classical logic*, which is achieved by allowing for any number of extra side conclusions in all the left (L) and right (R) rules. Contrarily, Gentzen's also introduced an *intuitionistic logic* LJ [14] as a variant of LK that just limits all rules to only allow exactly one conclusion at all times. Instead, the logic that comes out of Sequent Core lies in between these two systems: sometimes there can only be one conclusion, and sometimes there can be many. Furthermore, it happens that the well-typed terms of Sequent Core correspond to well-typed expressions of Core (see Theorem 4 in the following Section 3), so Sequent Core still captures a similar notion of purity from the λ-calculus. This demonstrates that there is room for more subtle variations on intuitionistic logic that lie between the freedom of LK and the purity of LJ.
- Unlike LK, Figure 5 is not logically consistent as is, corresponding to the fact that the presented type system for Sequent Core allows for non-terminating programs and does not force exhaustiveness of case analysis. In particular, the Rec rule can prove anything and would of course be left out of a consistent subset of the logic. Similarly, the Case and TLK rules should only be used to build proofs by exhaustive case analysis at a type, as in the (informally) derived TL rule:

$$\dfrac{\forall K : \overrightarrow{\forall a:\kappa}.\overrightarrow{\forall b:\kappa'}.\vec{\sigma} \to T\ \vec{\tau}.\ \dfrac{\Gamma, \overrightarrow{b:\kappa'}, \overrightarrow{\sigma\{\tau/a\}} \vdash \Delta}{\Gamma, T\ \vec{\tau} \vdash \Delta}\ \textsf{TLK}}{\Gamma, T\ \vec{\tau} \vdash \Delta}\ \textsf{Case}$$

  So a consistent subset of Figure 5 is attainable by further restricting the rules and the types along these lines.[4]

Perhaps the most complex rules in Figure 4 are Jump and Label for multiple-input continuations. It may seem a bit bizarre that the polymorphism is pronounced "exists" instead of "forall," but luckily the above Curry-Howard reading in Figure 5 helps explain what's going on. There are two instances of the Jump and Label rules that are helpful to consider. The first is when we have exactly two monomorphic inputs:

$$\dfrac{\Gamma \vdash \tau_1 \quad \Gamma \vdash \tau_2}{\Gamma \vdash (\tau_1, \tau_2)}\ \textsf{Jump} \qquad \dfrac{\Gamma, \tau_1, \tau_2 \vdash \Delta}{\Gamma, (\tau_1, \tau_2) \vdash \Delta}\ \textsf{Label}$$

[4] Besides removing Rec and merging Case with TLK, both recursive types (implicitly available from assumptions $K : \tau$) and the MultiCut would also need to be restricted for consistency. Interestingly, the Γ-Δ partitioning of **let** bindings described in Section 2.2.3 provides a sufficient consistency criterion for MultiCut. So maintaining the distinction between active conclusions (*i.e.*, terms) and assumptions (*i.e.*, continuations) as presented in Figure 4 in the logic is enough to tame MultiCut.

These are the right and left rules for the (tensor) product type in the sequent calculus—the right Jump rule allows only one conclusion like LJ and the left Label rule allows multiple conclusions like LK—illustrating the (tuple) product nature of multiple-input continuations and jumps. Second is when we have one polymorphic type quantified over one input:

$$\dfrac{\Gamma \vdash \sigma : \kappa \quad \Gamma \vdash \tau\{\sigma/a\}}{\Gamma \vdash \exists a{:}\kappa.\tau}\ \textsf{Jump} \qquad \dfrac{\Gamma, a : \kappa, \tau \vdash \Delta}{\Gamma, \exists a{:}\kappa.\tau \vdash \Delta}\ \textsf{Label}$$

These are exactly the right and left rules for existential types in the sequent calculus—with the same comparison to the right rule of LJ and the left rule of LK—which justifies the use of $\exists$ for the types of labels and jumps. Notice that this special case giving existential quantification is formally dual[5] to the rules $\forall$L and $\forall$R for universal quantification by transporting conclusions on the right of the turnstile ($\vdash$) to the left and vice versa (except for the quantified σ and a, which stay put).

Furthermore, notice that the static scope of type variables uniformly gives the correct logical provisos for the quantifiers in the $\forall$R, Label, and TLK rules. For example, the following command from Section 2.2.2

$$\begin{array}{l}\mathbf{let}\ j{:}\exists a{:}\star.(a, a{\to}Bool) = \tilde{\mu}[a{:}\star, x{:}a, f{:}a{\to}Bool].\ \langle f \,\|\, x \cdot \mathsf{ret}\rangle \\ \mathbf{in}\,\mathsf{jump}\ j\ Bool\ True\ not\end{array}$$

is typable by the sequent ($\varepsilon \vdash \mathsf{ret} : Bool$) because the type of the free variable ret does not reference the locally quantified a. However, the seemingly similar command

$$\begin{array}{l}\mathbf{let}\ j{:}\exists a{:}\star.(a, a{\to}a) = \tilde{\mu}[a{:}\star, x{:}a, f{:}a{\to}a].\ \langle f \,\|\, x \cdot \mathsf{ret}\rangle \\ \mathbf{in}\,\mathsf{jump}\ j\ Bool\ True\ not\end{array}$$

is not typable by the sequent ($\varepsilon \vdash \mathsf{ret} : a$)—or any other one—because a is local to the definition of j, and thus cannot escape in the type of ret. Pattern matching on existential data types in Haskell follows restrictions that are analogous to these scoping rules.

Finally, observe that the type system of Figure 4 is enough to ensure that well-typed programs don't go wrong according to the operational semantics of Figure 2. In particular, type safety follows from standard lemmas for *progress* (every well-typed command is either a final state or can take a step) and *preservation* (every well-typed command is still well-typed after taking a step).

Proposition 2 (Type safety)**.** *1. Progress: If* $c : (\varepsilon \vdash \mathsf{ret} : \tau)$ *without* Rec*, then* $c \mapsto c'$ *or* c *has one of the following forms:* $\langle W \,\|\, \mathsf{ret}\rangle$ *or* $\left\langle K(\vec{\sigma}, \vec{v}) \,\middle\|\, \mathbf{case\,of}\ \overrightarrow{alt} \right\rangle$ *where neither* $K(\overrightarrow{b{:}\kappa}, \overrightarrow{x{:}\sigma}) \to c$ *nor* $x : \tau \to c$ *are in* $\overrightarrow{alt}$*.*
2. Preservation: If $c : (\Gamma \vdash \Delta)$ *and* $c \mapsto c'$ *then* $c' : (\Gamma \vdash \Delta)$*.*

Note that an unknown **case** is a possibility allowed by the type system as is, and just like with Core, ensuring exhaustiveness of case analysis rules this final state out so that the only result is $\langle W \,\|\, \mathsf{ret}\rangle$. Also, because Figure 2 does not account for recursive **let**s, neither does the above progress proposition. Recursion is not a problem for progress, but it does take some additional care to treat explicitly.

3. Translating to and from Core

Core and Sequent Core are equally expressive, and it is illuminating to give translations between them, in both directions. In practical terms these translations were useful in our implementation, because in order to fit Sequent Core into the compiler pipeline, we need to translate from Core to Sequent Core and back.

[5] The Jump and Label are not exactly dual to $\forall$L and $\forall$R in the classical sense due to the restriction of right rules to only one conclusion. Lifting this restriction to get the more LK-like logic makes these rules classically dual.

$$S\,[\![\lambda x{:}\tau.e]\!] = \lambda x{:}\tau.S\,[\![e]\!] \qquad S\,[\![x]\!] = x$$
$$S\,[\![\Lambda a{:}\kappa.e]\!] = \Lambda a{:}\kappa.S\,[\![e]\!] \qquad S\,[\![K\ \vec{\sigma}\ \vec{e}]\!] = K(\vec{\sigma}, \overrightarrow{S\,[\![e]\!]})$$

$$S\,[\![\mathbf{let}\ bind\ \mathbf{in}\ e]\!] = \mu\mathsf{ret}.\ \mathbf{let}\ S\,[\![bind]\!]\ \mathbf{in}\ \langle S\,[\![e]\!] \,\|\, \mathsf{ret}\rangle$$
$$S\,[\![e\ e']\!] = \mu\mathsf{ret}.\ \langle S\,[\![e]\!] \,\|\, S\,[\![e']\!] \cdot \mathsf{ret}\rangle$$
$$S\,[\![e\ \tau]\!] = \mu\mathsf{ret}.\ \langle S\,[\![e]\!] \,\|\, \tau \cdot \mathsf{ret}\rangle$$
$$S\left[\!\!\left[\mathbf{case}\ e\ \mathbf{of}\ \overrightarrow{alt}\right]\!\!\right] = \mu\mathsf{ret}.\ \left\langle S\,[\![e]\!] \,\middle\|\, \mathbf{case\ of}\ \overrightarrow{S\,[\![alt]\!]}\right\rangle$$

$$S\,[\![x{:}\tau = e]\!] = x{:}\tau = S\,[\![e]\!]$$
$$S\,[\![\mathbf{rec}\,\{\overrightarrow{x{:}\tau = e}\}]\!] = \mathbf{rec}\left\{\overrightarrow{x{:}\tau = S\,[\![e]\!]}\right\}$$

$$S\,[\![x{:}\tau \to e]\!] = x{:}\tau \to \langle S\,[\![e]\!] \,\|\, \mathsf{ret}\rangle$$
$$S\,[\![K\ \overrightarrow{a{:}\kappa}\ \overrightarrow{x{:}\tau} \to e]\!] = K(\overrightarrow{a{:}\kappa}, \overrightarrow{x{:}\tau}) \to \langle S\,[\![e]\!] \,\|\, \mathsf{ret}\rangle$$

Figure 6. Definitional translation from Core to Sequent Core

Core to Sequent Core

$$S_a\,[\![\lambda x{:}\tau.e]\!] = \lambda x{:}\tau.S_a\,[\![e]\!] \qquad S_a\,[\![x]\!] = x$$
$$S_a\,[\![\Lambda a{:}\kappa.e]\!] = \Lambda a{:}\kappa.S_a\,[\![e]\!] \qquad S_a\,[\![K\ \vec{\sigma}\ \vec{e}]\!] = K(\vec{\sigma}, \overrightarrow{S_a\,[\![e]\!]})$$
$$S_a\,[\![e]\!] = \mu\mathsf{ret}.S_a\,[\![e]\!]\ \mathsf{ret} \qquad \text{where } e \text{ is a computation}$$

$$S_a\,[\![\mathbf{let}\ bind\ \mathbf{in}\ e]\!]\ k = \mathbf{let}\ S_a\,[\![bind]\!]\ \mathbf{in}\ S_a\,[\![e]\!]\ k$$
$$S_a\,[\![e\ e']\!]\ k = S_a\,[\![e]\!]\ (S_a\,[\![e']\!] \cdot k)$$
$$S_a\,[\![e\ \tau]\!]\ k = S_a\,[\![e]\!]\ (\tau \cdot k)$$
$$S_a\left[\!\!\left[\mathbf{case}\ e\ \mathbf{of}\ \overrightarrow{alt}\right]\!\!\right]\ k = \mathbf{let}\ \overrightarrow{bind}\ \mathbf{in}\ S_a\,[\![e]\!]\ (\mathbf{case\ of}\ \overrightarrow{S_a\,[\![alt]\!]\ k'})$$
$$\text{where } (\overrightarrow{bind}, k') = \mathsf{shrink}(k)$$
$$S_a\,[\![e]\!]\ k = \langle S_a\,[\![e]\!] \,\|\, k\rangle \qquad \text{where } e \text{ is a value}$$

$$S_a\,[\![x{:}\tau = e]\!] = x{:}\tau = S_a\,[\![e]\!]$$
$$S_a\,[\![\mathbf{rec}\,\{\overrightarrow{x{:}\tau = e}\}]\!] = \mathbf{rec}\left\{\overrightarrow{x{:}\tau = S_a\,[\![e]\!]}\right\}$$
$$S_a\,[\![x{:}\tau \to e]\!]\ k = x{:}\tau \to S_a\,[\![e]\!]\ k$$
$$S_a\,[\![K\ \overrightarrow{a{:}\kappa}\ \overrightarrow{x{:}\tau} \to e]\!]\ k = K(\overrightarrow{a{:}\kappa}, \overrightarrow{x{:}\tau}) \to S_a\,[\![e]\!]\ k$$

Sequent Core to Core

$$D\,[\![\lambda x{:}\tau.v]\!] = \lambda x{:}\tau.D\,[\![v]\!] \qquad D\,[\![x]\!] = x$$
$$D\,[\![\Lambda a{:}\kappa.v]\!] = \Lambda a{:}\kappa.D\,[\![v]\!] \qquad D\,[\![K(\vec{\sigma}, \vec{v})]\!] = K\ \vec{\sigma}\ \overrightarrow{D\,[\![v]\!]}$$
$$D\,[\![\mu\mathsf{ret}.c]\!] = D\,[\![c]\!]$$

$$D\,[\![\mathbf{let}\ bind\ \mathbf{in}\ c]\!] = \mathbf{let}\ D\,[\![bind]\!]\ \mathbf{in}\ D\,[\![c]\!]$$
$$D\,[\![\langle v \,\|\, k\rangle]\!] = D\,[\![k]\!]\,[D\,[\![v]\!]]$$
$$D\,[\![\mathsf{jump}\ j\ \vec{\sigma}\ \vec{v}]\!] = j\ \vec{\sigma}\ \overrightarrow{D\,[\![v]\!]}$$

$$D\,[\![v \cdot k]\!] = D\,[\![k]\!]\,[\square\ D\,[\![v]\!]] \qquad D\,[\![\mathsf{ret}]\!] = \square$$
$$D\,[\![\sigma \cdot k]\!] = D\,[\![k]\!]\,[\square\ \sigma] \qquad D\left[\!\!\left[\mathbf{case\ of}\ \overrightarrow{alt}\right]\!\!\right] = \mathbf{case}\ \square\ \mathbf{of}\ \overrightarrow{D\,[\![alt]\!]}$$

$$D\,[\![x{:}\tau = v]\!] = \big(x{:}\tau = D\,[\![v]\!]\big)$$
$$D\,[\![j{:}\exists\overrightarrow{a{:}\kappa}.(\vec{\tau}) = \tilde{\mu}[\overrightarrow{a{:}\kappa}, \overrightarrow{x{:}\tau}].c]\!] = \big(j{:}\forall\overrightarrow{a{:}\kappa}.\vec{\tau} \to \sigma = \Lambda\overrightarrow{a{:}\kappa}.\lambda\overrightarrow{x{:}\tau}.D\,[\![c]\!]\big)$$
$$\text{where } \mathsf{ret} : \sigma$$
$$D\left[\!\!\left[\mathbf{rec}\left\{\overrightarrow{bp}\right\}\right]\!\!\right] = \mathbf{rec}\left\{\overrightarrow{D\,[\![bp]\!]}\right\}$$
$$D\,[\![x{:}\sigma \to c]\!] = x{:}\sigma \to D\,[\![c]\!]$$
$$D\,[\![K(\overrightarrow{a{:}\kappa}, \overrightarrow{x{:}\sigma}) \to c]\!] = K\ \overrightarrow{a{:}\kappa}\ \overrightarrow{x{:}\sigma} \to D\,[\![c]\!]$$

Figure 7. Round-trip translations from Core to Sequent Core (S_a) and back to Core (D)

3.1 The Definitional Translation

Luckily, we can leverage the relationship between natural deduction and sequent calculus [9] to come up with a definitional translation from Sequent Core to Core, as shown in Figure 6. Notice that value expressions—lambdas and applied constructors, corresponding to introductory forms of natural deduction, as well as variables—translate one-for-one as terms of Sequent Core. Dealing with computations—applications and **case** expressions, corresponding to elimination forms, as well as **let** expressions—requires a μ-abstraction, since they are about introducing a continuation or setting up a binding in Sequent Core. Note how the introduction of μ-abstractions turns the focus on the computation in an expression: the operator of an application or the discriminant of a **case** becomes the term in a command, where the rest becomes a continuation waiting for its result.

3.2 A More Efficient Translation

Unfortunately, while the definitional translation into Sequent Core is straightforward, it is not very practical due to an excessive number of μ-abstractions. For example, the simple application $f\ 1\ 2\ 3$ is translated as

$$S\,[\![((f\ 1)\ 2)\ 3]\!] = \mu\mathsf{ret}.\langle\mu\mathsf{ret}.\langle\mu\mathsf{ret}.\langle f \,\|\, 1 \cdot \mathsf{ret}\rangle \,\|\, 2 \cdot \mathsf{ret}\rangle \,\|\, 3 \cdot \mathsf{ret}\rangle$$

instead of the direct $\mu\mathsf{ret}.\ \langle f \,\|\, 1 \cdot 2 \cdot 3 \cdot \mathsf{ret}\rangle$. These μ-abstractions are analogous to "administrative λ-abstractions" generated by CPS translations, since they both represent some bookkeeping that needs to be cleaned up before we get to the interesting part of a program. Thus, we want the analog of an administrative-free translation [32] for the sequent calculus, S_a, which we achieve by aggressively performing μ reductions during translation as shown in Figure 7 to give a reduced program in Sequent Core.

There is one caveat with the reduced translation into Sequent Core, though. One case during translation has a chance to duplicate the continuation by μ reduction: specifically, with **case**. Naïvely, the μ-reduced translation for **case** would be:

$$\left[\!\!\left[\mathbf{case}\ e\ \mathbf{of}\ \overrightarrow{alt}\right]\!\!\right]\ k = [\![e]\!]\ \ (\mathbf{case\ of}\ \overrightarrow{[\![alt]\!]\ \ k})$$

Because a **case** might have multiple alternatives, each alternative gets its own embedded copy of the given continuation. Simply copying the continuation is not good enough, as many programs can cause chain reactions of duplication, unacceptably inflating the size of the program (see Section 4.2). In practice we need to ensure that the continuation is small enough before duplicating it. Specifically, we force the continuation to be small by shrinking it, which we achieve by introducing extra value bindings for large arguments in a call stack and continuation bindings for the alternatives of a **case**. For example, the (large) call stack $v_1 \cdot v_2 \cdot \mathsf{ret}$ can be shrunk to $x \cdot y \cdot \mathsf{ret}$ along with the bindings $x = v_1$ and $y = v_2$. Additionally, the (large) **case** continuation

$$\mathbf{case\ of}\ K_1(x, y) \to c_1;\, K_2(a, b, z) \to c_2$$

can be shrunk down to

$$\mathbf{case\ of}\ K_1(x, y) \to \mathsf{jump}\ j_1\ x\ y;\, K_2(a, b, z) \to \mathsf{jump}\ j_2\ a\ b\ z$$

along with the bindings $j_1 = \tilde{\mu}[x, y].c_1$ and $j_2 = \tilde{\mu}[a, b, z].c_2$. Thus, when translating a **case** expression, we first shrink the given continuation, set up any bindings that the shrinking process created, and then copy the shrunken continuation in each alternative, as shown in Figure 7.

3.3 Translating Back to Core

But we don't just want to translate one way, we also want the ability to come back to Core. That way, we can compose together both Core and Sequent Core passes by translating to and fro. Since Sequent

Core contains continuations, an obvious way to translate back to Core would be through a CPS translation. However, we want to make sure that a round trip through translations gives us back a similar program to what we originally had. This added round-trip stipulation means that CPS is right out: the program we would get from a CPS translation would be totally different from the one we started with. Even worse, if we were to iterate these round trips, it would compound the problem.

Thus, we look instead for a direct-style translation from Sequent Core to Core, which essentially reverses the translation into Sequent Core. This translation does not rely on types, but it does require properly scoped labels as described in Section 2.2.3. The scope restriction ensures that a Sequent Core program can be directly interpreted as a purely functional Core program without the use of any control effects.

3.4 Round Trips

The question remains: does a round-trip translation yield a similar program? To be more precise, we should expect, as a minimum criterion, that the round-trip translation respects *observational equivalence*: the same behavior in all contexts. We consider two Core expressions observationally equivalent, $e_1 \cong e_2$, whenever $C[e_1]$ terminates if and only if $C[e_2]$ does for all contexts C. Observational equivalence of two Sequent Core terms is similar. The answer is yes: the round-trip translation starting from Core may introduce some bindings and perform some commutative conversions, but the result is still observationally equivalent to where we started. Likewise, all round trips starting from Sequent Core produce observationally equivalent programs: The difference here is that some μ reductions may be performed by the round trip and, unfortunately, all continuation bindings are converted to value bindings.

Proposition 3 (Round trip). *$D\left[\!\left[S_a[\![e]\!]\right]\!\right] \cong e$ and $S_a\left[\!\left[D[\![v]\!]\right]\!\right] \cong v$.*

Also note that both directions of translation are type-preserving, as expected: the outputs of S_a and D are well-typed whenever their inputs are. However, unlike with a CPS transformation, the types are (largely) unchanged. In particular, the type of a Core expression is not changed by translation, which is evident by the fact that S_a doesn't change the types of bindings. Going the other way, D only changes the types of labels and nothing else, which is again evident by the translation of bindings. So the type of Sequent Core terms does not change by translation either.

Proposition 4 (Type preservation). *If $\Gamma \vdash e : \tau$ in Core then $\Gamma \vdash S_a[\![e]\!] : \tau$. If $\Gamma \vdash v : \tau$ in Sequent Core then $\Gamma \vdash D[\![v]\!] : \tau$.*

It is unfortunate that continuation bindings are lost en route during a round-trip translation starting from Sequent Core, as observed above. We had those continuation bindings for a reason, and they should not be erased. Fortunately though, there is a program transformation known as *contification* (described later in Section 5) which can recover the lost continuation bindings (and more) from the soup of value bindings, effectively *re-contifying* them. That means we can move between Core and Sequent Core with wild abandon without losing anything.

4. From Theory to Practice

To find out whether Sequent Core is a practical intermediate language, we implemented a plugin for the Glasgow Haskell Compiler (GHC), a mature, production-quality compiler for Haskell. In this section we reflect what we learned from this experience.

4.1 Sequent Core in GHC

GHC's simplifier is *the* central piece of GHC's optimization pipeline, comprising around 5,000 lines of Haskell code that has been refined over two decades. It applies a large collection of optimizing transformations, including inlining, β-reduction, η-conversion, let floating, case-of-case, case-of-known-constructor, etc.

We implemented a Sequent Core plug-in that can be used with an unmodified GHC. The plug-in inserts itself into the Core-to-Core optimization pipeline and replaces each invocation of GHC's simplifier by the following procedure: convert from Core to Sequent Core; apply the same optimizing transformations as the existing simplifier; and convert back to Core. Here is what we learned:

- Sequent Core is clearly up to the job. In a few months we were able to replicate all of the cleverness that GHC's Simplifier embodies, and our experiments confirm that the performance of the resulting code is essentially identical (see Section A in the appendix for details). Considering the maturity of GHC's existing Simplifier, this is a good result.
- We originally anticipated that Sequent Core could simplify GHC's Simplifier, since the latter accumulates and transforms an explicit continuation (called a $\mathit{SimplCont}$) in the style of a zipper [18], which is closely analogous to Sequent Core's continuations. Thus, we could say that Sequent Core gives a language to the logic that lies in the heart of GHC's Simplifier, providing a more direct representation of GHC optimizations.

  In practice, we found that Sequent Core did not dramatically reduce the lines of code of the Simplifier. The syntax of Sequent Core jumps straight to the interesting action, avoiding the need to accumulate a continuation. However, the Simplifier is complex enough, and requires enough auxiliary information, that the lines of code saved here were a drop in the bucket. The savings were further offset by functions need to traverse the additional syntactic structures of Sequent Core.

So on the practical front, we are not yet ready to abandon Core in favor of Sequent Core in GHC. However, we did find an aspect of optimization for which Sequent Core was qualitatively better than Core: the treatment of join points, to which we turn next.

4.2 Join Points and Case-of-Case

Optimizing compilers commonly push code down into the branches of conditional constructs when possible, bringing **if**s and **case**s to the top level [39, 40]. Besides clarifying the possible code paths, this tends to put intermediate results in their proper context, which enables further optimizations.

GHC performs such code motion aggressively. The most ambitious example is the *case-of-case transform* [30, 38]. Consider:

$$\mathit{half}\ x = \mathbf{if}\ \mathit{even}\ x\ \mathbf{then}\ \mathit{Just}(x\ \text{`}\mathit{div}\text{`}\ 2)\ \mathbf{else}\ \mathit{Nothing}$$

After desugaring and inlining, *half* is written in Core as:

$$\begin{array}{l} \mathit{half} = \lambda x.\ \mathbf{case}\ (\mathbf{case}\ x\ \text{`}\mathit{mod}\text{`}\ 2\ \mathbf{of}\ 0 \to \mathit{True} \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \_ \to \mathit{False})\ \mathbf{of} \\ \qquad\qquad \mathit{True} \ \to \mathit{Just}(x\ \text{`}\mathit{div}\text{`}\ 2) \\ \qquad\qquad \mathit{False} \to \mathit{Nothing} \end{array}$$

Notice how the outer boolean **case** will receive a True or False value, depending on the result of $(x\ \text{`}\mathit{mod}\text{`}\ 2)$. We can make this fact clearer by applying the case-of-case transform, bringing the whole outer **case** inside each branch of the inner **case**:

$$\begin{array}{l} \mathit{half} = \lambda x.\ \mathbf{case}\ x\ \text{`}\mathit{mod}\text{`}\ 2\ \mathbf{of} \\ \qquad\qquad 0 \to \mathbf{case}\ \mathit{True}\ \mathbf{of}\ \mathit{True} \ \to \mathit{Just}(x\ \text{`}\mathit{div}\text{`}\ 2) \\ \qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathit{False} \to \mathit{Nothing} \\ \qquad\qquad \_ \to \mathbf{case}\ \mathit{False}\ \mathbf{of}\ \mathit{True} \ \to \mathit{Just}(x\ \text{`}\mathit{div}\text{`}\ 2) \\ \qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathit{False} \to \mathit{Nothing} \end{array}$$

Happily, case-of-case has revealed an easy simplification, giving:

$$\mathit{half} = \lambda x.\,\mathbf{case}\ x\text{`}\mathit{mod}\text{`}\ 2\ \mathbf{of}\ 0 \to \mathit{Just}(x\ \text{`}\mathit{div}\text{`}\ 2) \\ \_ \to \mathit{Nothing}$$

Of course, this is the ideal outcome, because the outer **case** ultimately vanished entirely. But if case-of-case did not reveal further simplifications, we would have duplicated the outer case, whose alternatives might be of arbitrary size.[6]

Like many compilers, including the original ANF implementation [13], GHC avoids excessive code duplication by abstracting large **case** alternatives into named functions that serve as join points. A typical result looks like this:

$$\begin{aligned} f : \mathit{Int} \to \mathit{Int} = \lambda x.\ &\mathbf{let}\, j{:}\mathit{Maybe\ Int} \to \mathit{Int} = \lambda w.\ \ldots \\ &\mathbf{in}\,\mathbf{case}\ g\ x\ \mathbf{of}\ \mathit{Left}\ y \ \ \to j\ (\mathit{Just}\ y) \\ &\qquad\qquad\qquad\ \mathit{Right}\ \_ \to j\ \mathit{Nothing} \end{aligned}$$

It may appear that we have introduced extra overhead—allocating a closure for j (at the **let**) and calling the function. But a function like j has special properties: it is only tail-called, and it is never captured in a closure. GHC's code generator takes advantage of these properties and compiles the tail call to j into two instructions: adjust the stack pointer and jump. Apart from this special treatment in the code generator, GHC's Core Simplifier does not treat join points specially: they are just local function bindings, subject to the usual optimizations. Collapsing multiple concepts into one can be a strength—but as we see next, it can also be a weakness.

4.3 Losing Join Points

Continuing the example of the previous section, suppose f is called in the following way: $\mathbf{case}\, f\ x\, \mathbf{of}\, 0 \to \mathit{False}; \_ \to \mathit{True}$. If f is inlined at this call site, another case-of-case transformation will occur, and after some further simplifications, we get this:

$$\begin{aligned} &\mathbf{let}\, j{:}\mathit{Maybe\ Int} \to \mathit{Int} = \lambda w.\ \ldots \\ &\mathbf{in}\,\mathbf{case}\ g\ x\ \mathbf{of}\ \mathit{Left}\ y \ \ \to \mathbf{case}\ j\ (\mathit{Just}\ y)\ \mathbf{of}\ 0 \to \mathit{False} \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \_ \to \mathit{True} \\ &\qquad\qquad\qquad\ \mathit{Right}\ \_ \to \mathbf{case}\ j\ \mathit{Nothing}\ \mathbf{of}\ 0 \to \mathit{False} \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \_ \to \mathit{True} \end{aligned}$$

Now j is no longer tail-called and must be compiled as a regular function, with all the overhead entailed. Case-of-case has ruined a perfectly good join point!

This does not happen in Sequent Core. Here is the same function f in Sequent Core:

$$\begin{aligned} &f : \mathit{Int} \to \mathit{Int} = \lambda x.\mu\mathsf{ret}. \\ &\quad \mathbf{let}\, j{:}\mathit{Maybe\ Int} = \tilde{\mu}[w].\ \ldots \mathsf{ret} \ldots \\ &\quad \mathbf{in}\ \langle\, g \,\|\, x \cdot \mathbf{case\,of}\ \mathit{Left}\ y \ \ \to \mathsf{jump}\ j\ \mathit{Just}(y) \\ &\qquad\qquad\qquad\qquad\quad\ \mathit{Right}\ \_ \to \mathsf{jump}\ j\ \mathit{Nothing}\,\rangle \end{aligned}$$

This time, j is represented by a labeled continuation accepting a $\mathit{Maybe\ Int}$. Moreover, observe that the body of j refers to the ret bound the surrounding μ. In Sequent Core, the case-of-case transformation is implemented by a μ reduction, which substitutes a **case** continuation for ret in a computation. For example, inlining f into the command $\langle f \,\|\, x \cdot \mathbf{case\,of}\ 0 \to \mathit{False}; \_ \to \mathit{True}\rangle$ followed by routine Sequent Core simplification instead gives us:

$$\begin{aligned} &\mathbf{let}\, j{:}\mathit{Maybe\ Int} = \tilde{\mu}[w].\ldots \mathbf{case\,of}\ 0 \to \mathit{False}; \_ \to \mathit{True} \ldots \\ &\mathbf{in}\ \langle\, g \,\|\, x \cdot \mathbf{case\,of}\ \mathit{Left}\ y \ \ \to \mathsf{jump}\ j\ \mathit{Just}(y) \\ &\qquad\qquad\qquad\qquad\ \mathit{Right}\ \_ \to \mathsf{jump}\ j\ \mathit{Nothing}\,\rangle \end{aligned}$$

Notice what happened here: just by substituting for ret, we did not push the continuation into the alternatives of the **case**, but instead it naturally flowed into the body of j. Jumps are *stable* in Sequent Core, and the operational semantics of Sequent Core has showed us how to perform case-of-case without ruining any join points!

Could we do the same in Core? Well, yes: the case-of-case transform should (somehow) push the outer **case** into the join point j itself, rather than wrapping it around the *calls* to j, as in

$$\begin{aligned} &\mathbf{let}\, j{:}\mathit{Maybe\,Int}{\to}\mathit{Bool}{=}\lambda w.\ldots \mathbf{case}\, e\, \mathbf{of}\ 0{\to}\mathit{False}; \_{\to}\mathit{True} \ldots \\ &\mathbf{in}\,\mathbf{case}\ g\ x\ \mathbf{of}\ \mathit{Left}\ y \ \ \to j\ (\mathit{Just}\ y) \\ &\qquad\qquad\qquad\ \mathit{Right}\ \_ \to j\ \mathit{Nothing} \end{aligned}$$

where the $\mathbf{case}\, e\, \mathbf{of}\ 0 \to \mathit{False}; \_ \to \mathit{True}$ means to wrap every expression e that returns from j with the **case** analysis. This effect is hard to achieve in Core as it stands, because join points are not distinguished and so the above substitution is not obvious. However, it is natural and straightforward in Sequent Core.

5. Contification

As we saw in Section 3, the translation from Sequent Core to Core is lossy. Sequent Core maintains a distinction between multiple-input continuations (join points) and ordinary functions because they have a different operational and logical reading, but Core only has functions. Converting Core to Sequent Core produces a program with no join points; and even if the Sequent Core Simplifier creates some, they will be lost in the next round trip through Core.

CPS-based compilers often employ a demotion technique called *contification* [20] that turns ordinary functions into continuations. Since direct jumps are faster than function calls, this operation is useful in its own right, but for us it is essential to make the round trip from Sequent Core to Core and back behave like an identity function. So, whenever we translate to Sequent Core, we also perform a simple contification pass that is just thorough enough to find and restore any continuation bindings that could have been lost in translation. In other words, contification picks up what we dropped while going back and forth between the two representations, and hence we are "re-contifying." But we may also discover, and then exploit, join points that happened to be written by the user (Section 5.2).

The mechanics of contification are straightforward. In essence, contification converts function calls (which need to return) into direct jumps (which don't) by baking the calling context into the body of the function. For example, suppose we have this code:

$$\begin{aligned} &\mathbf{let}\, f = \lambda y.\mu\mathsf{ret}.c \\ &\mathbf{in}\ \langle\, g \,\|\, x \cdot \mathbf{case\,of}\ A\ z \to \langle\, f \quad\ \|\, z \cdot \mathsf{ret}\,\rangle \\ &\qquad\qquad\qquad\qquad\ B\ \_ \to \langle\, \mathit{True} \,\|\, \mathsf{ret}\,\rangle \\ &\qquad\qquad\qquad\qquad\ C \quad\ \to \langle\, f \quad\ \|\, \mathit{True} \cdot \mathsf{ret}\,\rangle\rangle \end{aligned}$$

Here f is an ordinary function, bound by the **let** and called in two of the three branches of the **case**. Moreover, both its calls are saturated tail calls, and f is not captured inside a thunk or function closure. Under these circumstances, it is semantics-preserving to replace f with a join point and replace its calls with more efficient direct jumps, thus:

$$\begin{aligned} &\mathbf{let}\, j = \tilde{\mu}[y].c \\ &\mathbf{in}\ \langle\, g \,\|\, x \cdot \mathbf{case\,of}\ A\ z \to \mathsf{jump}\ j\ z \\ &\qquad\qquad\qquad\qquad\ B\ \_ \to \langle \mathit{True} \,\|\, \mathsf{ret}\rangle \\ &\qquad\qquad\qquad\qquad\ C \quad\ \to \mathsf{jump}\ j\ \mathit{True}\,\rangle \end{aligned}$$

This transformation is sound even if the binding of f is recursive, provided all the recursive calls obey the same rules.

We saw in Section 4.2 that GHC already performs a similar analysis during code generation to identify tail calls that can be converted to jumps. However, this conversion happens just *once* and only *after* all Core-to-Core optimizations are finished. Here, we bring contification forward as a pass that can happen in the midst of the main optimization loop, giving a language to talk about join points for other optimizations to exploit.

[6] We see the same code duplication issue arise during translation in Section 3; we avoid duplication with the same solution in both instances.

5.1 Analysis and Transformation

We divide the contification algorithm into two phases, an *analysis phase* and a *transformation phase*. The analysis phase finds functions that can be contified; then the transformation phase carries out the necessary rewrites in one sweep. See Section B in the appendix for a more detailed description of the algorithm.

In the analysis phase, we are interested in answering the all-important question: given a $\mathbf{let}$-bound function f (a potential join point), is every call to f a saturated tail call? Sequent Core lets us state this condition precisely: all its calls must be of the form $\langle f \parallel \vec{v} \cdot \mathsf{ret}\rangle$, *where the* ret *is the same* ret *that is in scope at f's binding site*. This is another occasion on which it is helpful to think of ret as a lexically-scoped variable bound by $\mu\mathsf{ret}$.

To answer the question, the algorithm for the analysis phase gathers data bottom-up, similar to a free-variable analysis, and marks which $\mathbf{let}$-bound functions may be replaced with labeled continuations. During traversal, the analysis determines the full set of variables that actually appear free in an expression, which we call the *free set*, as well as the subset of those variables that only appear as tail calls, which we call the *good set*. Three basic rules govern the upward propagation of these sets of variables:

- The head variable of a tail call—that is, any f in a command of the form $\langle f \parallel \vec{v} \cdot \mathsf{ret}\rangle$—is good so long as it is not free in the arguments.
- Terms cannot have any good variables, since labels cannot appear free in a term. The exception to this rule is a bound function that will be contified, since of course it won't be a term anymore; thus contification has a cascading effect.
- When considering other forms of expressions with several subexpressions, a variable is good on the whole if it is good in at least one subexpression and bad in none of them.

If a $\mathbf{let}$-bound variable is considered good after analyzing its entire scope, then we mark it for contification. For a non-recursive binding $\mathbf{let}\, f = v \,\mathbf{in}\, c$, the scope is just the body of the $\mathbf{let}$, c. For a set of mutually recursive bindings $\mathbf{let\,rec}\left\{\overrightarrow{f = \lambda\vec{x}.\mu\mathsf{ret}.c}\right\}\mathbf{in}\, c'$, the scope includes all the function bodies $\vec{c}$ as well as c'. Note that we can't contify only part of a recursive set; it's all or none. The reason for this restriction is that if we only contified some of the functions in a recursive binding, then those newly labeled continuations would be out of scope for the uncontified functions left behind, thus breaking the mutual recursion.

Once the analysis is complete, the transformation itself is straightforward. Note that we assume that all functions begin with a series of lambdas and then one μ; this can always be arranged, since v and $\mu\mathsf{ret}.\langle v \parallel \mathsf{ret}\rangle$ are equivalent by η-conversion. At each binding $f = \lambda\vec{x}.\mu\mathsf{ret}.c$ marked for contification, we pick a fresh label j and rewrite the binding as $j = \tilde{\mu}[\vec{x}].c$. Then, at every call site of a contified f, we rewrite $\langle f \parallel \vec{v} \cdot \mathsf{ret}\rangle$ as $\mathsf{jump}\; j\; \vec{v}$.

This algorithm is similar to Kennedy's [20], except that we only contify functions called with the ret continuation rather than any single continuation. Clearly, this algorithm impacts fewer functions than the more general one. We also implemented a more comprehensive (and thus more expensive) contification pass for GHC, only to find it offered little impact on performance. Thus, the simpler algorithm seems to lie in a sweet spot between ease of implementation, cost of execution, and recovery of previously identified join points.

5.2 Discovering Join Points

By contifying aggressively enough, we may be able to discover—and exploit—additional join points that happen to arise naturally in the program. Consider the standard library function *find* that returns the first value in a list satisfying a predicate, or $\mathit{Nothing}$ if none satisfy it:

$$
\begin{array}{l}
\mathit{find} : (a \to \mathit{Bool}) \to [a] \to \mathit{Maybe}\; a \\
\mathit{find} = \lambda p.\lambda xs.\,\mathbf{let}\; go = \lambda ys.\,\mathbf{case}\; ys\; \mathbf{of} \\
\qquad\qquad\qquad\qquad\qquad [\,] \to \mathit{Nothing} \\
\qquad\qquad\qquad\qquad\qquad (y : ys') \to \mathbf{case}\; p\; y\; \mathbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{True} \to \mathit{Just}\; y \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{False} \to go\, ys' \\
\qquad\qquad \mathbf{in}\; go\; xs
\end{array}
$$

Here *go* is a join point, and our contification analysis can discover that fact.[7] Now suppose that *find* is called as:

$$
\begin{array}{l}
f = \lambda xs.\,\mathbf{case}\; \mathit{find}\; \mathit{even}\; xs\; \mathbf{of}\; \mathit{Nothing} \to 0 \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{Just}\; x \to x + 1
\end{array}
$$

In GHC today, after inlining and performing some routine simplifications, we get

$$
\begin{array}{l}
f = \lambda xs.\,\mathbf{let}\; go = \lambda ys.\,\mathbf{case}\; ys\; \mathbf{of} \\
\qquad\qquad\qquad\qquad\qquad [\,] \to \mathit{Nothing} \\
\qquad\qquad\qquad\qquad\qquad (y : ys') \to \mathbf{case}\; y\; `\mathit{mod}`\; 2\; \mathbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad 0 \to \mathit{Just}\; y \\
\qquad\qquad\qquad\qquad\qquad\qquad \_ \to go\; ys' \\
\qquad\quad \mathbf{in}\,\mathbf{case}\; go\; xs\; \mathbf{of}\; \mathit{Nothing} \to 0 \\
\qquad\qquad\qquad\qquad\qquad \mathit{Just}\; x \to x + 1
\end{array}
$$

And now *go* is no longer a join point, another example of the phenomenon we previously saw in Section 4.3. In Core, this is as far as we can go. In Sequent Core, however, *go* can be contified, and after the dust settles, we end up with the analog of:

$$
\begin{array}{l}
f = \lambda xs.\,\mathbf{let}\; go = \lambda ys.\,\mathbf{case}\; ys\; \mathbf{of} \\
\qquad\qquad\qquad\qquad\qquad [\,] \to 0 \\
\qquad\qquad\qquad\qquad\qquad (y : ys') \to \mathbf{case}\; y\; `\mathit{mod}`\; 2\; \mathbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad 0 \to y + 1 \\
\qquad\qquad\qquad\qquad\qquad\qquad \_ \to go\; ys' \\
\qquad\quad \mathbf{in}\; go\; xs
\end{array}
$$

This code is much better: the allocation of the function closure for *go* and the intermediate value $(\mathit{Just}\; y)$ are both avoided, the outer **case** disappears, and the recursive call in *go* turns into a jump. In Sequent Core, once a join point is discovered by contification, it *stays* a join point.

5.3 Caveats

Since idiomatic Haskell code uses curried functions, we must be careful about what we consider a tail call. Note the rewrites above assumed the number of arguments in the tail calls matched the number of outer lambdas in the function. Thus, for our purposes, all tail calls must also be *exact*—each must provide exactly as many arguments as there are outer lambdas in the function definition. This restriction can be lifted so long as the calls are *consistent* in the number of arguments passed, also known as the *call arity* [7]. However, it is unclear that this is useful often enough to warrant the additional complexity. For the purpose of re-contification after round-trip translations, all relevant tail calls will already be exact.

Thus far, we have neglected to mention the impact of polymorphism on contification. Polymorphic functions are no problem, with one proviso: a function cannot be contified if it is polymorphic in its return type. To see why, suppose we have some function $f : \forall a{:}\star.\mathit{Int} \to a \to a$ and we generate the continuation:

$$
j = \tilde{\mu}[a{:}\star, n{:}\mathit{Int}, x{:}a].\,\langle f \parallel a \cdot n \cdot x \cdot \mathsf{ret}\rangle
$$

[7] Note that *go* is recursive, but recursive join points are fine as long as they are properly tail-called.

We have a problem: the continuation here cannot be well-typed, because ret is a free variable that comes from the outside, so ret cannot by typed by *this* a since a was not in scope when ret was bound. Note that this is not an issue particular to Sequent Core; the same phenomenon arises in typed CPS languages. Contification *in a typed setting* must always be careful here.

Like arity, this is not a fatal issue. Just as we *can* call f with True by passing *Bool* as the type argument, we can contify f by fixing the appropriate return type in context. For example, if the ret in scope has type *Bool*, then the following is well-typed:

$$j = \tilde{\mu}[n{:}\mathit{Int}, x{:}\mathit{Bool}].\, \langle f \parallel \mathit{Bool} \cdot n \cdot x \cdot \mathsf{ret}\rangle$$

However, this situation appears to be vanishingly rare in practice. And in the case of re-contification, Sequent Core join points never have a polymorphic return type when translated to Core.[8] Thus, while correctness demands that we at least *check* for polymorphic return types, re-contification can simply give up on them.

6. Related Work

6.1 Relation to Sequent Calculi

We might ask how and why Sequent Core differs from similar computational interpretations of the sequent calculus, like the $\overline{\lambda}\mu\tilde{\mu}$-calculus [9] or System L [26] for example. A primary difference is that in lieu of $\mathbf{let}$ bindings, these previous languages have continuations of the form $\tilde{\mu}x.c$, as mentioned in Section 2.2.2. As it turns out, they are not needed in Sequent Core. Indeed, a common reading for $\tilde{\mu}x.c$ at the top of a command is:

$$\langle v \parallel \tilde{\mu}x.c\rangle = (\mathbf{let}\, x = v\, \mathbf{in}\, c)$$

where the $\tilde{\mu}$ is replaced with an explicit substitution via $\mathbf{let}$. Then, using the call-by-name semantics for Sequent Core, all $\tilde{\mu}$-abstractions can be *lifted*[9] to the top of a command, so every $\tilde{\mu}$-abstraction can be written as a $\mathbf{let}$. Under a call-by-value semantics, $\tilde{\mu}$-abstractions play a more crucial role as noted by Curien and Herbelin [9], but in that case they are exactly the same as a default $\mathbf{case}$ in Sequent Core: $\tilde{\mu}x.c = \mathbf{case\,of}\; x \to c$. So again, the extra continuation is not needed. Considering the fact that the recursion so elegantly expressed by a $\mathbf{let}$ cannot be represented with $\tilde{\mu}$-abstractions alone gives $\mathbf{let}$ its primary role in Sequent Core.

The other differences between Sequent Core and previous sequent calculi are the labeled, multiple-input continuations and jumps. The exact formulations of these constructs were designed with the needs of a practical compiler in mind, but they do have a more theoretical reading as well. In particular, we could (and indeed at one point we did) consider adding general existential types to the language. That way a multiple-input continuation $\tilde{\mu}[\overrightarrow{a}, \overrightarrow{x}].c$ might be interpreted as just a $\mathbf{case}$ continuation $\mathbf{case\,of}\;(\overrightarrow{a}, \overrightarrow{x}) \to c$, so that labels just refer to single-input continuations and jumps are just cut pairs $\langle(\overrightarrow{\sigma}, \overrightarrow{v}) \parallel j\rangle$. However, for this to represent the correct operational cost at run time, it is crucial that these existential tuples are *unboxed* [29], meaning that they are values of a truly positive type [26] unlike the normally boxed Haskell data types. Additionally, unboxed (existential) tuples give less restraint for labels and jumps than the syntactic limitations implicitly imposed in Figure 1. The result is an unfortunately heavy-handed encoding unless stricter measures are taken for these positive types, as in Zeilberger's [46] interpretation of focalization.

[8] Consider a term containing a local join point with no intervening μ:

$$\mu\mathsf{ret}.\;\ldots\,\mathbf{let}\, j{:}\exists\overrightarrow{a{:}\kappa}.(\overrightarrow{\tau}) = \tilde{\mu}[\overrightarrow{a{:}\kappa}, \overrightarrow{x{:}\tau}].c\,\mathbf{in}\ldots$$

Assuming the term has type σ, the μ-bound ret will also have type σ. After translation, j's type becomes $\forall\overrightarrow{a{:}\kappa}.\overrightarrow{\tau} \to \sigma$, where $\overrightarrow{a}$ cannot occur free in the return type σ due to the typical hygiene requirements of translation.

[9] This lifting can be done by the ς reductions of [44] and [25].

6.2 CPS as an Intermediate Language

Though continuations had been actively researched for nearly a decade [34], the first use of CPS for compilation appeared in 1978, in the Rabbit compiler for Scheme [40]. Steele was interested in the connection between Scheme and the λ-calculus [43], and CPS was a way to "elucidate some important compilation issues," such as evaluation order and intermediate results, while maintaining that connection. He also noted the ease of translation to an imperative machine language. Standard ML of New Jersey [1] is another prominent example; it even performs such low-level tasks as register assignment within the CPS language itself.

So what's stopping GHC from just adopting CPS? One answer is "but *which* CPS?" Usually the "CPS" intermediate language refers to "the language produced by the call-by-value CPS transform." Surely we would not use *this* "CPS" to compile a non-strict language like Haskell. Of course, there are call-by-name [17, 32] and call-by-need [27] CPS transforms, but (to our knowledge) they have not been used in compilers before, leaving us in unknown territory.

More importantly, the effect of *any* CPS transform is to fix an evaluation order in the syntax of the program, but GHC routinely exploits the ability to reorder calculations, like shifting between call-by-need and call-by-value, which gives more flexibility for optimizing a pure, lazy language like Haskell [30].

Hence an advantage of a sequent calculus for GHC: like the λ-calculus, *the syntax does not fix a specific evaluation order*. For example, we illustrated both call-by-name (Figure 2) and call-by-need (Figure 3) readings for the same Sequent Core programs, and call-by-value would be valid, too. So we can still reason about Haskell programs with call-by-name semantics while implementing them efficiently with call-by-need.

There is one more advantage shared by both Core and Sequent Core, but not CPS, which is critically important for GHC. Specifically, GHC allows for arbitrary rewrite rules that transform function calls [31], which enable user optimizations like stream fusion [8]. Both Core and Sequent Core make expressing and implementing these custom rules easy, since both languages make nested function call structure in expressions like $\mathit{map}\; f\; (\mathit{map}\; g\; \mathit{xs})$ apparent: either as a chain of applications or a call stack. Instead, CPS represents nested functional calls in the source *abstractly* as in:

$$\lambda k.\mathit{map}\; g\; (\lambda h.h\; \mathit{xs}\; (\lambda \mathit{ys}.\mathit{map}\; f\; (\lambda h'.h'\; \mathit{ys}\; k)))$$

To understand the original call structure of the program requires chasing information through several indirections. Instead, Sequent Core represents function calls *structurally* as stacks that can be immediately inspected, so bringing continuations to GHC without getting in the way of what GHC already does. In this light, we can view the sequent calculus as a "strategically defunctionalized" [36] CPS language. There is an essential trade-off in the expressive capabilities of rigid structure versus free abstraction [35], and the additional structure provided by call stacks enables more optimizations, like rewrite rules, by making continuations scrutable.

6.3 ANF as an Intermediate Language

In 1992, Sabry and Felleisen [37] demonstrated that the actions of a CPS compiler could be understood in terms of the original source code. Hence, though a CPS transform could express call-by-value semantics for λ-terms, that same semantics could be expressed as reductions in the original term. The new semantics extended Plotkin's call-by-value λ-calculus [32] with more reductions and hence further possible optimizations. Flanagan *et. al.* [13] argued that the new semantics obviated the need for CPS in the compiler, since the same work would be performed by systematically applying the new reductions, putting the source terms into *administrative normal form* (ANF). Representations in ANF became popular in the following years, as its ease of implementation provides an obvious benefit over CPS, but its costs took some time to appreciate in practical use. In 2007, Kennedy [20] outlined some of these issues, encouraging compiler writers to "give CPS a second chance."

It is worthwhile to point out that Sequent Core does not suffer from the same difficulties as ANF described by Kennedy. Sequent Core does not require renormalization after routine transformations: the syntax is closed under reductions like inlining. Furthermore, the various *commutative conversions* in these direct-style representations—such as case-of-case discussed in Section 4.2—are uniformly represented through μ-abstraction and μ-reduction in Sequent Core [26], which is a strength shared with CPS. Likewise, labeled continuations in Sequent Core serve an analogous purpose to labeled continuations in CPS, which preserve code sharing during these commutative conversions. Therefore, we can say that the sequent calculus can do everything CPS can, so Sequent Core retains the advantages of continuations laid out by Kennedy.

6.4 Other Representations

Our focus has been on functional programming, but of course compilation in the imperative world has long been dominated by the *static single-assignment form* (SSA) [10]. While it is known that SSA can be converted to CPS [19], the flat structure of SSA may be more convenient for representing imperative programs wherein block layout is more natural and higher-order features are not used.

We don't have to choose between flat and higher-order, however. Thorin [23] is a graph-based representation aiming to support both imperative and functional code by combining a flat structure for ease of code transformation and first-class closures for implementing higher-order languages. However, Thorin is still intended for use in strict languages with pervasive side effects; it remains to be seen whether such a representation could be adapted for high-level optimizations in a non-strict regime such as Haskell.

7. Reflections on Intermediate Languages

There are many different goals we have for an intermediate language in an optimizing compiler, some of which seem at odds with one another. In a perfect world, an intermediate representation would, among other things:

1. Have a simple grammar, which makes it easy to traverse and transform programs written in the language. For example, if the grammar is represented as data types in a functional language, there should be a minimal number of (mutually) recursive data types that represent the main workhorse of runtime behavior.
2. Have a simple operational meaning, which makes it easy to analyze and understand the performance and behavior of programs. For example, it should be easy to find the "next" step of the program from the top of the syntax tree, and language constructs should be easy to compile to machine code.
3. Be as flexible in evaluation order as the source language permits, to permit as many transformations and out-of-order reductions as possible during optimization.
4. Make it easy to express control flow and shared join points, to reduce code size without hurting performance.
5. Make it easy to apply arbitrary rewrite rules expressed in the source language, especially for curried function applications when they appear pervasively in the source language.

We summarize the trade-offs for different representation styles, using $+$ for "good", "$-$" for "not good", and blank for neutral, in Figure 8. Is there a way to get the best of all worlds? Perhaps, just as Sabry and Felleisen showed that you can get the advantages of CPS by using direct-style ANF, we would be able to get the advantages of a sequent calculus in a direct-style variant of Core.

	Core	Sequent Core	CPS
Simple grammar	$+$		
Operational reading	$+$	$++$	$++$
Flexible eval order	$+$	$+$	$-$
Control flow	$-$	$++$	$++$
Rewrite rules	$+$	$+$	$-$

Figure 8. Advantages of different representation styles

In particular, the killer advantage of Sequent Core has turned out to be its treatment of join points, which are different from both functions and continuations, and the more powerful case-of-case transformations that they support (Section 4.2). Informed by this experience, we speculate that it should be possible to augment Core with explicit join points, rather than treat them as ordinary bindings the way GHC does now. We are actively exploring this line of work, using Sequent Core as our model that gives us a foundation for the theory and design of a direct-style λ-calculus with join points. Such a λ-calculus would sit between Core and Sequent Core by having both a simple syntactic structure while also preserving the extra information about control flow. Thus, Sequent Core can currently be seen as a laboratory for compiler intermediate representations.

In developing Sequent Core, we had a love/hate relationship with purity—specifically, with the absence of control effects. On the one hand, keeping Sequent Core "pure" lets us easily leverage the existing technology for compiling the λ-calculus efficiently. The restrictions on continuation variables and jumps create the direct-style correspondence between the two, enabling the same techniques for simplification and call-by-need evaluation (in contrast to [5, 6]). On the other hand, the sequent calculus gives rise to a language of first-class control effects in its natural state, equivalent to adding callcc to the λ-calculus. Thus, the *classical* sequent calculus is more expressive [11], and lets us collapse otherwise distinct concepts—like control flow and data flow, or functions and data structures—into symmetrical dual pairs. Here, we chose to restrain Sequent Core and maintain the connection with Core. 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.

Looking back to the table of trade-offs, we see that Sequent Core strikes a middle ground between Core and CPS. Beside the point about simple grammar—for which it is hard to improve upon the elegance of the λ-calculus—Sequent Core manages to combine the advantages of both direct and continuation-passing styles. Clearly, the focus of our comparison was between Core and Sequent Core, for which we conclude that the sequent calculus shows us how to bring control flow and continuation-passing-style optimizations to GHC without getting in the way of what GHC already does well. But this is a two-way road: the sequent calculus can also teach us how to bring flexibility and direct-style optimizations, like rewrite rules, to CPS compilers by bringing the *structures* underlying continuations out of the *abstractions*. We chalk this up as another in a long line of wins for the Curry-Howard isomorphism: in the debate between direct and continuation-passing style compilers, the logic tells us how we might have our cake and eat it too.

Acknowledgements

We would like to thank Iavor S. Diatchki for his help and input on the design and implementation of Sequent Core, and to thank Matthias Felleisen and Olin Shivers for discussions on the merits of continuations passing style. Paul Downen and Zena M. Ariola have been supported by NSF grant CCF-1423617.

References

[1] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, NY, USA, 1992. ISBN 0-521-41695-7.

[2] Z. M. Ariola and H. Herbelin. Control reduction theories: The benefit of structural substitution. *Journal of Functional Programming*, 18(3):373–419, May 2008. ISSN 0956-7968. doi: 10.1017/S0956796807006612. URL `http://dx.doi.org/10.1017/S0956796807006612`.

[3] Z. M. Ariola, J. Maraist, M. Odersky, M. Felleisen, and P. Wadler. A call-by-need lambda calculus. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 233–246, New York, NY, USA, 1995. ACM. ISBN 0-89791-692-1. doi: 10.1145/199448.199507. URL `http://doi.acm.org/10.1145/199448.199507`.

[4] Z. M. Ariola, A. Bohannon, and A. Sabry. Sequent calculi and abstract machines. *ACM Transactions on Programming Languages and Systems*, 31(4):13:1–13:48, May 2009. ISSN 0164-0925. doi: 10.1145/1516507.1516508. URL `http://doi.acm.org/10.1145/1516507.1516508`.

[5] Z. M. Ariola, H. Herbelin, and A. Saurin. Classical call-by-need and duality. In *Proceedings of the 10th International Conference on Typed Lambda Calculi and Applications*, TLCA'11, pages 27–44, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-21690-9. URL `http://dl.acm.org/citation.cfm?id=2021953.2021961`.

[6] Z. M. Ariola, P. Downen, H. Herbelin, K. Nakata, and A. Saurin. Classical call-by-need sequent calculi: The unity of semantic artifacts. In *Proceedings of the 11th International Conference on Functional and Logic Programming*, FLOPS'12, pages 32–46, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-29821-9. doi: 10.1007/978-3-642-29822-6_6. URL `http://dx.doi.org/10.1007/978-3-642-29822-6_6`.

[7] J. Breitner. Call arity. In *Trends in Functional Programming - 15th International Symposium, TFP 2014, Soesterberg, The Netherlands, May 26-28, 2014. Revised Selected Papers*, pages 34–50, 2014. doi: 10.1007/978-3-319-14675-1_3. URL `http://dx.doi.org/10.1007/978-3-319-14675-1_3`.

[8] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 315–326, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-815-2. doi: 10.1145/1291151.1291199. URL `http://doi.acm.org/10.1145/1291151.1291199`.

[9] P. Curien and H. Herbelin. The duality of computation. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, pages 233–243, 2000. doi: 10.1145/351240.351262. URL `http://doi.acm.org/10.1145/351240.351262`.

[10] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991. doi: 10.1145/115372.115320. URL `http://doi.acm.org/10.1145/115372.115320`.

[11] M. Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17(1-3):35–75, Dec. 1991. ISSN 0167-6423. doi: 10.1016/0167-6423(91)90036-W. URL `http://dx.doi.org/10.1016/0167-6423(91)90036-W`.

[12] M. Felleisen and D. P. Friedman. Control operators, the SECD-machine, and the λ-calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–219. North Holland Press, Amsterdam, 1986.

[13] 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, pages 237–247, New York, NY, USA, 1993. ACM. ISBN 0-89791-598-4. doi: 10.1145/155090.155113. URL `http://doi.acm.org/10.1145/155090.155113`.

[14] G. Gentzen. Investigations into logical deduction. In M. Szabo, editor, *Collected papers of Gerhard Gentzen*, pages 68–131. North-Holland, 1969.

[15] H. Herbelin. A lambda-calculus structure isomorphic to gentzen-style sequent calculus structure. In *Computer Science Logic, 8th International Workshop, CSL '94, Kazimierz, Poland, September 25-30, 1994, Selected Papers*, pages 61–75, 1994. doi: 10.1007/BFb0022247. URL `http://dx.doi.org/10.1007/BFb0022247`.

[16] H. Herbelin. Explicit substitutions and reducibility. *Journal of Logic and Computation*, 11(3):431–451, 2001. doi: 10.1093/logcom/11.3.431. URL `http://logcom.oxfordjournals.org/content/11/3/431.abstract`.

[17] M. Hofmann and T. Streicher. Continuation models are universal for lambda-mu-calculus. In *Proceedings, 12th Annual IEEE Symposium on Logic in Computer Science, Warsaw, Poland, June 29 - July 2, 1997*, pages 387–395, 1997. doi: 10.1109/LICS.1997.614964. URL `http://dx.doi.org/10.1109/LICS.1997.614964`.

[18] G. Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, Sept. 1997. ISSN 0956-7968. doi: 10.1017/S0956796897002864. URL `http://dx.doi.org/10.1017/S0956796897002864`.

[19] R. A. Kelsey. A correspondence between continuation passing style and static single assignment form. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*, IR '95, pages 13–22, New York, NY, USA, 1995. ACM. ISBN 0-89791-754-5. doi: 10.1145/202529.202532. URL `http://doi.acm.org/10.1145/202529.202532`.

[20] A. Kennedy. Compiling with continuations, continued. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 177–190, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-815-2. doi: 10.1145/1291151.1291179. URL `http://doi.acm.org/10.1145/1291151.1291179`.

[21] D. A. Kranz, R. Kelsey, J. Rees, P. Hudak, and J. Philbin. ORBIT: an optimizing compiler for Scheme. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction, Palo Alto, California, USA, June 25-27, 1986*, pages 219–233, 1986. doi: 10.1145/12276.13333. URL `http://doi.acm.org/10.1145/12276.13333`.

[22] J. Launchbury. A natural semantics for lazy evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 144–154, New York, NY, USA, 1993. ACM. ISBN 0-89791-560-7. doi: 10.1145/158511.158618. URL `http://doi.acm.org/10.1145/158511.158618`.

[23] 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, pages 202–212, Washington, DC, USA, 2015. IEEE Computer Society. ISBN 978-1-4799-8161-8. URL `http://dl.acm.org/citation.cfm?id=2738600.2738626`.

[24] A. Meyer and S. Cosmadakis. Semantical Paradigms: Notes for an Invited Lecture. Technical report, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, July 1988.

[25] G. Munch-Maccagnoni. Focalisation and classical realisability. In *Proceedings of the 23rd CSL International Conference and 18th EACSL Annual Conference on Computer Science Logic*, CSL'09/EACSL'09, pages 409–423, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 3-642-04026-8, 978-3-642-04026-9. URL `http://dl.acm.org/citation.cfm?id=1807662.1807695`.

[26] G. Munch-Maccagnoni. *Syntax and Models of a non-Associative Composition of Programs and Proofs*. PhD thesis, Univ. Paris Diderot, 2013.

[27] C. Okasaki, P. Lee, and D. Tarditi. Call-by-need and continuation-passing style. *Lisp and Symbolic Computation*, 7(1):57–82, Jan. 1994. ISSN 0892-4635. doi: 10.1007/BF01019945. URL `http://dx.doi.org/10.1007/BF01019945`.

[28] M. Parigot. Lambda-my-calculus: An algorithmic interpretation of classical natural deduction. In *Proceedings of the International Conference on Logic Programming and Automated Reasoning*, LPAR '92, pages 190–201, London, UK, UK, 1992. Springer-Verlag. ISBN 3-540-55727-X. URL `http://dl.acm.org/citation.cfm?id=645706.663989`.

[29] S. L. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Proceedings*

of the 5th ACM Conference on Functional Programming Languages and Computer Architecture, pages 636–666, London, UK, UK, 1991. Springer-Verlag. ISBN 3-540-54396-1. URL `http://dl.acm.org/citation.cfm?id=645420.652528`.

[30] S. L. Peyton Jones and A. L. M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1-3): 3–47, Sept. 1998. ISSN 0167-6423. doi: 10.1016/S0167-6423(97)00029-4. URL `http://dx.doi.org/10.1016/S0167-6423(97)00029-4`.

[31] S. L. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *2001 Haskell Workshop*. ACM SIGPLAN, September 2001. URL `http://research.microsoft.com/apps/pubs/default.aspx?id=74064`.

[32] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975. doi: 10.1016/0304-3975(75)90017-1.

[33] J.-C. Raoult and J. Vuillemin. Operational and semantic equivalence between recursive programs. *Journal of the Association for Computing Machinery*, 27(4):772–796, Oct. 1980. ISSN 0004-5411. doi: 10.1145/322217.322229. URL `http://doi.acm.org/10.1145/322217.322229`.

[34] J. C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3-4):233–248, 1993.

[35] J. C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-oriented Programming*, pages 13–23. MIT Press, Cambridge, MA, USA, 1994. ISBN 0-262-07155-X. URL `http://dl.acm.org/citation.cfm?id=186677.186680`.

[36] J. C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, Dec. 1998. ISSN 1388-3690. doi: 10.1023/A:1010027404223. URL `http://dx.doi.org/10.1023/A:1010027404223`.

[37] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. In *LISP and Functional Programming*, pages 288–298, 1992. doi: 10.1145/141471.141563. URL `http://doi.acm.org/10.1145/141471.141563`.

[38] A. L. M. Santos. *Compilation by Transformation in Non-Strict Functional Languages*. PhD thesis, University of Glasgow, 1995.

[39] T. Standish, D. Harriman, D. Kibler, and J. Neighbors. *The Irvine program transformation catalogue*. 1976.

[40] G. L. Steele, Jr. RABBIT: A compiler for SCHEME. Technical Report AITR-474, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1978.

[41] G. L. Steele, Jr. and G. J. Sussman. Lambda: The ultimate declarative. Memo AIM-379, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1976.

[42] G. L. Steele, Jr. and G. J. Sussman. Lambda: The ultimate imperative. Memo AIM-353, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1976.

[43] G. J. Sussman and G. L. Steele, Jr. SCHEME: An interpreter for untyped lambda-calculus. Memo AIM-349, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1975.

[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. URL `http://doi.acm.org/10.1145/944705.944723`.

[45] P. Wadler. Propositions as types. *Communications of the ACM*, 58(12): 75–84, Nov. 2015. ISSN 0001-0782. doi: 10.1145/2699407. URL `http://doi.acm.org/10.1145/2699407`.

[46] N. Zeilberger. *The Logical Basis of Evaluation Order and Pattern-Matching*. PhD thesis, Carnegie Mellon University, 2009.

A. Benchmarks

Tables 1 and 2 show the results of the `spectral` and `real` NoFib tests for GHC 7.8.4 modified to use the new Sequent Core version of the simplifier, versus the baseline GHC 7.8.4. There are wins and losses; the losses are relatively few but serious (most notably `spectral/rewrite` and `real/cacheprof`).

It is difficult to glean much from the details, largely because rewriting the simplifier with a new intermediate representation is such a drastic change. We hope to use the Sequent Core experience to make more modest changes to the original simplifier, for which it should be easier to tease out the effects of particular changes.

B. Contification Algorithm

The algorithm A is shown in Figure 9. At each command or continuation, the traversal produces a triple $(\mathcal{F}, \mathcal{G}, \mathcal{C})$ of a *free set* $\mathcal{F}$, a *good set* $\mathcal{G}$, and a *contifiable set* $\mathcal{C}$, with $\mathcal{G} \subseteq \mathcal{F}$ and $\mathcal{C} \cap \mathcal{F} = \emptyset$. The free set contains the variables occurring free in the command; the good set contains just the "good" ones, that is, those variables that only occur free as tail-called functions; and the contifiable set contains the functions marked for contification. We assume here that all binders are distinct.[10] For terms, the procedure is the same, except that only $\mathcal{F}$ and $\mathcal{C}$ are returned—since terms are continuation-closed, *no* function occurring free in a term can be contified, so the good set for a term is always empty.

At each binding $\mathbf{let}\, f = v\, \mathbf{in}\, c$, if $A\, [\![c]\!] = (\mathcal{F}, \mathcal{G}, \mathcal{C})$, we contify f (that is, add it to $\mathcal{C}$) if and only if $f \in \mathcal{G}$. For recursive bindings, the procedure is the same, only of course the combined analysis for the body and the definitions must be used.

The definition of $\oplus$ says that, in an expression with two subexpressions, the good variables are those that are

- good on the left and absent on the right, or
- good on the right and absent on the left, or
- good on both sides.

Alternatively, we could track the free set and the *bad* set, and then $\oplus$ would simply take the unions. Using the good set makes the algoritm more flexible, however; many extensions require tracking something about the calls to each function, such as the arity, which is easy if the good set is represented as the domain of a finite map.

C. Proof of Correspondence (Proposition 1)

The proof is by bisimulation. After establishing some reduction relations and their algebraic properties, we will define a readback function, use it to define a bisimulation, then prove that the bisimulation preserves termination.

C.1 Reduction

We define $\to$ as the compatible closure of $\mapsto$ along with one additional rule. Note that while $\mapsto$ only relates commands, $\to$ extends to terms and continuations as well.

The new rule is a form of η-rule:

$$\mu\mathsf{ret}.\langle v \,\|\, \mathsf{ret}\rangle \to v \qquad (\mu_\eta)$$

Applying the μ_η-rule does not affect observable behavior, but it will be necessary for relating the two calculi. Note that it is never a standard reduction (unless it happens to coincide with a standard μ-reduction).

We now call $\mapsto$ *standard reduction*. Accordingly, $\to$ includes *non-standard reduction*; to denote non-standard reduction specifically, we write $\rightarrowtail$. Hence $\to$ is the disjoint union of $\mapsto$ and $\rightarrowtail$.

[10] The actual code annotates the binders rather than gathering a set, so it avoids making this assumption.

Test	Size	Allocs	Time	Elapsed	Memory
ansi	-0.0%	-11.8%	0.000	0.000	0.0%
atom	-0.0%	0.0%	+0.9%	+0.9%	0.0%
awards	0.0%	0.0%	0.000	0.000	0.0%
banner	0.0%	0.0%	0.000	0.000	0.0%
boyer	+0.0%	0.0%	0.020	0.020	0.0%
boyer2	-0.3%	+5.2%	0.000	0.000	0.0%
calendar	-0.0%	-0.6%	0.000	0.000	0.0%
cichelli	+0.2%	+2.3%	0.040	0.040	0.0%
circsim	+0.0%	-0.1%	-5.5%	-5.5%	+5.9%
clausify	+0.0%	0.0%	0.020	0.020	0.0%
comp_lab_zift	-0.0%	+0.1%	0.100	0.100	+14.3%
constraints	-0.0%	-2.9%	-6.3%	-6.2%	0.0%
cryptarithm1	0.0%	0.0%	-0.7%	-0.7%	0.0%
cryptarithm2	-0.2%	+0.0%	0.010	0.010	0.0%
cse	-0.1%	-1.0%	0.000	0.000	0.0%
eliza	-0.0%	-1.8%	0.000	0.000	0.0%
event	-0.0%	-2.2%	0.074	0.074	0.0%
expert	+0.0%	-0.6%	0.000	0.000	0.0%
fft	+0.1%	+1.3%	0.020	0.020	-10.0%
fft2	-0.0%	+0.1%	0.030	0.030	0.0%
fibheaps	0.0%	0.0%	0.020	0.020	0.0%
fish	0.0%	0.0%	0.010	0.010	0.0%
gcd	-0.0%	0.0%	0.016	0.016	0.0%
genfft	-0.0%	-0.0%	0.020	0.020	0.0%
ida	+0.0%	+1.0%	0.050	0.050	0.0%
integer	+0.0%	0.0%	-0.7%	-0.9%	0.0%
knights	0.0%	-0.0%	0.000	0.000	0.0%
lcss	-0.0%	-0.0%	-2.6%	-2.6%	0.0%
life	-0.0%	-0.0%	0.140	0.140	0.0%
listcompr	+0.0%	+0.0%	0.050	0.050	0.0%
listcopy	+0.0%	+0.0%	0.050	0.050	0.0%
mandel	-0.0%	-0.0%	0.030	0.030	0.0%
mandel2	-0.0%	-0.0%	0.000	0.000	0.0%
minimax	+0.0%	+0.0%	0.000	0.000	0.0%
multiplier	+0.0%	-3.1%	0.070	0.070	0.0%
nucleic2	0.0%	0.0%	0.030	0.030	0.0%
para	+0.3%	-2.7%	0.162	0.162	0.0%
parstof	+0.0%	-0.3%	0.000	0.000	0.0%
power	+0.1%	-0.0%	-3.1%	-3.9%	0.0%
pretty	-0.0%	+0.0%	0.000	0.000	0.0%
primetest	-0.0%	-0.0%	0.056	0.056	0.0%
puzzle	-0.0%	-17.1%	0.082	0.082	0.0%
rewrite	+0.0%	+19.2%	0.010	0.010	0.0%
scc	0.0%	0.0%	0.000	0.000	0.0%
sched	0.0%	0.0%	0.010	0.010	0.0%
simple	-0.6%	-4.9%	0.150	0.150	+3.4%
solid	-0.0%	0.0%	0.080	0.080	0.0%
sorting	0.0%	0.0%	0.000	0.000	0.0%
sphere	+0.0%	0.0%	0.022	0.022	0.0%
transform	-0.2%	-0.6%	0.198	0.200	0.0%
treejoin	+0.0%	0.0%	0.090	0.090	0.0%
typecheck	-0.0%	-0.0%	0.132	0.134	0.0%
wang	-0.0%	0.0%	0.064	0.064	+5.0%
wave4main	+0.0%	-0.0%	0.130	0.130	0.0%
Min	-0.6%	-17.1%	-6.3%	-6.2%	-10.0%
Max	+0.3%	+19.2%	+0.9%	+0.9%	+14.3%
Geometric Mean	-0.0%	-0.5%	-2.6%	-2.7%	+0.3%

Table 1. Results for the `spectral` NoFib tests.

Contification analysis of terms: $\boxed{(\mathcal{F},\mathcal{C}) = A\,[\![v]\!]}$

$$\begin{aligned}
A\,[\![x]\!] &= (\{x\},\emptyset) \\
A\,[\![\lambda x{:}\tau.v]\!] &= (\mathcal{F}\setminus\{x\},\mathcal{C}) && \text{where } (\mathcal{F},\mathcal{C}) = A\,[\![v]\!] \\
A\,[\![\Lambda a{:}\kappa.v]\!] &= A\,[\![v]\!] \\
A\,[\![K(\vec{\sigma},\vec{v})]\!] &= (\textstyle\bigcup\vec{\mathcal{F}},\bigcup\vec{\mathcal{C}}) && \text{where } \overrightarrow{(\mathcal{F},\mathcal{C})} = \overrightarrow{A\,[\![v]\!]} \\
A\,[\![\mu\mathsf{ret}.c]\!] &= (\mathcal{F},\mathcal{C}) && \text{where } (\mathcal{F},\emptyset,\mathcal{C}) = A\,[\![c]\!]
\end{aligned}$$

Contification analysis of continuations: $\boxed{(\mathcal{F},\mathcal{G},\mathcal{C}) = A\,[\![k]\!]}$

$$\begin{aligned}
A\,[\![v\cdot k]\!] &= (\mathcal{F},\emptyset,\mathcal{C})\oplus A\,[\![k]\!] && \text{where } (\mathcal{F},\mathcal{C}) = A\,[\![v]\!] \\
A\,[\![\sigma\cdot k]\!] &= A\,[\![k]\!] \\
A\,[\![\mathsf{ret}]\!] &= (\emptyset,\emptyset,\emptyset) \\
A\left[\!\!\left[\mathbf{case\,of}\,\overrightarrow{alt}\right]\!\!\right] &= \bigoplus\overrightarrow{A\,[\![alt]\!]}
\end{aligned}$$

Contification analysis of commands: $\boxed{(\mathcal{F},\mathcal{G},\mathcal{C}) = A\,[\![c]\!]}$

$$\begin{aligned}
A\,[\![\mathbf{let}\;bind\;\mathbf{in}\;c]\!] &= A\,[\![bind]\!]_{A[\![c]\!]} \\
A\,[\![\langle v\parallel k\rangle]\!] &= (\mathcal{F},\mathcal{G},\mathcal{C})\oplus A\,[\![k]\!] && \text{where } (\mathcal{F},\mathcal{C}) = A\,[\![v]\!] \\
&&& \mathcal{G} = \begin{cases}\{f\} & \text{if } v = f, k = \vec{v'}\cdot\mathsf{ret}, |\vec{v'}| = arity(f) \\ \emptyset & \text{otherwise}\end{cases} \\
A\,[\![\mathsf{jump}\;j\;\vec{\sigma}\;\vec{v}]\!] &= (\textstyle\bigcup\vec{\mathcal{F}},\bigcup\vec{\mathcal{C}}) && \text{where } \overrightarrow{(\mathcal{F},\mathcal{C})} = \overrightarrow{A\,[\![v]\!]}
\end{aligned}$$

Contification analysis of bindings: $\boxed{(\mathcal{F},\mathcal{G},\mathcal{C}) = A\,[\![bind]\!]_{(\mathcal{F}_b,\mathcal{G}_b,\mathcal{C}_b)}}$

$$\begin{aligned}
A\,[\![f{:}\tau = v]\!]_{(\mathcal{F}_b,\mathcal{G}_b,\mathcal{C}_b)} &= (\mathcal{F}'\setminus\{f\},\mathcal{G}'\setminus\{f\},\mathcal{C}'') \\
&\text{where} \quad (\mathcal{F},\mathcal{C}) = A\,[\![v]\!] \\
&\qquad (\mathcal{F}',\mathcal{G}',\mathcal{C}') = (\mathcal{F},\emptyset,\mathcal{C})\oplus(\mathcal{F}_b,\mathcal{G}_b,\mathcal{C}_b) \\
&\qquad \mathcal{C}'' = \mathcal{C}'\cup(\{f\}\cap\mathcal{G}_b) \\
A\,[\![j{:}\tau = \tilde{\mu}[\overrightarrow{a{:}\kappa},\overrightarrow{x{:}\sigma}].c]\!]_{(\mathcal{F}_b,\mathcal{G}_b,\mathcal{C}_b)} &= (\mathcal{F}\setminus\{\vec{x}\},\mathcal{G}\setminus\{\vec{x}\},\mathcal{C})\oplus(\mathcal{F}_b,\mathcal{G}_b,\mathcal{C}_b) \\
&\text{where } (\mathcal{F},\mathcal{G},\mathcal{C}) = A\,[\![c]\!] \\
A\left[\!\!\left[\mathbf{rec}\left\{\overrightarrow{f{:}\tau = v}\right\}\right]\!\!\right]_{(\mathcal{F}_b,\mathcal{G}_b,\mathcal{C}_b)} &= (\mathcal{F}''\setminus\{\vec{f}\},\mathcal{G}'\setminus\{\vec{f}\},\mathcal{C}''') \\
&\text{where} \quad \overrightarrow{(\mathcal{F},\mathcal{C})} = \overrightarrow{A\,[\![v]\!]} \\
&\qquad (\mathcal{F}'',\mathcal{G}',\mathcal{C}'') = (\textstyle\bigcup\vec{\mathcal{F}},\emptyset,\bigcup\vec{\mathcal{C}})\oplus(\mathcal{F}_b,\mathcal{G}_b,\mathcal{C}_b) \\
&\qquad \mathcal{C}''' = \begin{cases}\mathcal{C}''\cup\mathcal{G}' & \text{if } \{\vec{f}\}\subseteq\mathcal{G}' \\ \mathcal{C}'' & \text{otherwise}\end{cases} \\
A\left[\!\!\left[\mathbf{rec}\left\{\overrightarrow{j{:}\tau = \tilde{\mu}[\overrightarrow{a{:}\kappa},\overrightarrow{x{:}\sigma}].c}\right\}\right]\!\!\right]_{(\mathcal{F}_b,\mathcal{G}_b,\mathcal{C}_b)} &= (\mathcal{F}',\mathcal{G}',\mathcal{C}') \\
&\text{where} \quad \overrightarrow{(\mathcal{F},\mathcal{G},\mathcal{C})} = \overrightarrow{A\,[\![c]\!]} \\
&\qquad (\mathcal{F}',\mathcal{G}',\mathcal{C}') = \left(\bigoplus\overrightarrow{(\mathcal{F}\setminus\{\vec{x}\},\mathcal{G}\setminus\{\vec{x}\},\mathcal{C})}\right)\oplus(\mathcal{F}_b,\mathcal{G}_b,\mathcal{C}_b)
\end{aligned}$$

Contification analysis of alternatives: $\boxed{(\mathcal{F},\mathcal{G},\mathcal{C}) = A\,[\![alt]\!]}$

$$\begin{aligned}
A\,[\![x{:}\tau\to c]\!] &= (\mathcal{F}\setminus\{x\},\mathcal{G}\setminus\{x\},\mathcal{C}) && \text{where } (\mathcal{F},\mathcal{G},\mathcal{C}) = A\,[\![c]\!] \\
A\,[\![K(\overrightarrow{a{:}\kappa},\overrightarrow{x{:}\tau})\to c]\!] &= (\mathcal{F}\setminus\{\vec{x}\},\mathcal{G}\setminus\{\vec{x}\},\mathcal{C}) && \text{where } (\mathcal{F},\mathcal{G},\mathcal{C}) = A\,[\![c]\!]
\end{aligned}$$

Combination of contification analyses: $\boxed{(\mathcal{F}',\mathcal{G}',\mathcal{C}') = (\mathcal{F}_1,\mathcal{G}_1,\mathcal{C}_1)\oplus(\mathcal{F}_2,\mathcal{G}_2,\mathcal{C}_2)}$

$$(\mathcal{F}_1,\mathcal{G}_1,\mathcal{C}_1)\oplus(\mathcal{F}_2,\mathcal{G}_2,\mathcal{C}_2) = (\mathcal{F}_1\cup\mathcal{F}_2,\ (\mathcal{G}_1\setminus\mathcal{F}_2)\cup(\mathcal{G}_2\setminus\mathcal{F}_1)\cup(\mathcal{G}_1\cap\mathcal{G}_2),\mathcal{C}_1\cup\mathcal{C}_2)$$

Figure 9. The analysis phase A of the contification pass, including the operator $\oplus$ for combining analyses.

Test	Size	Allocs	Time	Elapsed	Memory
anna	-0.1%	+1.2%	0.152	0.152	0.0%
bspt	+1.8%	0.0%	0.020	0.020	0.0%
cacheprof	+0.5%	+19.7%	+10.9%	+10.9%	+4.5%
compress	+0.1%	0.0%	-0.6%	-0.6%	0.0%
compress2	-0.2%	0.0%	+0.5%	+0.5%	0.0%
fem	+0.1%	-0.1%	0.046	0.046	0.0%
fluid	+0.4%	+0.1%	0.010	0.010	0.0%
fulsom	+0.7%	-8.8%	-4.9%	-4.9%	-7.1%
gamteb	-0.1%	+0.1%	0.056	0.056	0.0%
gg	-0.1%	+2.3%	0.028	0.028	0.0%
grep	+0.0%	0.0%	0.000	0.000	0.0%
hidden	-0.2%	-0.9%	+0.7%	+1.0%	0.0%
hpg	-0.1%	-0.2%	0.132	0.132	0.0%
infer	-0.5%	-0.1%	0.100	0.100	0.0%
lift	-0.2%	-1.2%	0.000	0.000	0.0%
maillist	+0.0%	-0.0%	0.060	0.060	+2.6%
mkhprog	-0.1%	-0.0%	0.000	0.000	0.0%
parser	-0.2%	+2.1%	0.050	0.050	0.0%
pic	-0.0%	-0.7%	0.010	0.010	0.0%
prolog	+0.2%	+0.3%	0.000	0.000	0.0%
reptile	+0.1%	+0.0%	0.022	0.022	0.0%
rsa	-0.0%	-0.0%	0.010	0.010	0.0%
scs	+0.2%	-0.6%	-0.0%	-0.0%	0.0%
symalg	-0.2%	-0.0%	0.000	0.000	0.0%
veritas	+0.4%	-0.1%	0.000	0.000	0.0%
Min	-0.5%	-8.8%	-4.9%	-4.9%	-7.1%
Max	+1.8%	+19.7%	+10.9%	+10.9%	+4.5%
Geometric Mean	+0.1%	+0.4%	+1.0%	+1.0%	-0.0%

Table 2. Results for the `real` NoFib tests.

To prove standardization, we will also make use of a *parallel reduction* relation $\Rightarrow$. Parallel reduction consists of the simultaneous reduction of some number of redexes, possibly zero, appearing in the same term. Clearly, parallel reduction sits between reduction and its reflexive-transitive closure:

$$\to\ \subset\ \Rightarrow\ \subset\ \to^\star$$

Finally, we have *non-standard parallel reduction*, $\twoheadrightarrowtail$, which may contract several redexes but *not* the standard redex.

C.2 Algebraic Properties

We will need a few algebraic properties of the CBN calculus. Most important is standardization (Proposition 8).

An easy property is that irreducibility is preserved by general reduction, and unaffected by non-standard reduction:

Lemma 5. *If $c \to c'$ then $c \not\mapsto$ implies $c' \not\mapsto$. Furthermore, if $c \rightarrowtail c'$ then $c \not\mapsto$ iff $c' \not\mapsto$.*

Proof. The second property (if $c \rightarrowtail c'$ then $c \not\mapsto$ iff $c' \not\mapsto$) can be shown by cases on the possible non-standard reductions, since a non-standard reduction never introduces or destroys a standard redex. The first property is implied by the second because when a command without a standard step is reduced, that reduction must have been non-standard to begin with. $\square$

Also relatively simple is a standard substitution lemma, though there are fine points to its statement.

Lemma 6 (Substitution). *1. If $c \to c'$ then $c\,\{\sigma/a\} \to c'\,\{\sigma/a\}$.*
2. If $c \to c'$ then $c\,\{v/x\} \to c'\,\{v/x\}$.
3. If $c \to c'$ then $c\,\{k/\mathsf{ret}\} \to c'\,\{k/\mathsf{ret}\}$.
4. If $c \to c'$ then $c\left\{c''\overrightarrow{\{\sigma/a\}}\,\overrightarrow{\{v/x\}}/\mathsf{jump}\ j\ \vec{\sigma}\ \vec{v}\right\} \to^\star c'\left\{c''\overrightarrow{\{\sigma/a\}}\,\overrightarrow{\{v/x\}}/\mathsf{jump}\ j\ \vec{\sigma}\ \vec{v}\right\}$.
5. If $v \to v'$ then $c\,\{v/x\} \to^\star c\,\{v'/x\}$.
6. If $k \to k'$ then $c\,\{k/\mathsf{ret}\} \to^\star c\,\{k'/\mathsf{ret}\}$.
7. If $c' \to c''$ then $c\left\{c'\overrightarrow{\{\sigma/a\}}\,\overrightarrow{\{v/x\}}/\mathsf{jump}\ j\ \vec{\sigma}\ \vec{v}\right\} \to^\star c\left\{c''\overrightarrow{\{\sigma/a\}}\,\overrightarrow{\{v/x\}}/\mathsf{jump}\ j\ \vec{\sigma}\ \vec{v}\right\}$.

Proof. By mutual induction. The one subtlety is that substituting a type, term, or continuation into a command cannot duplicate or destroy redexes in the original command, but substituting a join point can. Hence clauses 1–3 specify $\to$ but clause 4 specifies $\to^\star$.

Here is the crucial case of clause 4: Suppose $c \equiv \mathsf{jump}\ j\ \overrightarrow{\sigma'}\ \overrightarrow{v'}$. Since c cannot take a standard reduction, the reduction must occur in some subterm. Hence $\overrightarrow{v'} \to \overrightarrow{v''}$ and $c' \equiv \mathsf{jump}\ j\ \overrightarrow{\sigma'}\ \overrightarrow{v''}$. Now:

$$\begin{aligned}
&c\left\{c''\overrightarrow{\{\sigma/a\}}\,\overrightarrow{\{v/x\}}/\mathsf{jump}\ j\ \vec{\sigma}\ \vec{v}\right\}\\
&\quad\equiv c''\,\overrightarrow{\{\sigma'/a\}}\,\overrightarrow{\{v'/x\}}\\
&\quad\to^\star c''\,\overrightarrow{\{\sigma'/a\}}\,\overrightarrow{\{v''/x\}} && \text{(by 5)}\\
&\quad\equiv c'\left\{c''\overrightarrow{\{\sigma/a\}}\,\overrightarrow{\{v/x\}}/\mathsf{jump}\ j\ \vec{\sigma}\ \vec{v}\right\}
\end{aligned}$$
$\square$

Parallel reduction enjoys a similar substitution lemma (in fact, it is why parallel reduction is useful!).

Lemma 7. *Let z denote any term, continuation, or command, and likewise z'.*

1. If $z \Rightarrow z'$, then $z\,\{\tau/a\} \Rightarrow z'\,\{\tau/a\}$.
2. If $z \Rightarrow z'$ and $v \Rightarrow v'$, then $z\,\{v/x\} \Rightarrow z'\,\{v'/x\}$.
3. If $z \Rightarrow z'$ and $k \Rightarrow k'$, then $z\,\{k/\mathsf{ret}\} \Rightarrow z'\,\{k/\mathsf{ret}\}$.
4. If $z \Rightarrow z'$ and $c \Rightarrow c'$, then $z\left\{c\overrightarrow{\{\sigma/a\}}\,\overrightarrow{\{v/x\}}/\mathsf{jump}\ j\ \vec{\sigma}\ \vec{v}\right\} \Rightarrow z'\left\{c'\overrightarrow{\{\sigma/a\}}\,\overrightarrow{\{v/x\}}/\mathsf{jump}\ j\ \vec{\sigma}\ \vec{v}\right\}$.

Proof. Each proceeds by induction, with 4 relying on 1 and 2. We show the crucial case in 4.

Suppose $z \equiv \mathsf{jump}\ j\ \overrightarrow{\sigma'}\ \overrightarrow{v'}$. Hence $\overrightarrow{v' \Rightarrow v''}$ and $z' \equiv \mathsf{jump}\ j\ \overrightarrow{\sigma'}\ \overrightarrow{v''}$. Then:

$$\begin{aligned}
&z\left\{c\overrightarrow{\{\sigma/a\}}\,\overrightarrow{\{v/x\}}/\mathsf{jump}\ j\ \vec{\sigma}\ \vec{v}\right\}\\
&\quad\equiv c\overrightarrow{\{\sigma'/a\}}\,\overrightarrow{\{v'/x\}}\\
&\quad\Rightarrow c'\,\overrightarrow{\{\sigma'/a\}}\,\overrightarrow{\{v''/x\}} && \text{(by 1 and 2)}\\
&\quad\equiv z'\left\{c'\overrightarrow{\{\sigma/a\}}\,\overrightarrow{\{v/x\}}/\mathsf{jump}\ j\ \vec{\sigma}\ \vec{v}\right\}
\end{aligned}$$
$\square$

Now we have the tools to prove standardization:

Proposition 8 (Standardization)**.** *If $c \to^\star c' \not\mapsto$ then $c \mapsto^\star c_1 \not\mapsto$ and $c_1 \to^\star c'$.*

We will prove Proposition 8 in several parts. The crux is the diagram in Fig. 10, which allows us to take any series of standard reductions and move them upward, postponing any non-standard reductions until later.

$$\begin{tikzcd}
c \arrow[r, mapsto, dashed, "\star"] \arrow[d, tail, "\star"] & c' \arrow[d, tail, dashed, "\star"] \\
d \arrow[r, mapsto, "\star"] & d'
\end{tikzcd}$$

Figure 10. Core of standardization proof.

A direct attempt at this diagram will fail, however, as we cannot get a footing. What holds for single reductions is:

$$\begin{tikzcd} c \arrow[r, dashed, mapsto, "\star"] \arrow[d, tail] & c' \arrow[d, dashed, tail, "\star"] \\ d \arrow[r, mapsto] & d' \end{tikzcd}$$

Moving a standard reduction forward may produce extra work, so there may be extra steps between c' and d'; also, the non-standard reduction may happen to be the next standard reduction, leading to *more* standard steps between c and c'. But this diagram cannot be "tiled" by induction to produce Fig. 10.

Happily, the extra work created by moving a standard reduction forward always has a particular form—the duplicated redexes can all be reduced in parallel. Thus we can obtain a more helpful diagram using parallel reduction.

Lemma 9. *If $c \Rightarrow d$ and $d \mapsto d'$, then $c \mapsto^\star c'$ with $c' \Rightarrow d'$.*

$$\begin{tikzcd} c \arrow[r, dashed, mapsto, "\star"] \arrow[d, Rightarrow] & c' \arrow[d, dashed, Rightarrow] \\ d \arrow[r, mapsto] & d' \end{tikzcd}$$

Proof. Proceed by induction on the number of reductions in $c \Rightarrow d$. If $c \equiv d$, we can take $c' \equiv d$ and we are done.

Otherwise, consider whether $c \Rightarrow d$ takes the standard reduction. If it does, we can refactor it as $c \mapsto c_1 \Rightarrow d$:

$$\begin{tikzcd} c \arrow[r, mapsto] \arrow[d, Rightarrow] & c_1 \arrow[r, dashed, mapsto, "\star"] \arrow[dl, Rightarrow] & c' \arrow[d, dashed, Rightarrow] \\ d \arrow[rr, mapsto] & & d' \end{tikzcd}$$

Since $c_1 \Rightarrow d$ by fewer reductions than $c \Rightarrow d$, we can finish using the induction hypothesis.

Otherwise, since we are not performing the standard reduction (which is always at the top level in our language), c and d have the same top-level form, simplifying the case analysis:

- The case where c is a jump is impossible, since then d would have to be a jump and a jump cannot take a standard reduction.
- Suppose $c \equiv \mathbf{let}\, x = v \,\mathbf{in}\, c_0$ and $d \equiv \mathbf{let}\, x = v' \,\mathbf{in}\, c_0'$ with $v \Rightarrow v'$ and $c_0 \Rightarrow c_0'$. Then $d \mapsto d' \equiv c_0' \{v'/x\}$. Let $c' \equiv c_0 \{v/x\}$; then $c \mapsto c'$, and by Lemma 7, $c' \Rightarrow d'$.
- The case for $c \equiv \mathbf{let}\, j = \tilde{\mu}[\vec{a}, \vec{x}].d_0 \,\mathbf{in}\, c_0$ is similar.
- Finally, we have that $c \equiv \langle v \,\|\, k\rangle$ and $d \equiv \langle v' \,\|\, k'\rangle$ with $v \Rightarrow v'$ and $k \Rightarrow k'$. Continue by case analysis on v:
  - The case $v \equiv x$ is impossible since $\langle x \,\|\, k'\rangle$ cannot take a standard reduction.
  - If $v \equiv \mu\mathsf{ret}.c_0$, we must account for a possible μ_η-reduction. In this case, $c_0 \equiv \langle v_0 \,\|\, \mathsf{ret}\rangle$ and $v_0 \Rightarrow v'$.

$$\begin{tikzcd} v_0 \arrow[d, Rightarrow] & k \arrow[d, Rightarrow] & c \equiv \langle \mu\mathsf{ret}.\, \langle v_0 \,\|\, \mathsf{ret}\rangle \,\|\, k\rangle \arrow[r, dashed, mapsto] \arrow[d, Rightarrow] & \langle v_0 \,\|\, k\rangle \arrow[dl, dashed, Rightarrow] \\ v' & k' & d \equiv \langle v' \,\|\, k'\rangle \arrow[r, mapsto] & d' \end{tikzcd}$$

Since we have removed one reduction, we can use the induction hypothesis to finish:

$$\begin{tikzcd} c \equiv \langle \mu\mathsf{ret}.\, \langle v_0 \,\|\, \mathsf{ret}\rangle \,\|\, k\rangle \arrow[r, mapsto] \arrow[d, Rightarrow] & \langle v_0 \,\|\, k\rangle \arrow[r, dashed, mapsto, "\star"] \arrow[dl, Rightarrow] & c' \arrow[d, dashed, Rightarrow] \\ d \equiv \langle v' \,\|\, k'\rangle \arrow[rr, mapsto] & & d' \end{tikzcd}$$

If there is no μ_η-reduction in v, then $v' \equiv \mu\mathsf{ret}.c_0'$ with $c_0 \Rightarrow c_0'$:

$$\begin{tikzcd} c_0 \arrow[d, Rightarrow] & k \arrow[d, Rightarrow] & c \equiv \langle \mu\mathsf{ret}.c_0 \,\|\, k\rangle \arrow[r, dashed, mapsto] \arrow[d, Rightarrow] & c_0 \{k/\mathsf{ret}\} \arrow[d, dashed, Rightarrow] \\ c_0' & k' & d \equiv \langle \mu\mathsf{ret}.c_0' \,\|\, k'\rangle \arrow[r, mapsto] & c_0' \{k'/\mathsf{ret}\} \end{tikzcd}$$

Here we take $c' \equiv c_0 \{k/\mathsf{ret}\}$ and finish by Lemma 7.

  - In the other cases, v is a WHNF, so k is a matching continuation (that is, not ret and not a mismatched **case**). We show the case for $v \equiv \lambda x.v_0$; the Λ-case is simpler, and the constructor case is more complex but no more illuminating.

$$\begin{tikzcd} v_0, v_1, k_0 \arrow[d, Rightarrow] & \langle \lambda x.v_0 \,\|\, v_1 \cdot k_0\rangle \arrow[r, dashed, mapsto] \arrow[d, Rightarrow] & \langle v_0 \{v_1/x\} \,\|\, k_0\rangle \arrow[d, dashed, Rightarrow] \\ v_0', v_1', k_0' & \langle \lambda x.v_0' \,\|\, v_1' \cdot k_0\rangle \arrow[r, mapsto] & \langle v_0' \{v_1'/x\} \,\|\, k_0'\rangle \end{tikzcd}$$

As before, along the right side we use Lemma 7. ☐

Now we need to reconcile Lemma 9 with Fig. 10. First, we can break down $\Rightarrow$ into standard reduction ($\mapsto$) followed by non-standard parallel reduction ($\mathrel{>\!\!\!\Rightarrow}$):

Lemma 10. *If $c \Rightarrow c'$ then $c \mapsto^\star\mathrel{>\!\!\!\Rightarrow} c'$.*

Proof. By induction on the number of reductions taken by the derivation of $c \Rightarrow c'$. If $c \equiv c'$, we are done. Otherwise, if $c \Rightarrow c'$ does not take the standard reduction, we are again done. Finally, if it does take the standard reduction, then $c \mapsto c_1 \Rightarrow c'$ where $c_1 \Rightarrow c'$ takes fewer reductions, so we finish using the induction hypothesis. ☐

Now we can move Lemma 9 closer by referring to non-standard (parallel) reduction:

Lemma 11. *If $c \mathrel{>\!\!\!\Rightarrow} d$ and $d \mapsto d'$, then $c \mapsto^\star c'$ with $c' \mathrel{>\!\!\!\Rightarrow} d'$.*

$$\begin{tikzcd} c \arrow[r, dashed, mapsto, "\star"] \arrow[d, tail, Rightarrow] & c' \arrow[d, dashed, tail, Rightarrow] \\ d \arrow[r, mapsto] & d' \end{tikzcd}$$

Proof. Immediate from Lemmas 9 and 10. ☐

This is the diagram that we can "tile" to produce Fig. 10.

Lemma 12. *If $c \rightarrowtail^\star d$ and $d \mapsto^\star d'$, then $c \mapsto^\star c'$ with $c' \rightarrowtail^\star d'$ (see Fig. 10).*

Proof. Follows from Lemma 11. By induction, its diagram can be "tiled horizontally" to give:

$$\begin{tikzcd} c \arrow[r, dashed, mapsto, "\star"] \arrow[d, tail, Rightarrow] & c' \arrow[d, dashed, tail, Rightarrow] \\ d \arrow[r, mapsto, "\star"] & d' \end{tikzcd}$$

Now that the top and bottom match, we can "tile vertically":

$$\begin{tikzcd} c \arrow[r, dashed, mapsto, "\star"] \arrow[d, tail, Rightarrow, "\star"] & c' \arrow[d, dashed, tail, Rightarrow, "\star"] \\ d \arrow[r, mapsto, "\star"] & d' \end{tikzcd}$$

But then $\mathrel{>\!\!\!\Rightarrow}^\star = \rightarrowtail^\star$, and we're done. ☐

Now we are ready to prove standardization.

Proof of Proposition 8.

$$
\begin{tikzcd}
c \arrow[r, dashed, mapsto, "\star"] \arrow[dr, "\star"'] & c_1 \not\mapsto \arrow[d, dashed, "\star"] \\
& c' \not\mapsto
\end{tikzcd}
$$

Proceed by induction on $c \to^\star c'$. If there are no standard reductions in the sequence, then we can take $c_1 \equiv c$ and we're done by Lemma 5.

Thus assume there is at least one standard reduction; isolating the first one, we have $c \rightarrowtail^\star d \mapsto d' \to^\star c'$.

$$
\begin{tikzcd}
c \arrow[d, tail, "\star"'] \arrow[dr, "\star"] & & \\
d \arrow[r, mapsto] & d' \arrow[dr, "\star"] & \\
& & c' \not\mapsto
\end{tikzcd}
$$

We can fill in the upper-left corner by Lemma 12 and the lower-right corner by the induction hypothesis:

$$
\begin{tikzcd}
c \arrow[r, dashed, mapsto, "\star"] \arrow[d, tail, "\star"'] \arrow[dr, "\star"] & {} \arrow[d, tail, dashed, "\star"] & \\
d \arrow[r, mapsto] & d' \arrow[r, dashed, mapsto, "\star"] \arrow[dr, "\star"] & {} \not\mapsto \arrow[d, dashed, "\star"] \\
& & c' \not\mapsto
\end{tikzcd}
$$

Finally, applying Lemma 12 again, along with Lemma 5, gives us our c_1:

$$
\begin{tikzcd}
c \arrow[r, dashed, mapsto, "\star"] \arrow[d, tail, "\star"'] \arrow[dr, "\star"] & {} \arrow[d, tail, dashed, "\star"] \arrow[r, dashed, mapsto, "\star"] & c_1 \not\mapsto \arrow[d, tail, dashed, "\star"] \\
d \arrow[r, mapsto] & d' \arrow[r, dashed, mapsto, "\star"] \arrow[dr, "\star"] & {} \not\mapsto \arrow[d, dashed, "\star"] \\
& & c' \not\mapsto
\end{tikzcd}
$$
□

We will also have occasion to move a standard reduction *after* a non-standard one:

Lemma 13. *If $c \mapsto c'$ and $c \to^\star d$, then $d \mapsto^{0/1} d'$ with $c' \to^\star d'$.*

$$
\begin{tikzcd}
c \arrow[r, mapsto] \arrow[d, "\star"'] & c' \arrow[d, dashed, "\star"] \\
d \arrow[r, dashed, mapsto, "0/1"] & d'
\end{tikzcd}
$$

Proof. Proceed by induction on $c \to^\star d$. If $c \equiv d$, then trivially we take $d' \equiv c'$.

Now suppose $c \to c_1 \to^\star d$. By the induction hypothesis, if we can find c_1' with $c' \to^\star c_1'$ and $c_1 \mapsto c_1'$, then we get d' with $d \mapsto^{0/1} d'$ and $c_1' \to^\star d'$ and we're done.

$$
\begin{tikzcd}
c \arrow[r, mapsto] \arrow[d] & c' \arrow[d, dashed, "\star"] \\
c_1 \arrow[r, dashed, mapsto] \arrow[d, "\star"'] & c_1' \arrow[d, dotted, "\star"] \\
d \arrow[r, dotted, mapsto, "0/1"] & d'
\end{tikzcd}
$$

Alternatively, if we find that $c_1' \equiv c_1$ (i.e., that $c' \to^\star c_1$), then we can pick $d' \equiv d$ as well.

Now, if in fact $c \mapsto c_1$, then we must have that $c' \equiv c_1$ so we take $c_1' \equiv c_1$. Otherwise, $c \to c_1$ by a non-standard reduction. Considering the cases for $c \mapsto c_1$, none can be interfered with by a reduction in a subterm, and such a reduction can always be performed afterward (though it may be replicated if the subterm is a substituend in the right-hand side of the rule). Thus the standard reduction can still take place in c_1, with the nonstandard reduction postponed to part of $c' \to^\star c_1'$. □

C.3 Readback Function

To show the correspondence between the call-by-name and call-by-need calculi, we will use a *bisimulation*. Key to defining the bisimulation will be our *readback function*, defined in Fig. 11.

The key property of the readback is this:

Lemma 14. *If*

$$\langle \mathcal{H};\ \mathcal{J}, \mathcal{R};\ c\rangle \rightsquigarrow \langle \mathcal{H}';\ \mathcal{J}', \mathcal{R}';\ c'\rangle,$$

then

$$\mathcal{U}(\langle \mathcal{H};\ \mathcal{J}, \mathcal{R};\ c\rangle) \to^\star \mathcal{U}(\langle \mathcal{H}';\ \mathcal{J}', \mathcal{R}';\ c'\rangle).$$

First, we will need a simple fact about reduction:

Lemma 15. *If $c \to c'$, then*

1. $\mathcal{U}_H(\mathcal{H})(c) \to \mathcal{U}_H(\mathcal{H})(c')$
2. $\mathcal{U}_J(\mathcal{J})(c) \to^\star \mathcal{U}_J(\mathcal{J})(c')$
3. $\mathcal{U}_R(\mathcal{R})(c) \to^\star \mathcal{U}_R(\mathcal{R})(c')$

Proof. Parts 1 and 2 are immediate from Lemma 6, since $\mathcal{U}_H$ and $\mathcal{U}_J$ produce substitutions.

For part 3, note that $\mathcal{U}_R(\mathcal{R})(c)$ always has the form $c\sigma$ for some substitution σ (which may include c in a substituend), so the substitution argument still applies—except that multiple reductions may be necessary since c may be copied. □

Proof of Lemma 14. By case analysis on the reduction:

- $(\beta^{\to})$

$$
\begin{aligned}
&\mathcal{U}(\langle \mathcal{H};\ \mathcal{J}, \mathcal{R};\ \langle \lambda x{:}\tau.v_1 \,\|\, v_2 \cdot k\rangle\rangle) \\
&\quad \equiv \mathcal{U}_H(\mathcal{H})(\mathcal{U}_R(\mathcal{R})(\mathcal{U}_J(\mathcal{J})(\langle \lambda x{:}\tau.v_1 \,\|\, v_2 \cdot k\rangle))) \\
&\quad \to^\star \mathcal{U}_H(\mathcal{H})(\mathcal{U}_R(\mathcal{R})(\mathcal{U}_J(\mathcal{J})(\langle v_1\,\{v_2/x\} \,\|\, k\rangle))) \\
&\quad \equiv (\mathcal{U}_H(\mathcal{H}) \circ \{v_2/x\})(\mathcal{U}_R(\mathcal{R})(\mathcal{U}_J(\mathcal{J})(\langle v_1 \,\|\, k\rangle))) \\
&\quad \equiv \mathcal{U}(\langle \mathcal{H}, x = v_2;\ \mathcal{J}, \mathcal{R};\ \langle v_1 \,\|\, k\rangle\rangle)
\end{aligned}
$$

- $(\beta^{\forall})$

$$
\begin{aligned}
&\mathcal{U}(\langle \mathcal{H};\ \mathcal{J}, \mathcal{R};\ \langle \Lambda a{:}\kappa.v \,\|\, \tau \cdot k\rangle\rangle) \\
&\quad \equiv \mathcal{U}_H(\mathcal{H})(\mathcal{U}_R(\mathcal{R})(\mathcal{U}_J(\mathcal{J})(\langle \Lambda a{:}\kappa.v \,\|\, \tau \cdot k\rangle))) \\
&\quad \to^\star \mathcal{U}_H(\mathcal{H})(\mathcal{U}_R(\mathcal{R})(\mathcal{U}_J(\mathcal{J})(\langle v\,\{\tau/a\} \,\|\, k\rangle))) \\
&\quad \equiv \mathcal{U}(\langle \mathcal{H};\ \mathcal{J}, \mathcal{R};\ \langle v\,\{\tau/a\} \,\|\, k\rangle\rangle)
\end{aligned}
$$

- Similarly for $\mathbf{case}_{cons}$ and $\mathbf{case}_{def}$.

$$\mathcal{U}(\langle \mathcal{H};\ \mathcal{J}, \mathcal{R};\ c\rangle) = \mathcal{U}_H(\mathcal{H})(\mathcal{U}_R(\mathcal{R})(\mathcal{U}_J(\mathcal{J})(c)))$$

$$\begin{aligned} \mathcal{U}_H(\varepsilon) &= id \\ \mathcal{U}_H(\mathcal{H}, x = v) &= \mathcal{U}_H(\mathcal{H}) \circ \{v/x\} \\ \mathcal{U}_H(\mathcal{H}, x = \bullet) &= \mathcal{U}_H(\mathcal{H}) \end{aligned}$$

$$\begin{aligned} \mathcal{U}_J(\varepsilon) &= id \\ \mathcal{U}_J(\mathcal{J}, j = \tilde{\mu}[\overrightarrow{a{:}\kappa}\ \overrightarrow{x{:}\tau}].c) &= \mathcal{U}_J(\mathcal{J}) \circ \left\{ c\overrightarrow{\{\tau/a\}}\overrightarrow{\{v/x\}}/\mathsf{jump}\ j\ \vec{\tau}\ \vec{v} \right\} \end{aligned}$$

$$\begin{aligned} \mathcal{U}_R(\varepsilon)(c) &= id \\ \mathcal{U}_R((k, \mathcal{J}) : \mathcal{R})(c) &= \mathcal{U}_R(\mathcal{R})(c\, \{\mathcal{U}_J(\mathcal{J})(k)/\mathsf{ret}\}) \\ \mathcal{U}_R(\mathbf{upd}\, x : \mathcal{R})(c) &= (\mathcal{U}_R(\mathcal{R})(c))\, \{\mu\mathsf{ret}.c/x\} \end{aligned}$$

Figure 11. Readback function.

- (μ)

$$\begin{aligned} &\mathcal{U}(\langle \mathcal{H};\ \mathcal{J}, \mathcal{R};\ \langle \mu\mathsf{ret}.c \,\|\, k\rangle\rangle) \\ &\quad \equiv \mathcal{U}_H(\mathcal{H})(\mathcal{U}_R(\mathcal{R})(\mathcal{U}_J(\mathcal{J})(\langle \mu\mathsf{ret}.c \,\|\, k\rangle))) \\ &\quad \equiv \mathcal{U}_H(\mathcal{H})(\mathcal{U}_R(\mathcal{R})(\langle \mu\mathsf{ret}.c \,\|\, \mathcal{U}_J(\mathcal{J})(k)\rangle)) \\ &\quad \to^\star \mathcal{U}_H(\mathcal{H})(\mathcal{U}_R(\mathcal{R})(c\, \{\mathcal{U}_J(\mathcal{J})(k)/\mathsf{ret}\})) \\ &\quad \equiv \mathcal{U}_H(\mathcal{H})(\mathcal{U}_R((k, \mathcal{J}) : \mathcal{R})(c)) \\ &\quad \equiv \mathcal{U}_H(\mathcal{H})(\mathcal{U}_R((k, \mathcal{J}) : \mathcal{R})(\mathcal{U}_J(\varepsilon)(c))) \\ &\quad \equiv \mathcal{U}(\langle \mathcal{H};\ \varepsilon, (k, \mathcal{J}) : \mathcal{R};\ c\rangle) \end{aligned}$$

Note that in pushing the substitution $\mathcal{U}_J(\mathcal{J})$ into the continuation of the command above, specifically:

$$\mathcal{U}_J(\mathcal{J})(\langle \mu\mathsf{ret}.c \,\|\, k\rangle) \equiv \langle \mu\mathsf{ret}.c \,\|\, \mathcal{U}_J(\mathcal{J})(k)\rangle$$

we exploit the fact that the term $\mu\mathsf{ret}.c$ must be continuation-closed, so that it is unaffected by $\mathcal{U}_J(\mathcal{J})$.

- (jump)
Assuming that

$$j = \tilde{\mu}[\overrightarrow{a{:}\kappa}, \overrightarrow{x{:}\tau}].c \in \mathcal{J},$$

$$\begin{aligned} &\mathcal{U}(\langle \mathcal{H};\ \mathcal{J}, \mathcal{R};\ \mathsf{jump}\ j\ \vec{\sigma}\ \vec{v}\rangle) \\ &\quad \equiv \mathcal{U}_H(\mathcal{H})(\mathcal{U}_R(\mathcal{R})(\mathcal{U}_J(\mathcal{J})(\mathsf{jump}\ j\ \vec{\sigma}\ \vec{v}))) \\ &\quad \equiv \mathcal{U}_H(\mathcal{H})(\mathcal{U}_R(\mathcal{R})(\mathcal{U}_J(\mathcal{J})(c\overrightarrow{\{\sigma/a\}}\overrightarrow{\{v/x\}}))) \\ &\quad \equiv (\mathcal{U}_H(\mathcal{H}) \circ \overrightarrow{\{v/x\}})(\mathcal{U}_R(\mathcal{R})(\mathcal{U}_J(\mathcal{J})(c\overrightarrow{\{\sigma/a\}}))) \\ &\quad \equiv \mathcal{U}(\langle \mathcal{H}, \overrightarrow{x = v};\ \mathcal{J}, \mathcal{R};\ c\overrightarrow{\{\sigma/a\}}\rangle) \end{aligned}$$

- (lookup)
Assuming that

$$x = V \in \mathcal{H},$$

$$\begin{aligned} &\mathcal{U}(\langle \mathcal{H};\ \mathcal{J}, \mathcal{R};\ \langle x \,\|\, k\rangle\rangle) \\ &\quad \equiv \mathcal{U}_H(\mathcal{H})(\mathcal{U}_R(\mathcal{R})(\mathcal{U}_J(\mathcal{J})(\langle x \,\|\, k\rangle))) \\ &\quad \equiv \mathcal{U}_H(\mathcal{H})(\mathcal{U}_R(\mathcal{R})(\mathcal{U}_J(\mathcal{J})(\langle V \,\|\, k\rangle))) \\ &\quad \equiv \mathcal{U}(\langle \mathcal{H};\ \mathcal{J}, \mathcal{R};\ \langle V \,\|\, k\rangle\rangle) \end{aligned}$$

- (lazysubst)
Assuming that

$$x = K(\vec{\sigma}, \vec{v}) \in \mathcal{H},$$

$$\begin{aligned} &\mathcal{U}(\langle \mathcal{H};\ \mathcal{J}, \mathcal{R};\ \langle x \,\|\, k\rangle\rangle) \\ &\quad \equiv \mathcal{U}_H(\mathcal{H})(\mathcal{U}_R(\mathcal{R})(\mathcal{U}_J(\mathcal{J})(\langle x \,\|\, k\rangle))) \\ &\quad \equiv \mathcal{U}_H(\mathcal{H}, x = K(\vec{\sigma}, \vec{v})) \\ &\qquad (\mathcal{U}_R(\mathcal{R})(\mathcal{U}_J(\mathcal{J})(\langle K(\vec{\sigma}, \vec{v}) \,\|\, k\rangle))) \\ &\quad \equiv (\mathcal{U}_H(\mathcal{H}) \circ \{K(\vec{\sigma}, \vec{v})/x\}) \\ &\qquad (\mathcal{U}_R(\mathcal{R})(\mathcal{U}_J(\mathcal{J})(\langle K(\vec{\sigma}, \vec{v}) \,\|\, k\rangle))) \\ &\quad \equiv (\mathcal{U}_H(\mathcal{H}) \circ \{K(\vec{\sigma}, \vec{v})/x\} \circ \overrightarrow{\{v/y\}}) \\ &\qquad (\mathcal{U}_R(\mathcal{R})(\mathcal{U}_J(\mathcal{J})(\langle K(\vec{\sigma}, \vec{y}) \,\|\, k\rangle))) \\ &\quad \equiv (\mathcal{U}_H(\mathcal{H}) \circ \overrightarrow{\{v/y\}} \circ \{K(\vec{\sigma}, \vec{y})/x\}) \\ &\qquad (\mathcal{U}_R(\mathcal{R})(\mathcal{U}_J(\mathcal{J})(\langle K(\vec{\sigma}, \vec{y}) \,\|\, k\rangle))) \\ &\quad \equiv \langle \mathcal{H}, \overline{y = v}, x = K(\vec{\sigma}, \vec{y});\ \mathcal{J}, \mathcal{R};\ \langle K(\vec{\sigma}, \vec{y}) \,\|\, k\rangle\rangle \end{aligned}$$

- (force)
Assuming that

$$x = \mu\mathsf{ret}.c \in \mathcal{H},$$

$$\begin{aligned} &\mathcal{U}(\langle \mathcal{H};\ \mathcal{J}, \mathcal{R};\ \langle x \,\|\, k\rangle\rangle) \\ &\quad \equiv \mathcal{U}_H(\mathcal{H})(\mathcal{U}_R(\mathcal{R})(\mathcal{U}_J(\mathcal{J})(\langle x \,\|\, k\rangle))) \\ &\quad \equiv \mathcal{U}_H(\mathcal{H})(\mathcal{U}_R(\mathcal{R})(\mathcal{U}_J(\mathcal{J})(\langle \mu\mathsf{ret}.c \,\|\, k\rangle))) \\ &\quad \equiv \mathcal{U}_H(\mathcal{H})(\mathcal{U}_R(\mathcal{R})(\langle \mu\mathsf{ret}.c \,\|\, \mathcal{U}_J(\mathcal{J})(k)\rangle)) \\ &\quad \to^\star \mathcal{U}_H(\mathcal{H})(\mathcal{U}_R(\mathcal{R})(c\, \{\mathcal{U}_J(\mathcal{J})(k)/\mathsf{ret}\})) \\ &\quad \equiv \mathcal{U}_H(\mathcal{H})(\mathcal{U}_R((k, \mathcal{J}) : \mathcal{R})(c)) \\ &\quad \equiv \mathcal{U}_H(\mathcal{H}, x = \bullet, x = \mu\mathsf{ret}.c)(\mathcal{U}_R((k, \mathcal{J}) : \mathcal{R})(c)) \\ &\quad \equiv \mathcal{U}_H(\mathcal{H}, x = \bullet)((\mathcal{U}_R((k, \mathcal{J}) : \mathcal{R})(c))\, \{\mu\mathsf{ret}.c/x\}) \\ &\quad \equiv \mathcal{U}_H(\mathcal{H}, x = \bullet)(\mathcal{U}_R(\mathbf{upd}\, x : (k, \mathcal{J}) : \mathcal{R})(c)) \\ &\quad \equiv \mathcal{U}_H(\mathcal{H}, x = \bullet)(\mathcal{U}_R(\mathbf{upd}\, x : (k, \mathcal{J}) : \mathcal{R})(\mathcal{U}_J(\varepsilon)(c))) \\ &\quad \equiv \mathcal{U}(\langle \mathcal{H}, x = \bullet;\ \varepsilon, \mathbf{upd}\, x : (k, \mathcal{J}) : \mathcal{R};\ c\rangle) \end{aligned}$$

- (update)

$$\begin{aligned} &\mathcal{U}(\langle \mathcal{H};\ \mathcal{J}, \mathbf{upd}\, x : \mathcal{R};\ \langle W \,\|\, \mathsf{ret}\rangle\rangle) \\ &\quad \equiv \mathcal{U}_H(\mathcal{H})(\mathcal{U}_R(\mathbf{upd}\, x : \mathcal{R})(\mathcal{U}_J(\mathcal{J})(\langle W \,\|\, \mathsf{ret}\rangle))) \\ &\quad \equiv \mathcal{U}_H(\mathcal{H})(\mathcal{U}_R(\mathcal{R})(\mathcal{U}_J(\mathcal{J})(\langle W \,\|\, \mathsf{ret}\rangle))\, \{\mu\mathsf{ret}.\langle W \,\|\, \mathsf{ret}\rangle/x\}) \\ &\quad \to^\star \mathcal{U}_H(\mathcal{H})(\mathcal{U}_R(\mathcal{R})(\mathcal{U}_J(\mathcal{J})(\langle W \,\|\, \mathsf{ret}\rangle))\, \{W/x\}) \\ &\quad \equiv (\mathcal{U}_H(\mathcal{H}) \circ \{W/x\})(\mathcal{U}_R(\mathcal{R})(\mathcal{U}_J(\mathcal{J})(\langle W \,\|\, \mathsf{ret}\rangle))) \\ &\quad \equiv \mathcal{U}(\langle \mathcal{H}, x = W;\ \mathcal{J}, \mathcal{R};\ \langle W \,\|\, \mathsf{ret}\rangle\rangle) \end{aligned}$$

- (ret)

$$\begin{aligned}
&\mathcal{U}(\langle \mathcal{H};\ \mathcal{J}, (k', \mathcal{J}') : \mathcal{R};\ \langle W \,\|\, \mathsf{ret} \rangle \rangle) \\
&\quad \equiv \mathcal{U}_H(\mathcal{H})(\mathcal{U}_R(\mathcal{R})(\mathcal{U}_J(\mathcal{J})(\langle W \,\|\, \mathsf{ret} \rangle))\ \{\mathcal{U}_J(\mathcal{J}')(k')/\mathsf{ret}\}) \\
&\quad \equiv \mathcal{U}_H(\mathcal{H})(\mathcal{U}_R(\mathcal{R})(\langle W \,\|\, \mathcal{U}_J(\mathcal{J}')(k') \rangle)) \\
&\quad \equiv \mathcal{U}_H(\mathcal{H})(\mathcal{U}_R(\mathcal{R})(\mathcal{U}_J(\mathcal{J}')(\langle W \,\|\, k' \rangle))) \\
&\quad \equiv \mathcal{U}(\langle \mathcal{H};\ \mathcal{J}', \mathcal{R};\ \langle W \,\|\, k' \rangle \rangle)
\end{aligned}$$

Note that in discarding $\mathcal{U}_J(\mathcal{J})$ above, we exploit the fact that W is continuation-closed, and hence the substitution $\mathcal{U}_J(\mathcal{J})$ was accomplishing nothing. Similarly, W is unaffected by $\mathcal{U}_J(\mathcal{J}')$, so we can move the latter out.

- $(\mathbf{let}_{val})$

$$\begin{aligned}
&\mathcal{U}(\langle \mathcal{H};\ \mathcal{J}, \mathcal{R};\ \mathbf{let}\, x = v \,\mathbf{in}\, c \rangle) \\
&\quad \equiv \mathcal{U}_H(\mathcal{H})(\mathcal{U}_R(\mathcal{R})(\mathcal{U}_J(\mathcal{J})(\mathbf{let}\, x = v \,\mathbf{in}\, c))) \\
&\quad \to^\star \mathcal{U}_H(\mathcal{H})(\mathcal{U}_R(\mathcal{R})(\mathcal{U}_J(\mathcal{J})(c\,\{v/x\}))) \\
&\quad \equiv (\mathcal{U}_H(\mathcal{H}) \circ \{v/x\})(\mathcal{U}_R(\mathcal{R})(\mathcal{U}_J(\mathcal{J})(c))) \\
&\quad \equiv \mathcal{U}(\langle \mathcal{H}, x = v;\ \mathcal{J}, \mathcal{R};\ c \rangle)
\end{aligned}$$

- Similarly for $\mathbf{let}_{cont}$. □

We will also require that the readback respects termination:

Lemma 16. *If $\langle \mathcal{H};\ \mathcal{J}, \mathcal{R};\ c \rangle \not\rightsquigarrow$, then $\mathcal{U}(\langle \mathcal{H};\ \mathcal{J}, \mathcal{R};\ c \rangle) \not\mapsto$.*

Proof. There are two forms of irreducible state in the call-by-need semantics: a missing $\mathbf{case}$ alternative and a WHNF passed to ret in an empty stack. The former reads back as a similarly stuck term, and the latter reads back as a WHNF. □

C.4 Bisimulation

Now we use the readback to define our bisimulation:

Definition 17. *Let $\sim$ relate call-by-name terms to call-by-need states, such that $c \sim \mathcal{S}$ when $c \to^\star \mathcal{U}(\mathcal{S})$.*

Lemma 18 (Bisimulation)**.** *Let $c \sim \mathcal{S}$.*

1. *If $c \mapsto c'$, then $\mathcal{S} \rightsquigarrow^\star \mathcal{S}'$ with $c' \sim \mathcal{S}'$.*
2. *If $\mathcal{S} \rightsquigarrow \mathcal{S}'$, then $c \mapsto^\star c'$ with $c' \sim \mathcal{S}'$.*

$$\begin{tikzcd}
c \arrow[r, mapsto] \arrow[d, "\star"] & c' \arrow[d, dashed, "\star"] & c \arrow[r, mapsto, dashed, "\star"] \arrow[d, "\star"] & c' \arrow[d, dashed, "\star"] \\
d \arrow[d, phantom, "\mathcal{U}"] & d' \arrow[d, phantom, "\mathcal{U}"] & d \arrow[d, phantom, "\mathcal{U}"] & d' \arrow[d, phantom, "\mathcal{U}"] \\
\mathcal{S} \arrow[r, squiggly, dashed, "\star"] & \mathcal{S}' & \mathcal{S} \arrow[r, squiggly] & \mathcal{S}'
\end{tikzcd}$$

Proof. 2 is a corollary of Lemma 14; we simply take $c' \equiv c$ so that we get $c \equiv c' \to^\star d \to^\star d'$.[11]

For 1, suppose $c \sim S$ and $c \mapsto c'$. By definition of $\sim$, we have $c \to^\star d \equiv \mathcal{U}(\mathcal{S})$:

$$\begin{tikzcd}
c \arrow[r, mapsto] \arrow[d, "\star"] & c' \\
d \arrow[d, phantom, "\mathcal{U}"] & \\
\mathcal{S} &
\end{tikzcd}$$

By Lemma 13, we have $d \mapsto^{0/1} d' \leftarrow^\star c'$:

$$\begin{tikzcd}
c \arrow[r, mapsto] \arrow[d, "\star"] & c' \arrow[d, dashed, "\star"] \\
d \arrow[r, mapsto, dashed, "0/1"] \arrow[d, phantom, "\mathcal{U}"] & d' \\
\mathcal{S} &
\end{tikzcd}$$

If $d \equiv d'$, then we can pick $\mathcal{S}' \equiv \mathcal{S}$ and we're done. Otherwise, assume $d \mapsto d'$. It will suffice to show that $\mathcal{S} \rightsquigarrow^\star \mathcal{S}'$ with $\mathcal{U}(\mathcal{S}') \equiv d'$:

$$\begin{tikzcd}
c \arrow[r, mapsto] \arrow[d, "\star"] & c' \arrow[d, "\star"] \\
d \arrow[r, mapsto] \arrow[d, phantom, "\mathcal{U}"] & d' \arrow[d, phantom, "\mathcal{U}"] \\
\mathcal{S} \arrow[r, squiggly, dashed, "\star"] & \mathcal{S}'
\end{tikzcd}$$

As noted before, $\mathcal{U}$ produces a substitution—$\mathcal{U}(\langle \mathcal{H};\ \mathcal{J}, \mathcal{R};\ c \rangle) \equiv c\sigma$ for some σ. In general, for a substitution σ, if $c\sigma \mapsto c'\sigma$, then at least one of the following is true:

1. $c \equiv \langle v \,\|\, k \rangle \mapsto c'$.
2. $c \equiv \mathsf{jump}\ j\ \vec{\sigma}\ \vec{v}$ and $j \in \operatorname{dom} \sigma$.
3. $c \equiv \langle v \,\|\, \mathsf{ret} \rangle$ and $\mathsf{ret} \in \operatorname{dom} \sigma$.
4. $c \equiv \langle x \,\|\, k \rangle$ and $x \in \operatorname{dom} \sigma$.

The last three cases may apply multiple times, but not infinitely many as bindings are not recursive (each substitution reduces the size of the context). Eventually we must land on case 1. Thus we may proceed by induction on the number of substitutions required to expose a redex.

In case 1, we have $c \equiv \langle v \,\|\, k \rangle \mapsto c'$. This means that $\mathcal{S} \equiv \langle \mathcal{H};\ \mathcal{J}, \mathcal{R};\ \langle v \,\|\, k \rangle \rangle$, and the substitution produced was not crucial to forming the redex. Therefore one of the "external" reduction rules—namely $\beta^{\to}$, $\beta^{\forall}$, $\mathbf{case}_{cons}$, $\mathbf{case}_{def}$, μ, $\mathbf{let}_{val}$, and $\mathbf{let}_{cont}$—must apply; each of them makes precisely the same substitutions as a corresponding call-by-name rule, only delaying some work by adding to $\mathcal{H}$, $\mathcal{J}$, or $\mathcal{R}$.

In case 2, the jump rule applies, and we apply the induction hypothesis. Similarly, case 3 is covered by some number of $\mathsf{updates}$ (each of which consumes an update frame) followed by a ret, and case 4 is covered by one of lookup, $\mathsf{lazysubst}$, and force. □

Lemma 19 (Bisimulation respects termination)**.** *Let $c \sim \mathcal{S}$.*

1. *If $c \not\mapsto$, then $\mathcal{S} \rightsquigarrow^\star \not\rightsquigarrow$.*
2. *If $\mathcal{S} \not\rightsquigarrow$, then $c \mapsto^\star \not\mapsto$.*

Proof. 1. By Lemma 5, $\mathcal{U}(\mathcal{S}) \not\mapsto$. From there, the case analysis is similar to that for Lemma 18, as internal reductions perform whatever substitutions are necessary for the stuck command to appear.
2. We know that $c \to^\star \mathcal{U}(\mathcal{S})$. By Lemma 16, we have that $\mathcal{U}(\mathcal{S}) \not\mapsto$; then the result holds by standardization (Proposition 8). □

The proposition is now reduced to a corollary:

Proof of Proposition 1. Since $c \sim \langle \varepsilon;\ \varepsilon, \varepsilon;\ c \rangle$, both directions follow directly from Lemmas 18 and 19 by induction on the reduction sequence. □

[11] Allowing $c \mapsto^\star c'$ is not necessary here, but it is *pro forma* for a bisimulation.

D. Proof of Type Safety (Proposition 2)

As is typical when proving type safety, we will require a lemma dealing with substitution and typing.

Lemma 20 (Substitution). *1. If $\Gamma \vdash \tau : \kappa$, then:*

(a) If $\Gamma, a : \kappa \vdash \sigma : \kappa'$, then $\Gamma\{\tau/a\} \vdash \sigma\{\tau/a\} : \kappa'$.

(b) If $\Gamma, a : \kappa \vdash v : \sigma$, then $\Gamma\{\tau/a\} \vdash v\{\tau/a\} : \sigma\{\tau/a\}$.

(c) If $\Gamma, a : \kappa \mid k : \sigma \vdash \Delta$, then $\Gamma\{\tau/a\} \mid k\{\tau/a\} : \sigma\{\tau/a\} \vdash \Delta\{\tau/a\}$.

(d) If $c : (\Gamma, a : \kappa \vdash \Delta)$, then $c\{\tau/a\} : (\Gamma\{\tau/a\} \vdash \Delta\{\tau/a\})$.

2. If $\Gamma \vdash v : \tau$, then:

(a) If $\Gamma, x : \tau \vdash v' : \sigma$, then $\Gamma \vdash v'\{v/x\} : \sigma$.

(b) If $\Gamma, x : \tau \mid k : \sigma \vdash \Delta$, then $\Gamma \mid k\{v/x\} : \sigma \vdash \Delta$.

(c) If $c : (\Gamma, x : \tau \vdash \Delta)$, then $c\{v/x\} : (\Gamma \vdash \Delta)$.

(d) For any k, $v \equiv v\{k/\mathsf{ret}\}$.

(e) For any c and j, $v \equiv v\left\{c\overrightarrow{\{\sigma/a\}}\overrightarrow{\{v'/x\}}/\mathsf{jump}\ j\ \vec{\sigma}\ \vec{v'}\right\}$.

3. If $\Gamma \mid k : \tau \vdash \Delta, \mathsf{ret} : \sigma$, then:

(a) If $\Gamma \mid k' : \sigma' \vdash \Delta, \mathsf{ret} : \tau$, then $\Gamma \mid k'\{k/\mathsf{ret}\} : \sigma' \vdash \Delta, \mathsf{ret} : \sigma$.

(b) If $c : (\Gamma \vdash \Delta, \mathsf{ret} : \tau)$, then $c\{k/\mathsf{ret}\} : (\Gamma \vdash \Delta, \mathsf{ret} : \sigma)$.

4. If $c : (\Gamma, \overrightarrow{a : \kappa}, \overrightarrow{x : \tau} \vdash \Delta)$, then:

(a) If $\Gamma \mid k : \sigma' \vdash \Delta, j : \exists\overrightarrow{a{:}\kappa}.(\vec{\tau})$, then
$\Gamma \mid k\left\{c\overrightarrow{\{\sigma/a\}}\overrightarrow{\{v/x\}}/\mathsf{jump}\ j\ \vec{\sigma}\ \vec{v}\right\} : \sigma' \vdash \Delta$.

(b) If $c' : (\Gamma \vdash \Delta, j : \exists\overrightarrow{a{:}\kappa}.(\vec{\tau}))$, then
$c'\left\{c\overrightarrow{\{\sigma/a\}}\overrightarrow{\{v/x\}}/\mathsf{jump}\ j\ \vec{\sigma}\ \vec{v}\right\} : (\Gamma \vdash \Delta)$.

Proof. A series of straightforward (if large) mutual inductions. 2(d) and 2(e) are trivial since well-typed terms have no free continuation variables. 4 relies on the first three to handle the structural substitution.

For example, in 4(b), for the case where $c' \equiv \mathsf{jump}\ j\ \vec{\sigma}\ \vec{v}$, we have

$$c'\left\{c\overrightarrow{\{\sigma/a\}}\overrightarrow{\{v/x\}}/\mathsf{jump}\ j\ \vec{\sigma}\ \vec{v}\right\} \equiv c\overrightarrow{\{\sigma/a\}}\overrightarrow{\{v/x\}}$$

The result then follows by (repeated applications of) 1(d) and 2(c). $\square$

Proof of Proposition 2. 1. A simple case analysis on c:

- If c is a **let**, one of the **let** rules applies.
- c cannot be a jump because its context has no join variables.
- Suppose $c \equiv \langle v \,\|\, k\rangle$. Then v is not a variable since it is closed. If it is a μ-abstraction, we can reduce no matter what k is. Otherwise, v is a WHNF; if k is ret, we are done, and otherwise either we can reduce or k is a stuck **case**.

2. An easy case analysis, applying Lemma 20 in each case. $\square$

E. Proof of Round-Trip Equivalence (Proposition 3)

E.1 Equational Reasoning

As is standard, we will avoid proving observational equivalence directly and instead rely on equational reasoning. To this end, we define equality in Sequent Core ($=$) as the reflexive-transitive-symmetric closure of $\to$ as defined in Section C.1. Note that the reduction theory of Sequent Core is confluent (here z and z' may be terms, continuations, or commands).

Proposition 21 (Confluence). *If $z_1 \leftarrow^{\star} z \rightarrow^{\star} z_2$ then there is a z' such that $z_1 \rightarrow^{\star} z' \leftarrow^{\star} z_2$.*

$$E \in EvalCxt ::= \square \mid E\ e \mid \mathbf{case}\ E\ \mathbf{of}\ \overrightarrow{alt}$$
$$W \in WHNF ::= \lambda x{:}\tau.e \mid \Lambda a{:}\kappa.e \mid x \mid K\,\vec{\sigma}\ \vec{e}$$

$$\begin{aligned}
(\lambda x{:}\tau.e)\ e' &\mapsto e\{e'/x\} \\
(\Lambda a{:}\kappa.e)\ \sigma &\mapsto e\{\sigma/a\} \\
\mathbf{case}\ K\vec{\sigma}\ \vec{e}\ \mathbf{of}\ \overrightarrow{alt} &\mapsto e'\overrightarrow{\{\sigma/b\}}\overrightarrow{\{e/x\}} && K\,\overrightarrow{b{:}\kappa}\ \overrightarrow{x{:}\tau} \to e' \in \overrightarrow{alt} \\
\mathbf{case}\ W\ \mathbf{of}\ \overrightarrow{alt} &\mapsto e'\{W/x\} && x \to e' \in \overrightarrow{alt} \\
\mathbf{let}\ x{:}\tau = e\ \mathbf{in}\ e' &\mapsto e'\{e/x\} \\
E[e] &\mapsto E[e'] && e \mapsto e'
\end{aligned}$$

Figure 12. Call-by-name operational semantics for Core

Proof. All reduction rules are left-linear, and there are no critical pairs. In particular, the only overlapping redexes are between μ reduction and μ_η reduction, but they lead to exactly the same result:

$$\langle \mu\mathsf{ret}.\, \langle v \,\|\, \mathsf{ret}\rangle \,\|\, k\rangle \to \langle v \,\|\, k\rangle$$

It then follows that the parallel reduction relation also defined in Section C.1 has the diamond property:

$$\begin{tikzcd}
 & z \arrow[dl, Rightarrow] \arrow[dr, Rightarrow] & \\
z_1 \arrow[dr, dashed, Rightarrow] & & z_2 \arrow[dl, dashed, Rightarrow] \\
 & z' &
\end{tikzcd}$$

And thus the single-step reduction relation is confluent. $\square$

Standardization (Proposition 8) and confluence (Proposition 21) then give us license to use equational reasoning to prove observational equivalence:

Proposition 22. *If $z = z'$, then $z \cong z'$.*

Proof. Suppose $z = z'$ and, without loss of generality for arbitrary C, $C[z] \mapsto^{\star} c \not\mapsto$. To show $z \cong z'$, we need to show that there is a c' such that $C[z'] \mapsto^{\star} c' \not\mapsto$. By confluence (Proposition 21), $z \rightarrow^{\star} z_1 \leftarrow^{\star} z'$, so since $\to$ is a congruence, $C[z] \rightarrow^{\star} C[z_1] \leftarrow^{\star} C[z']$.

$$\begin{tikzcd}
C[z] \arrow[r, mapsto, "\star"] \arrow[d, equal] \arrow[dr, dashed, "\star"] & c \not\mapsto \\
C[z'] \arrow[r, dashed, "\star"] & C[z_1]
\end{tikzcd}$$

Invoking confluence again, we get $c \rightarrow^{\star} c_1 \leftarrow^{\star} C[z_1]$. By Lemma 5, $c_1 \not\mapsto$. Now standardization (Proposition 8) gives us c' with $C[z'] \mapsto^{\star} c' \not\mapsto$.

$$\begin{tikzcd}
C[z] \arrow[rr, mapsto, "\star"] \arrow[dd, equal] \arrow[dr, "\star"] & & c \not\mapsto \arrow[dr, dashed, "\star"] & \\
 & C[z_1] \arrow[rr, dashed, "\star"] & & c_1 \not\mapsto \\
C[z'] \arrow[ur, "\star"] \arrow[rr, dashed, mapsto, "\star"] & & c' \not\mapsto \arrow[ur, dashed, "\star"] &
\end{tikzcd}$$ $\square$

In addition to the equational reasoning about Sequent Core terms (and commands and continuations), we will also need to reason equationally about Core terms to establish the round-trip equivalence. The theory for Core equations is built up in the same way as we did for Sequent Core. In particular, we equip Core with a standard call-by-name operational semantics ($\mapsto$), with the basic single-step rules and compatible closure under evaluation contexts illustrated in Figure 12. For the general reduction of Core expressions ($\to$), we take the compatible closure of the single-step operational relation ($\mapsto$) along with the additional rule for performing a generalized case-of-case:

$$E[\mathbf{case}\, e' \,\mathbf{of}\, \overrightarrow{pat \to e}] \to \mathbf{case}\, e' \,\mathbf{of}\, \overrightarrow{pat \to E[e]}$$

As before, we write the reflexive-transitive closures of $\mapsto$ and $\to$ for Core as $\mapsto^\star$ and $\to^\star$, respectively, and the reflexive-transitive-symmetric closure of $\to$ as $=$. Note that other forms of commutative conversions besides the generalized case-of-case hold up to equational reasoning due to other steps from the operational semantics, including:

$$\begin{aligned} E[\mathbf{let}\, x{:}\tau = e' \,\mathbf{in}\, e] &= \mathbf{let}\, x{:}\tau = e' \,\mathbf{in}\, E[e] \\ E[(\lambda x{:}\tau.e)\ e'] &= (\lambda x{:}\tau.E[e])\ e' \\ E[(\Lambda a{:}\kappa.e)\ \sigma] &= (\Lambda a{:}\kappa.E[e])\ \sigma \\ E[\mathbf{case}\, e' \,\mathbf{of}\, \overrightarrow{pat \to e}] &= \mathbf{case}\, e' \,\mathbf{of}\, \overrightarrow{pat \to E[e]} \end{aligned}$$

The first and the last equations in particular will be useful for reflecting the μ-reduction of Sequent Core back into Core.

Just like with Sequent Core, the standard semantics of Core enjoys both confluence and standardization. Therefore equational reasoning in Core is a valid method of establishing an observational equivalence in Core.

Proposition 23 (Confluence of Core)**.** *If $e_1 \leftarrow^\star e \to^\star e_2$ then there is an e' such that $e_1 \to^\star e' \leftarrow^\star e_2$.*

Proposition 24 (Standardization of Core)**.** *If $e \to^\star e' \not\mapsto^\star$ then $e \to^\star e_1 \not\mapsto^\star$ and $e_1 \to^\star e'$.*

Proposition 25. *If $e = e'$ then $e \cong e'$.*

Proof. The same reasoning as for Proposition 22, except for using confluence (Proposition 23) and standardization (Proposition 24) for Core instead of for Sequent Core. □

E.2 Proof

For simplicity, we will prove round-trip equivalence for the compositional translation S rather than the administrative-free translation S_a. In other words, we will use the following fact:

Proposition 26. $S\,[\![e]\!] = S_a\,[\![e]\!]$.

Proof. Note that the S_a transformation has two forms on expressions:

$$S_a\,[\![e]\!] \qquad\qquad S_a\,[\![e]\!]\ k$$

It can be shown simultaneously that both $S\,[\![e]\!] = S_a\,[\![e]\!]$ and $\langle S\,[\![e]\!] \,\|\, k\rangle = S_a\,[\![e]\!]\ k$ hold by induction on the Core expression e. The most common difference between $S\,[\![e]\!]$ and $S_a\,[\![e]\!]$ is that $S\,[\![e]\!]$ μ-reduces to $S_a\,[\![e]\!]$. For example, in the case where $e \equiv e_1\ e_2$, we have:

$$\begin{aligned} \langle S\,[\![e_1\ e_2]\!] \,\|\, k\rangle &\equiv \langle \mu\mathsf{ret}.\, \langle S\,[\![e_1]\!] \,\|\, S\,[\![e_2]\!] \cdot \mathsf{ret}\rangle \,\|\, k\rangle \\ &= \langle S\,[\![e_1]\!] \,\|\, S\,[\![e_2]\!] \cdot k\rangle \\ &= S_a\,[\![e_1]\!]\ (S_a\,[\![e_2]\!] \cdot k) & \text{(by I.H.)} \\ &\equiv S_a\,[\![e_1\ e_2]\!]\ k \end{aligned}$$

The only other difference to account for is the shrink operation, which is undone by inlining the created **let** bindings. □

To make full use of this equivalence, we need to know that D preserves this equality, at least for programs without join points. (Join points can be accommodated, but it would complicate the proof.)

Lemma 27. *In the join-point-free fragment:*

1. *If $v = v'$, then $D\,[\![v]\!] = D\,[\![v']\!]$.*
2. *If $k = k'$ and $e = e'$, then $D\,[\![k]\!]\ [e] = D\,[\![k']\!]\ [e']$.*
3. *If $c = c'$, then $D\,[\![c]\!] = D\,[\![c']\!]$.*

Proof. By mutual induction on the derivation of $=$. Because the D translation is compositional and hygienic (it does not cause escape or capture of static variables), it suffices to show that each reduction rule is preserved. Crucially, we must deal with how translation interacts with continuation substitution. We claim:

$$\begin{aligned} D\,[\![c\,\{k/\mathsf{ret}\}]\!] &= D\,[\![k]\!]\ [D\,[\![c]\!]] \\ D\,[\![k'\,\{k/\mathsf{ret}\}]\!]\ [e] &= D\,[\![k]\!]\ [D\,[\![k']\!]\ [e]] \end{aligned}$$

The claim is proved by mutual induction. Important cases:

- For $c \equiv \langle v \,\|\, k'\rangle$:

$$\begin{aligned} &D\,[\![\langle v \,\|\, k'\rangle\,\{k/\mathsf{ret}\}]\!] \\ &\quad\equiv D\,[\![k'\,\{k/\mathsf{ret}\}]\!]\ [D\,[\![v]\!]] \\ &\quad= D\,[\![k]\!]\ [D\,[\![k']\!]\ [D\,[\![v]\!]]] & \text{(by I.H.)} \\ &\quad\equiv D\,[\![k]\!]\ [D\,[\![\langle v \,\|\, k'\rangle]\!]] \end{aligned}$$

- For $k' \equiv v \cdot k''$:

$$\begin{aligned} &D\,[\![(v \cdot k'')\,\{k/\mathsf{ret}\}]\!]\ [e] \\ &\quad\equiv D\,[\![v \cdot k''\,\{k/\mathsf{ret}\}]\!]\ [e] \\ &\quad\equiv D\,[\![k''\,\{k/\mathsf{ret}\}]\!]\ [\square\ D\,[\![v]\!]][e] \\ &\quad\equiv D\,[\![k''\,\{k/\mathsf{ret}\}]\!]\ [e\ D\,[\![v]\!]] \\ &\quad= D\,[\![k]\!]\ [D\,[\![k'']\!]\ [e\ D\,[\![v]\!]]] & \text{(by I.H.)} \\ &\quad\equiv D\,[\![k]\!]\ [D\,[\![k'']\!]\ [\square\ D\,[\![v]\!]][e]] \\ &\quad\equiv D\,[\![k]\!]\ [D\,[\![v \cdot k'']\!]\ [e]] \end{aligned}$$

- For $k' \equiv \mathbf{case\,of}\, \overrightarrow{pat \to c}$ (letting p stand for a pattern, which may be a default pattern):

$$\begin{aligned} &D\,[\![\mathbf{case\,of}\, \overrightarrow{pat \to c}\,\{k/\mathsf{ret}\}]\!]\ [e] \\ &\quad\equiv (\mathbf{case}\,\square\,\mathbf{of}\, \overrightarrow{pat \to D\,[\![c]\!])\,\{k/\mathsf{ret}\}}[e] \\ &\quad\equiv \mathbf{case}\,\square\,\mathbf{of}\, \overrightarrow{pat \to D\,[\![c\,\{k/\mathsf{ret}\}]\!]}[e] \\ &\quad\equiv \mathbf{case}\,e\,\mathbf{of}\, \overrightarrow{pat \to D\,[\![c\,\{k/\mathsf{ret}\}]\!]} \\ &\quad= \mathbf{case}\,e\,\mathbf{of}\, \overrightarrow{pat \to D\,[\![k]\!]\ [D\,[\![c]\!]]} & \text{(by I.H.)} \\ &\quad\leftarrow D\,[\![k]\!]\ [\mathbf{case}\,e\,\mathbf{of}\, \overrightarrow{pat \to D\,[\![c]\!]}] \\ &\quad\equiv D\,[\![k]\!]\ [(\mathbf{case}\,\square\,\mathbf{of}\, \overrightarrow{pat \to D\,[\![c]\!]})[e]] \\ &\quad\equiv D\,[\![k]\!]\ [D\,[\![\mathbf{case\,of}\, \overrightarrow{pat \to c}]\!]\ [e]] \end{aligned}$$

  Note that we have made use of the extra reduction rule to perform the case-of-case transform.

With the claim proved, we can handle μ-reduction. If $c \mapsto c'$ by μ, then $c \equiv \langle \mu\mathsf{ret}.c'' \,\|\, k\rangle$ and $c' \equiv c''\,\{k/\mathsf{ret}\}$. Then:

$$\begin{aligned} &D\,[\![\langle \mu\mathsf{ret}.c'' \,\|\, k\rangle]\!] \\ &\quad\equiv D\,[\![k]\!]\ [D\,[\![\mu\mathsf{ret}.c'']\!]] \\ &\quad\equiv D\,[\![k]\!]\ [D\,[\![c'']\!]] \\ &\quad= D\,[\![c''\,\{k/\mathsf{ret}\}]\!] \end{aligned}$$

The other cases of reduction are straightforward. $\square$

Now we are prepared to show that $D\left[\!\left[S_a\left[\!\left[e\right]\!\right]\right]\!\right] = e$. To show the other direction, that $S_a\left[\!\left[D\left[\!\left[v\right]\!\right]\right]\!\right] = v$, we must deal with the erasure of join points—since our direct-style language has no join points, we translate them back as ordinary functions. We can describe the effect this has in terms of the sequent calculus; this will simplify the proofs greatly.

Definition 28. *Define the* decontification *function* $V\left[\!\left[-\right]\!\right]$ *as homomorphic on all syntax except*

$$V\left[\!\left[j = \tilde{\mu}[\vec{a}, \vec{x}].c\right]\!\right] \equiv j = \overrightarrow{\Lambda a.}\overrightarrow{\lambda x.}\mu\mathsf{ret}.V\left[\!\left[c\right]\!\right]$$

and

$$V\left[\!\left[\mathsf{jump}\ j\ \vec{\sigma}\ \vec{v}\right]\!\right] \equiv \left\langle j \,\middle\|\, \vec{\sigma}\cdot\overrightarrow{V\left[\!\left[v\right]\!\right]}\cdot\mathsf{ret}\right\rangle.$$

Decontification is purely syntactic—it does not affect the observable behavior of the program.

Lemma 29. *For all v, k, and c with no free continuation variables,*

1. $V\left[\!\left[v\right]\!\right] = v$,
2. $V\left[\!\left[k\right]\!\right] = k$, *and*
3. $V\left[\!\left[c\right]\!\right] = c$.

Proof. To investigate the effect of structural substitution versus substitution of a decontified function, we will need a version of V that only decontifies *some* variables. Hence for each set of variables ρ, let V_ρ be homomorphic on all syntax except that

$$V_\rho\left[\!\left[\mathbf{let}\, j = \tilde{\mu}[\vec{a}, \vec{x}].c\,\mathbf{in}\, c'\right]\!\right] \equiv \mathbf{let}\, j = \overrightarrow{\Lambda a.}\overrightarrow{\lambda x.}\mu\mathsf{ret}.V_\rho\left[\!\left[c\right]\!\right]\,\mathbf{in}\, V_{\rho\cup\{j\}}\left[\!\left[c'\right]\!\right]$$

and

$$V_\rho\left[\!\left[\mathsf{jump}\ j\ \vec{\sigma}\ \vec{v}\right]\!\right] \equiv \begin{cases}\left\langle j \,\middle\|\, \vec{\sigma}\cdot\overrightarrow{V_\rho\left[\!\left[v\right]\!\right]}\cdot\mathsf{ret}\right\rangle, & j\in\rho\\ \mathsf{jump}\ j\ \vec{\sigma}\ \overrightarrow{V_\rho\left[\!\left[v\right]\!\right]}, & j\notin\rho.\end{cases}$$

It is obvious that

$$V\left[\!\left[z\right]\!\right] \equiv V_{\mathrm{fjv}(z)}\left[\!\left[z\right]\!\right]$$

for any term, continuation, or command z, where fjv gives the free join variables in its argument.

Now we can characterize the interaction of V with structural substitution. Supposing that $j\notin\rho$ and $\mathrm{fjv}(c')\subseteq\rho$, we claim:

$$V_\rho\left[\!\left[c\right]\!\right]\left\{V_\rho\left[\!\left[c'\right]\!\right]\overrightarrow{\{\sigma/a\}}\overrightarrow{\{v/x\}}/\mathsf{jump}\ j\ \vec{\sigma}\ \vec{v}\right\} = V_{\rho\cup\{j\}}\left[\!\left[c\right]\!\right]\left\{\overrightarrow{\Lambda a.}\overrightarrow{\lambda x.}\mu\mathsf{ret}.V_\rho\left[\!\left[c'\right]\!\right]/j\right\}$$

Note that we have tacitly made use of the fact that $V_\rho[\![c']\!]$ has no free join variables (since c' has only ρ as free join variables) so that $\mu\mathsf{ret}.V_\rho[\![c']\!]$ is well-typed.

The claim is proved by induction; the interesting case is this:

$$\begin{aligned}
&V_\rho\left[\!\left[\mathsf{jump}\ j\ \vec{\sigma}\ \vec{v}\right]\!\right]\left\{V_\rho\left[\!\left[c'\right]\!\right]\overrightarrow{\{\sigma/a\}}\overrightarrow{\{v/x\}}/\mathsf{jump}\ j\ \vec{\sigma}\ \vec{v}\right\}\\
&\quad\equiv (\mathsf{jump}\ j\ \vec{\sigma}\ \overrightarrow{V_\rho\left[\!\left[v\right]\!\right]})\left\{V_\rho\left[\!\left[c'\right]\!\right]\overrightarrow{\{\sigma/a\}}\overrightarrow{\{v/x\}}/\mathsf{jump}\ j\ \vec{\sigma}\ \vec{v}\right\}\\
&\quad\equiv V_\rho\left[\!\left[c'\right]\!\right]\overrightarrow{\{\sigma/a\}}\overrightarrow{\{V_\rho\left[\!\left[v\right]\!\right]/x\}}\\
&\quad\leftarrow^\star \left\langle\overrightarrow{\Lambda a.}\overrightarrow{\lambda x.}\mu\mathsf{ret}.V_\rho\left[\!\left[c'\right]\!\right] \,\middle\|\, \vec{\sigma}\cdot\overrightarrow{V_\rho\left[\!\left[v\right]\!\right]}\cdot\mathsf{ret}\right\rangle\\
&\quad\equiv \left\langle j \,\middle\|\, \vec{\sigma}\cdot\overrightarrow{V_\rho\left[\!\left[v\right]\!\right]}\cdot\mathsf{ret}\right\rangle\left\{\overrightarrow{\Lambda a.}\overrightarrow{\lambda x.}\mu\mathsf{ret}.V_\rho\left[\!\left[c'\right]\!\right]/j\right\}\\
&\quad\equiv \left\langle j \,\middle\|\, \vec{\sigma}\cdot\overrightarrow{V_{\rho\cup\{j\}}\left[\!\left[v\right]\!\right]}\cdot\mathsf{ret}\right\rangle\left\{\overrightarrow{\Lambda a.}\overrightarrow{\lambda x.}\mu\mathsf{ret}.V_\rho\left[\!\left[c'\right]\!\right]/j\right\}\\
&\quad\equiv V_{\rho\cup\{j\}}\left[\!\left[\mathsf{jump}\ j\ \vec{\sigma}\ \vec{v}\right]\!\right]\left\{\overrightarrow{\Lambda a.}\overrightarrow{\lambda x.}\mu\mathsf{ret}.V_\rho\left[\!\left[c'\right]\!\right]/j\right\}
\end{aligned}$$

Note that the penultimate step makes use of the fact that the subscript of V does not matter when operating on terms, since terms have no free join variables.

Now we can prove the lemma by mutual induction, generalizing each clause by replacing V with V_ρ. The crucial case is in clause 3:

$$\begin{aligned}
&V_\rho\left[\!\left[\mathbf{let}\, j = \tilde{\mu}[\vec{a}, \vec{x}].c\,\mathbf{in}\, c'\right]\!\right]\\
&\quad\equiv \mathbf{let}\, j = \overrightarrow{\Lambda a.}\overrightarrow{\lambda x.}\mu\mathsf{ret}.V_\rho\left[\!\left[c\right]\!\right]\,\mathbf{in}\, V_{\rho\cup\{j\}}\left[\!\left[c'\right]\!\right]\\
&\quad\to V_{\rho\cup\{j\}}\left[\!\left[c'\right]\!\right]\left\{\overrightarrow{\Lambda a.}\overrightarrow{\lambda x.}\mu\mathsf{ret}.V_\rho\left[\!\left[c\right]\!\right]/j\right\}\\
&\quad= V_\rho\left[\!\left[c'\right]\!\right]\left\{V_\rho\left[\!\left[c\right]\!\right]\overrightarrow{\{\sigma/a\}}\overrightarrow{\{v/x\}}/\mathsf{jump}\ j\ \vec{\sigma}\ \vec{v}\right\}\\
&\quad= c'\left\{c\overrightarrow{\{\sigma/a\}}\overrightarrow{\{v/x\}}/\mathsf{jump}\ j\ \vec{\sigma}\ \vec{v}\right\} && \text{(by I.H.)}\\
&\quad\leftarrow \mathbf{let}\, j = \tilde{\mu}[\vec{a}, \vec{x}].c\,\mathbf{in}\, c'
\end{aligned}$$

$\square$

Decontification also does not affect how the program translates back to direct style.

Lemma 30. *1.* $D\left[\!\left[V\left[\!\left[v\right]\!\right]\right]\!\right] \equiv D\left[\!\left[v\right]\!\right]$.
2. $D\left[\!\left[V\left[\!\left[k\right]\!\right]\right]\!\right] \equiv D\left[\!\left[k\right]\!\right]$.
3. $D\left[\!\left[V\left[\!\left[c\right]\!\right]\right]\!\right] \equiv D\left[\!\left[c\right]\!\right]$.

Proof. Easy induction. For example:

$$\begin{aligned}
&D\left[\!\left[V\left[\!\left[\mathsf{jump}\ j\ \vec{\sigma}\ \vec{v}\right]\!\right]\right]\!\right]\\
&\quad\equiv D\left[\!\left[\langle j \,\|\, \vec{\sigma}\cdot\vec{v}\cdot\mathsf{ret}\rangle\right]\!\right]\\
&\quad\equiv j\ \vec{\sigma}\ \vec{v}\\
&\quad\equiv D\left[\!\left[\mathsf{jump}\ j\ \vec{\sigma}\ \vec{v}\right]\!\right]
\end{aligned}$$

$\square$

Now for the meat of the proof. Decontification will free us from having to consider join points when translating to direct style and back.

Lemma 31. *1.* $D\left[\!\left[S\left[\!\left[e\right]\!\right]\right]\!\right] \equiv e$.
2. (a) $S\left[\!\left[D\left[\!\left[v\right]\!\right]\right]\!\right] = v$.
(b) $S\left[\!\left[D\left[\!\left[k\right]\!\right]\left[e\right]\right]\!\right] = \mu\mathsf{ret}.\left\langle S\left[\!\left[e\right]\!\right] \,\middle\|\, k\right\rangle$.
(c) $S\left[\!\left[D\left[\!\left[c\right]\!\right]\right]\!\right] = \mu\mathsf{ret}.c$.

Proof. 1. By induction on e:

- All cases where e is a value are trivial.
- For $e \equiv \mathbf{let}\, \mathit{bind}\,\mathbf{in}\, e$:

$$\begin{aligned}
&D\left[\!\left[S\left[\!\left[\mathbf{let}\, \mathit{bind}\,\mathbf{in}\, e\right]\!\right]\right]\!\right]\\
&\quad\equiv D\left[\!\left[\mu\mathsf{ret}.\,\mathbf{let}\, S\left[\!\left[\mathit{bind}\right]\!\right]\,\mathbf{in}\,\left\langle S\left[\!\left[e\right]\!\right] \,\middle\|\, \mathsf{ret}\right\rangle\right]\!\right]\\
&\quad\equiv \mathbf{let}\, D\left[\!\left[S\left[\!\left[\mathit{bind}\right]\!\right]\right]\!\right]\,\mathbf{in} (D\left[\!\left[\mathsf{ret}\right]\!\right])[D\left[\!\left[S\left[\!\left[e\right]\!\right]\right]\!\right]]\\
&\quad\equiv \mathbf{let}\, D\left[\!\left[S\left[\!\left[\mathit{bind}\right]\!\right]\right]\!\right]\,\mathbf{in}\, D\left[\!\left[S\left[\!\left[e\right]\!\right]\right]\!\right]\\
&\quad\equiv \mathbf{let}\, \mathit{bind}\,\mathbf{in}\, e && \text{(by I.H.)}
\end{aligned}$$

  We used in passing the fact that $D\left[\!\left[S\left[\!\left[\mathit{bind}\right]\!\right]\right]\!\right] \equiv \mathit{bind}$, which (under the induction hypothesis) is obvious in both cases of bind.
- For $e \equiv e'\ e''$:

$$\begin{aligned}
&D\left[\!\left[S\left[\!\left[e'\ e''\right]\!\right]\right]\!\right]\\
&\quad\equiv D\left[\!\left[\mu\mathsf{ret}.\left\langle S\left[\!\left[e'\right]\!\right] \,\middle\|\, S\left[\!\left[e''\right]\!\right]\cdot\mathsf{ret}\right\rangle\right]\!\right]\\
&\quad\equiv D\left[\!\left[S\left[\!\left[e''\right]\!\right]\cdot\mathsf{ret}\right]\!\right]\left[D\left[\!\left[S\left[\!\left[e'\right]\!\right]\right]\!\right]\right]\\
&\quad\equiv (D\left[\!\left[\mathsf{ret}\right]\!\right][\square\ D\left[\!\left[S\left[\!\left[e''\right]\!\right]\right]\!\right]])[D\left[\!\left[S\left[\!\left[e'\right]\!\right]\right]\!\right]]\\
&\quad\equiv (\square\ D\left[\!\left[S\left[\!\left[e''\right]\!\right]\right]\!\right])[D\left[\!\left[S\left[\!\left[e'\right]\!\right]\right]\!\right]]\\
&\quad\equiv D\left[\!\left[S\left[\!\left[e'\right]\!\right]\right]\!\right]\ D\left[\!\left[S\left[\!\left[e''\right]\!\right]\right]\!\right]\\
&\quad\equiv e'\ e'' && \text{(by I.H.)}
\end{aligned}$$

- The case for $e \equiv e'\ \tau$ is similar.
- For $e \equiv \mathbf{case}\, e'\, \mathbf{of}\, \overrightarrow{alt}$:

$$
\begin{aligned}
&D\left\llbracket S\left\llbracket \mathbf{case}\, e'\, \mathbf{of}\, \overrightarrow{alt}\right\rrbracket\right\rrbracket\\
&\quad\equiv D\left\llbracket \mu\mathsf{ret}.\left\langle S\llbracket e'\rrbracket \,\middle\|\, \mathbf{case\,of}\, \overrightarrow{S\llbracket alt\rrbracket}\right\rangle\right\rrbracket\\
&\quad\equiv D\left\llbracket \mathbf{case\,of}\, \overrightarrow{S\llbracket alt\rrbracket}\right\rrbracket\, [D\llbracket S\llbracket e'\rrbracket\rrbracket]\\
&\quad\equiv (\mathbf{case}\,\square\,\mathbf{of}\, \overrightarrow{D\llbracket S\llbracket alt\rrbracket\rrbracket})[D\llbracket S\llbracket e'\rrbracket\rrbracket]\\
&\quad\equiv \mathbf{case}\, D\llbracket S\llbracket e'\rrbracket\rrbracket\, \mathbf{of}\, \overrightarrow{D\llbracket S\llbracket alt\rrbracket\rrbracket}\\
&\quad\equiv \mathbf{case}\, e'\, \mathbf{of}\, \overrightarrow{alt} && \text{(by I.H.)}
\end{aligned}
$$

As with bindings, it is obvious that $D\llbracket S\llbracket alt\rrbracket\rrbracket \equiv alt$.

2. We can assume without loss of generality that we're in the join-point-free fragment of the language, since then by Lemmas 29 and 30, we will have

$$S\llbracket D\llbracket v\rrbracket\rrbracket \equiv S\llbracket D\llbracket V\llbracket v\rrbracket\rrbracket\rrbracket = V\llbracket v\rrbracket = v$$

(and similar statements for continuations and commands). Thus proceed by mutual induction on v, k, and c, assuming that none of them contain join points.

(a)
- All cases where v is a variable or WHNF are trivial.
- For $v \equiv \mu\mathsf{ret}.c$:

$$
\begin{aligned}
&S\llbracket D\llbracket \mu\mathsf{ret}.c\rrbracket\rrbracket\\
&\quad\equiv S\llbracket D\llbracket c\rrbracket\rrbracket\\
&\quad= \mu\mathsf{ret}.c && \text{(by (c))}
\end{aligned}
$$

(b)
- For $k \equiv \mathsf{ret}$:

$$
\begin{aligned}
&S\llbracket D\llbracket \mathsf{ret}\rrbracket\,[e]\rrbracket\\
&\quad\equiv S\llbracket \square[e]\rrbracket\\
&\quad\equiv S\llbracket e\rrbracket\\
&\quad= \mu\mathsf{ret}.\left\langle S\llbracket e\rrbracket \,\middle\|\, \mathsf{ret}\right\rangle
\end{aligned}
$$

- For $k \equiv v \cdot k'$:

$$
\begin{aligned}
&S\llbracket D\llbracket v\cdot k'\rrbracket\,[e]\rrbracket\\
&\quad\equiv S\llbracket D\llbracket k'\rrbracket\,[\square\ D\llbracket v\rrbracket][e]\rrbracket\\
&\quad\equiv S\llbracket D\llbracket k'\rrbracket\,[e\ D\llbracket v\rrbracket]\rrbracket\\
&\quad= \mu\mathsf{ret}.\left\langle S\llbracket e\ D\llbracket v\rrbracket\rrbracket \,\middle\|\, k'\right\rangle && \text{(by I.H.)}\\
&\quad\equiv \mu\mathsf{ret}.\left\langle \mu\mathsf{ret}.\left\langle S\llbracket e\rrbracket \,\middle\|\, S\llbracket D\llbracket v\rrbracket\rrbracket\cdot\mathsf{ret}\right\rangle \,\middle\|\, k'\right\rangle\\
&\quad\to \mu\mathsf{ret}.\left\langle S\llbracket e\rrbracket \,\middle\|\, S\llbracket D\llbracket v\rrbracket\rrbracket\cdot k'\right\rangle\\
&\quad= \mu\mathsf{ret}.\left\langle S\llbracket e\rrbracket \,\middle\|\, v\cdot k'\right\rangle && \text{(by (a))}
\end{aligned}
$$

- The case for $k \equiv \tau\cdot k'$ is similar.
- For $k = \mathbf{case\,of}\, \overrightarrow{alt}$:

$$
\begin{aligned}
&S\left\llbracket D\left\llbracket \mathbf{case\,of}\,\overrightarrow{alt}\right\rrbracket\,[e]\right\rrbracket\\
&\quad\equiv S\left\llbracket (\mathbf{case}\,\square\,\mathbf{of}\,\overrightarrow{D\llbracket alt\rrbracket})[e]\right\rrbracket\\
&\quad\equiv S\left\llbracket \mathbf{case}\, e\, \mathbf{of}\,\overrightarrow{D\llbracket alt\rrbracket}\right\rrbracket\\
&\quad\equiv \mu\mathsf{ret}.\left\langle S\llbracket e\rrbracket \,\middle\|\, \mathbf{case\,of}\,\overrightarrow{S\llbracket D\llbracket alt\rrbracket\rrbracket}\right\rangle\\
&\quad= \mu\mathsf{ret}.\left\langle S\llbracket e\rrbracket \,\middle\|\, \mathbf{case\,of}\,\overrightarrow{alt}\right\rangle && \text{(by (c))}
\end{aligned}
$$

(c)
- For $c \equiv \mathbf{let}\ bind\ \mathbf{in}\ c$:

$$
\begin{aligned}
&S\llbracket D\llbracket \mathbf{let}\ bind\ \mathbf{in}\ c\rrbracket\rrbracket\\
&\quad\equiv S\llbracket \mathbf{let}\ D\llbracket bind\rrbracket\ \mathbf{in}\ D\llbracket c\rrbracket\rrbracket\\
&\quad\equiv \mathbf{let}\ S\llbracket D\llbracket bind\rrbracket\rrbracket\ \mathbf{in}\ S\llbracket D\llbracket c\rrbracket\rrbracket\\
&\quad= \mathbf{let}\ bind\ \mathbf{in}\ c && \text{(by (a), I.H.)}
\end{aligned}
$$

In the last step, we use the assumption that there are no join points, and thus the binding is a value binding.

- For $c \equiv \langle v \,\|\, k\rangle$:

$$
\begin{aligned}
&S\llbracket D\llbracket \langle v\,\|\,k\rangle\rrbracket\rrbracket\\
&\quad\equiv S\llbracket D\llbracket k\rrbracket\,[D\llbracket v\rrbracket]\rrbracket\\
&\quad= \mu\mathsf{ret}.\left\langle S\llbracket D\llbracket v\rrbracket\rrbracket \,\middle\|\, k\right\rangle && \text{(by (b))}\\
&\quad= \mu\mathsf{ret}.\langle v\,\|\,k\rangle && \text{(by (a))}
\end{aligned}
$$

- The case where c is a jump is impossible by assumption. □

Proof of Proposition 3. From Proposition 26 and Lemma 31 we get $S_a\llbracket D\llbracket v\rrbracket\rrbracket = S\llbracket D\llbracket v\rrbracket\rrbracket = v$. From Proposition 26, Lemma 27, and Lemma 31 we get $D\llbracket S_a\llbracket e\rrbracket\rrbracket = D\llbracket S\llbracket e\rrbracket\rrbracket = e$. Finally, from Propositions 22 and 25 we have $S_a\llbracket D\llbracket v\rrbracket\rrbracket \cong v$ and $D\llbracket S_a\llbracket e\rrbracket\rrbracket \cong e$. □

F. Proof of Well-Typed Translation (Proposition 4)

To show that the translations between Core and Sequent Core are well-typed, we need to refer to the type system for Core, which is illustrated in Figure 13. Notice that, except for lacking the type for jumps, Core has exactly the same rules for determining the kinds of types as Sequent Core from Figure 4.

We already have that S_a is equivalent to S (Proposition 26). We can make use of this to prove type safety of S_a from S by extending type preservation:

Proposition 32 (Preservation under $=$)**.** *If $\Gamma_1 \vdash v_1 : \tau_1$, $\Gamma_2 \vdash v_2 : \tau_2$, and $v_1 = v_2$, then $\tau_1 \equiv \tau_2$.*

Proof. Uniqueness of types (i.e., the case where $v_1 \equiv v_2$) is obvious, since the typing rules are syntax-directed. Thus if we find v with $v_1 \to^\star v \leftarrow^\star v_2$, we are done, since type preservation (Proposition 2)[12] says that v has the same type as both v_1 and v_2. But confluence gives us exactly such a v. □

Proving type safety of S is now straightforward.

Lemma 33 (Type safety of S)**.**
If $\Gamma \vdash e : \tau$ in Core, then $\Gamma \vdash S\llbracket e\rrbracket : \tau$ in Sequent Core.

Proof. An easy induction on the typing derivation. For example, to handle term application, suppose we have:

$$
\dfrac{\begin{matrix}\mathcal{D}\\ \vdots\\ \Gamma \vdash e : \sigma \to \tau\end{matrix} \qquad \begin{matrix}\mathcal{E}\\ \vdots\\ \Gamma \vdash e' : \sigma\end{matrix}}{\Gamma \vdash e\ e' : \tau}\ {\to}\mathsf{E}
$$

[12] In fact, we need to extend type preservation from $\mapsto$ to $\to$, but this is trivial since $\to$ adds only the μ_η-rule (easily verified) and compatibility, and our type system is compositional.

$$\Gamma \in \mathit{Environment} ::= \varepsilon \mid \Gamma, x : \tau \mid \Gamma, a : \kappa \mid \Gamma, K : \tau \mid \Gamma, T : \kappa$$

Type kinding: $\boxed{\Gamma \vdash \tau : \kappa}$

$$\frac{}{\Gamma, a : \kappa \vdash a : \kappa}\ \textsf{TyVar} \qquad \frac{}{\Gamma, T : \kappa \vdash T : \kappa}\ \textsf{TyCon} \qquad \frac{\Gamma \vdash \sigma : \kappa' \to \kappa \quad \Gamma \vdash \tau : \kappa'}{\Gamma \vdash \sigma\, \tau : \kappa}\ \textsf{TyApp} \qquad \frac{\Gamma, a : \kappa \vdash \tau : \star}{\Gamma \vdash \forall a{:}\kappa.\tau : \star}\ \forall$$

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

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ \textsf{Var} \qquad \frac{\Gamma \vdash \mathit{bind} : \{\Gamma'\} \quad \Gamma, \Gamma' \vdash e' : \sigma}{\Gamma \vdash \mathbf{let}\, \mathit{bind}\, \mathbf{in}\, e' : \sigma}\ \textsf{Let}$$

$$\frac{\Gamma, x : \tau \vdash e : \sigma}{\Gamma \vdash \lambda x{:}\tau.e : \tau \to \sigma}\ {\to}\textsf{I} \qquad \frac{\Gamma \vdash e : \tau \to \sigma \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e\, e' : \sigma}\ {\to}\textsf{E} \qquad \frac{\Gamma, a : \kappa \vdash e : \tau}{\Gamma \vdash \Lambda a{:}\kappa.e : \forall a{:}\kappa.\tau}\ \forall\textsf{I} \qquad \frac{\Gamma \vdash e : \forall a{:}\kappa.\tau \quad \Gamma \vdash \sigma : \kappa}{\Gamma \vdash e\, \sigma : \tau\{\sigma/a\}}\ \forall\textsf{E}$$

$$\frac{K : \forall \overrightarrow{a{:}\kappa}.\forall \overrightarrow{b{:}\kappa'}.\overrightarrow{\tau'} \to T\,\vec{a} \quad \overrightarrow{\Gamma \vdash \sigma : \kappa'} \quad \overrightarrow{\Gamma \vdash e : \tau'\overrightarrow{\{\tau/a\}}\overrightarrow{\{\sigma/b\}}}}{\Gamma \vdash K\ \vec{\sigma}\ \vec{e}\ : T\,\vec{\tau}}\ \textsf{TKI} \qquad \frac{\Gamma \vdash e : \tau \quad \overrightarrow{\Gamma \mid \tau \vdash \mathit{alt} : \sigma}}{\Gamma \vdash \mathbf{case}\, e\, \mathbf{of}\, \overrightarrow{\mathit{alt}} : \sigma}\ \textsf{Case}$$

Alternative typing: $\boxed{\Gamma \mid \tau \vdash e : \sigma}$

$$\frac{\Gamma, x : \tau \vdash e : \sigma}{\Gamma \mid \tau \vdash x{:}\tau \to e : \sigma}\ \textsf{Deflt} \qquad \frac{K : \forall \overrightarrow{a{:}\kappa'}.\forall \overrightarrow{b{:}\kappa}.\vec{\sigma} \to T\,\vec{a} \quad \Gamma, \overrightarrow{b : \kappa}, \overrightarrow{x : \sigma\overrightarrow{\{\tau/a\}}} \vdash e : \sigma'}{\Gamma \mid T\,\vec{\tau} \vdash K\ \overrightarrow{b{:}\kappa}\ \overrightarrow{x{:}\sigma} \to e : \sigma'}\ \textsf{TKE}$$

Binding typing: $\boxed{\Gamma \vdash \mathit{bind} : \{\Gamma'\}}$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash x{:}\tau = e : \{x : \tau\}}\ \textsf{Name} \qquad \frac{\overrightarrow{\Gamma, \overrightarrow{x : \tau} \vdash e : \tau}}{\Gamma \vdash \mathbf{rec}\, \{\overrightarrow{x{:}\tau = e}\} : \{\Gamma'\}}\ \textsf{Rec}$$

Figure 13. Type system for Core

By the induction hypothesis, we then have:

$$\begin{array}{c} \mathcal{D}' \\ \vdots \\ \Gamma \vdash S\,[\![e]\!] : \sigma \to \tau \end{array} \qquad \begin{array}{c} \mathcal{E}' \\ \vdots \\ \Gamma \vdash S\,[\![e']\!] : \sigma \end{array}$$

Now, noting that

$$S\,[\![e\, e']\!] = \mu\mathsf{ret}.\left\langle S\,[\![e]\!] \,\middle\|\, S\,[\![e']\!] \cdot \mathsf{ret}\right\rangle,$$

we have:

$$\dfrac{\dfrac{\begin{array}{c} \mathcal{D}' \\ \vdots \\ \Gamma \vdash S\,[\![e]\!] : \sigma \to \tau \end{array} \qquad \dfrac{\begin{array}{c} \mathcal{E}' \\ \vdots \\ \Gamma \vdash S\,[\![e']\!] : \sigma \end{array} \quad \dfrac{}{\Gamma \mid \mathsf{ret} : \tau \vdash \mathsf{ret} : \tau}\ \textsf{Ret}}{\Gamma \mid S\,[\![e']\!] \cdot \mathsf{ret} : \sigma \to \tau \vdash \mathsf{ret} : \tau}\ {\to}\textsf{L}}{\left\langle S\,[\![e]\!] \,\middle\|\, S\,[\![e']\!] \cdot \mathsf{ret}\right\rangle : (\Gamma \vdash \mathsf{ret} : \tau)}\ \textsf{Cut}}{\Gamma \vdash \mu\mathsf{ret}.\left\langle S\,[\![e]\!] \,\middle\|\, S\,[\![e']\!] \cdot \mathsf{ret}\right\rangle : \tau}\ \textsf{Act}$$

□

Proving type safety of D hits a snag: while D does not change the type of a term, it *does* change the type of a join point. Namely, if a join point has type $\exists\vec{a}.(\vec{\sigma})$ and its context gives ret the type τ, it will become a function of type $\forall\vec{a}.\vec{\sigma} \to \tau$. Thus we define D_τ on types, homomorphically except for

$$D_\tau\,[\![\exists\vec{a}.(\vec{\sigma})]\!] \equiv \forall\vec{a}.\vec{\sigma} \to \tau.$$

Then, we have D operate on continuation contexts:

$$D\left[\!\!\left[\overrightarrow{j : \sigma}, \mathsf{ret} : \tau\right]\!\!\right] \equiv D_\tau\left[\!\!\left[\overrightarrow{j : \sigma}\right]\!\!\right] \equiv \overrightarrow{j : D_\tau\,[\![\sigma]\!]}.$$

Now we can state and prove the general form of type safety for D:

Lemma 34 (Type safety of D). *1. If $\Gamma \vdash v : \tau$, then $\Gamma \vdash D\,[\![v]\!] : \tau$.*

2. If $\Gamma \mid k : \sigma \vdash \Delta, \mathsf{ret} : \tau$ and $\Gamma \vdash e : \sigma$, then $\Gamma, D_\tau\,[\![\Delta]\!] \vdash D\,[\![k]\!]\,[e] : \tau$.

3. If $c : (\Gamma \vdash \Delta, \mathsf{ret} : \tau)$, then $\Gamma, D_\tau\,[\![\Delta]\!] \vdash D\,[\![c]\!] : \tau$.

4. If $\mathit{bind} : (\Gamma \mid \Delta' \vdash \Gamma' \mid \Delta, \mathsf{ret} : \tau)$, then $\Gamma, D_\tau\,[\![\Delta]\!] \vdash D\,[\![\mathit{bind}]\!] : \Gamma', D_\tau\,[\![\Delta']\!]$.

Proof. By mutual induction on the typing derivations. We show a few cases:

- In 2, suppose we have

$$\dfrac{\begin{array}{c} \mathcal{D} \\ \vdots \\ \Gamma \vdash v : \sigma \end{array} \quad \begin{array}{c} \mathcal{E} \\ \vdots \\ \Gamma \mid k : \sigma' \vdash \Delta, \mathsf{ret} : \tau \end{array}}{\Gamma \mid v \cdot k : \sigma \to \sigma' \vdash \Delta, \mathsf{ret} : \tau}\ {\to}\textsf{L}$$

  and also:

$$\begin{array}{c} \mathcal{F} \\ \vdots \\ \Gamma \vdash e : \sigma \to \sigma' \end{array}$$

  By the induction hypothesis, we then have:

$$\begin{array}{c} \mathcal{D}' \\ \vdots \\ \Gamma \vdash D\,[\![v]\!] : \sigma \end{array} \qquad \begin{array}{c} \Gamma \vdash e\, D\,[\![v]\!] : \sigma' \\ \vdots \\ \mathcal{E}' \\ \vdots \\ \Gamma, D_\tau\,[\![\Delta]\!] \vdash D\,[\![k]\!]\,[e\, D\,[\![v]\!]] : \tau \end{array}$$

  Noting that

$$D\,[\![v \cdot k]\!]\,[e] \equiv D\,[\![k]\!]\,[e\, D\,[\![v]\!]],$$

$$
\dfrac{\dfrac{}{\Gamma' \vdash j : \forall \overrightarrow{a:\kappa}.\overrightarrow{\sigma} \to \tau}\;\mathsf{Var} \qquad \overrightarrow{\dfrac{\overset{\overrightarrow{\mathcal{D}}}{\vdots}}{\Gamma \vdash \sigma' : \kappa}}}{\Gamma' \vdash j\,\overrightarrow{\sigma'} : \sigma\{\overrightarrow{\sigma'/a}\} \to \tau}\;\forall\mathsf{E} \qquad \overrightarrow{\dfrac{\overset{\overrightarrow{\mathcal{E}'}}{\vdots}}{\Gamma \vdash D\llbracket v \rrbracket : \sigma\{\overrightarrow{\sigma'/a}\}}}
$$
$$
\overline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}\;\overrightarrow{\to\mathsf{E}}
$$
$$
\Gamma' \vdash j\,\overrightarrow{\sigma'}\,\overrightarrow{D\llbracket v \rrbracket} : \tau
$$

Figure 14. Proof of Lemma 34, jump case

we see that $\mathcal{E}'$ is has the conclusion we require, so long as we can prove its premise. Thus we write:

$$
\begin{array}{c}
\dfrac{\overset{\mathcal{F}}{\vdots}\;\;\Gamma \vdash e : \sigma \to \sigma' \qquad \overset{\mathcal{D}'}{\vdots}\;\;\Gamma \vdash D\llbracket v \rrbracket : \sigma}{\Gamma \vdash e\; D\llbracket v \rrbracket : \sigma'}\;\to\mathsf{E} \\
\vdots \\
\mathcal{E}' \\
\vdots \\
\Gamma, D_\tau\llbracket \Delta \rrbracket \vdash D\llbracket k \rrbracket\,[e\; D\llbracket v \rrbracket] : \tau
\end{array}
$$

- In 3, suppose we have:

$$
\dfrac{\overrightarrow{\dfrac{\overset{\overrightarrow{\mathcal{D}}}{\vdots}}{\Gamma \vdash \sigma' : \kappa}} \qquad \overrightarrow{\dfrac{\overset{\overrightarrow{\mathcal{E}}}{\vdots}}{\Gamma \vdash v : \sigma\{\overrightarrow{\sigma'/a}\}}}}{\mathsf{jump}\; j\,\overrightarrow{\sigma'}\;\overrightarrow{v} : (\Gamma \vdash j : \exists\overrightarrow{a{:}\kappa}.(\overrightarrow{\sigma}), \Delta, \mathsf{ret} : \tau)}\;\mathsf{Jump}
$$

By the induction hypothesis, we have:

$$
\overrightarrow{\dfrac{\overset{\overrightarrow{\mathcal{E}'}}{\vdots}}{\Gamma \vdash D\llbracket v \rrbracket : \sigma\{\overrightarrow{\sigma'/a}\}}}
$$

Noting that

$$
D\left\llbracket \mathsf{jump}\; j\,\overrightarrow{\sigma'}\;\overrightarrow{v} \right\rrbracket \equiv j\,\overrightarrow{\sigma'}\;\overrightarrow{D\llbracket v \rrbracket}
$$

and

$$
D_\tau\llbracket j : \exists\overrightarrow{a:\kappa}.(\overrightarrow{\sigma}) \rrbracket \equiv j : \forall\overrightarrow{a:\kappa}.\overrightarrow{\sigma} \to \tau,
$$

letting

$$
\Gamma' \equiv \Gamma, j : \forall\overrightarrow{a:\kappa}.\overrightarrow{\sigma} \to \tau, D_\tau\llbracket \Delta \rrbracket\,,
$$

we then have the derivation in Figure 14[13].

- In 4, suppose we have:

$$
\dfrac{\overset{\mathcal{D}}{\vdots}\;\; c : (\Gamma, \overrightarrow{a:\kappa}, \overrightarrow{x:\sigma} \vdash \Delta, \mathsf{ret} : \tau)}{(j = \tilde{\mu}[\overrightarrow{a{:}\kappa}, \overrightarrow{x{:}\sigma}].c) : (\Gamma \mid j : \exists\overrightarrow{a{:}\kappa}.(\overrightarrow{\sigma}) \vdash \varepsilon \mid \Delta, \mathsf{ret} : \tau)}\;\mathsf{Label}
$$

By the induction hypothesis, we have:

$$
\begin{array}{c}
\mathcal{D}' \\
\vdots \\
\Gamma, \overrightarrow{a:\kappa}, \overrightarrow{x:\sigma}, D_\tau\llbracket \Delta \rrbracket \vdash D\llbracket c \rrbracket : \tau
\end{array}
$$

Noting that

$$
D\llbracket j = \tilde{\mu}[\overrightarrow{a{:}\kappa}, \overrightarrow{x{:}\sigma}].c \rrbracket \equiv \big(j = \Lambda\overrightarrow{a{:}\kappa}.\lambda\overrightarrow{x{:}\sigma}.D\llbracket c \rrbracket\big)
$$

and

$$
D_\tau\llbracket j : \exists\overrightarrow{a:\kappa}.(\overrightarrow{\sigma}) \rrbracket \equiv j : \forall\overrightarrow{a:\kappa}.\overrightarrow{\sigma} \to \tau,
$$

we then have:

$$
\dfrac{\dfrac{\dfrac{\overset{\mathcal{D}'}{\vdots}\;\;\Gamma, \overrightarrow{a{:}\kappa}, \overrightarrow{x{:}\sigma}, D_\tau\llbracket \Delta \rrbracket \vdash D\llbracket c \rrbracket : \tau}{\Gamma, \overrightarrow{a{:}\kappa}, D_\tau\llbracket \Delta \rrbracket \vdash \lambda\overrightarrow{x{:}\sigma}.D\llbracket c \rrbracket : \overrightarrow{\sigma} \to \tau}}{\Gamma, D_\tau\llbracket \Delta \rrbracket \vdash \Lambda\overrightarrow{a{:}\kappa}.\lambda\overrightarrow{x{:}\sigma}.D\llbracket c \rrbracket : \forall\overrightarrow{a{:}\kappa}.\overrightarrow{\sigma} \to \tau}\;\to\mathsf{I}}{\Gamma, D_\tau\llbracket \Delta \rrbracket \vdash (j = \Lambda\overrightarrow{a{:}\kappa}.\lambda\overrightarrow{x{:}\sigma}.D\llbracket c \rrbracket) : (j : \forall\overrightarrow{a{:}\kappa}.\overrightarrow{\sigma} \to \tau)}\;\forall\mathsf{I}
$$

□

Proof of Proposition 4. Immediate from Lemmas 33 and 34. □

[13] The reader may notice we make implicit use of weakening in this derivation.