In submission. This is the extended version with appendices.

Sequent calculus as a compiler intermediate language

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

Simon Peyton Jones
Microsoft Research Cambridge
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 be a good intermediate language, too? To explore this question we designed Sequent Core, a practically-oriented variant of 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 [37, 38] 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 [36], its successors (e.g. Orbit [18]), 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* [10], using the *Curry-Howard isomorphism* [41]: 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 [10], called the *sequent calculus*. Thanks to the Curry-Howard isomorphism, the sequent calculus can also be seen as a programming language [5, 11, 40] with an emphasis on *control flow*.

This raises an obvious question: *does 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 language of the sequent calculus called Sequent Core with the same expressiveness as System Fω, together with **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).
- The proof of the pudding is in the eating. We have implemented our intermediate language as a plugin 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 to instead use Sequent Core, notably the Simplifier.
- 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 like 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* [17] (Section 5).

So what kind of intermediate language do we get from the sequent calculus? Unsurprisingly, it bears a resemblance to *continuation-passing style*, a common technique in the λ-calculus for representing control flow inside a program. 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 dictates [41]: 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

Shared syntax of kinds and types

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

Syntax of Sequent Core

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

Syntax of Core

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

Figure 1. Syntax

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 "for free" is more expressive [7] 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 [34]. 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
$\quad \mathit{plusOne}{:}\mathit{Int}{\to}\mathit{Int}\ = \lambda x{:}\mathit{Int}.\ \ (+)\ \ x\ \ 1$

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

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* $j{:}\tau = \tilde{\mu}[\overrightarrow{a{:}\kappa}, \overrightarrow{x{:}\tau}].c$.[1] 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* c, which are a collection of local bindings wrapping either a *cut pair* $\langle v \parallel k \rangle$ or a *jump*. We defer the discussion of jumps to 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 $\mathbf{case}$ analysis or a return to ret.

In *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 state of many abstract machines (e.g. the CEK machine [8]) with a term v in the focus, and a continuation or call stack k that describes how that term is consumed.

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

Core

$$\begin{array}{l}
\quad \mathit{init}\ {:}\forall a.[a]{\to}[a] \\
\qquad = \Lambda a.\lambda \mathit{xs}{:}[a].\,\mathbf{case}\ \ \mathit{reverse}\ \ a\ \ \mathit{xs}\ \ \mathbf{of} \\
\qquad\qquad\qquad\qquad\qquad \begin{array}{lcl} [] & \to & [] \\ (y{:}\mathit{ys}) & \to & \mathit{reverse}\ \ a\ \ \mathit{ys} \end{array}
\end{array}$$

Sequent Core

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

As before, the outer structure is the same, but the $\mathbf{case}$ 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. 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.

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

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

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 in Figure 1 contain 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 immediately produce themselves as their result.

On the other hand, Core also includes expressions which *do* require evaluation: function application $e\ e'$, polymorphic instantiation $e\ \tau$, 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:

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

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 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, $\mu\mathsf{ret}.\ \langle g \,\|\, x \cdot \mathsf{ret} \rangle$ 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 $\langle f \,\|\, (\mu\mathsf{ret}.\ \langle g \,\|\, x \cdot \mathsf{ret} \rangle) \cdot \mathsf{ret} \rangle$.

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 $\langle v \,\|\, k \rangle$ 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 $\langle \mu\mathsf{ret}.c \,\|\, k \rangle$. But something is missing. How can we give a shared label to a command (i.e. 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

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

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 $\overline{\lambda}\mu\tilde{\mu}$ [5], 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 $\tilde{\mu}[a_1, \dots, a_n, x_1, \dots, x_m].c$ in the style of $\overline{\lambda}\mu\tilde{\mu}$. These continuations accept several inputs (named $x_1 \dots x_m$), whose types are polymorphic over the choice of types for $a_1 \dots a_n$, in order to run a command c. Intuitively, we may also think of these multiple-input continuations as a sequence of lambdas $\Lambda a_1 \dots a_n.\lambda x_1 \dots x_m.c$, 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 $j = \tilde{\mu}[x, y].\ \langle (+) \,\|\, x \cdot y \cdot \mathsf{ret} \rangle$. 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

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

will jump to the label j with the inputs Bool, True and not which results in $\langle \mathit{not} \,\|\, \mathit{True} \cdot \mathsf{ret} \rangle$. So when viewing Sequent Core from the perspective of an abstract machine, we could say that its language of commands 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 labeled continuations do not introduce a μ-binder. As a consequence, the ret found in $j = \tilde{\mu}[x, y].\ \langle (+) \,\|\, x \cdot y \cdot \mathsf{ret} \rangle$ refers to the nearest surrounding μ command, unlike the ret found in $f = \lambda x.\lambda y.\mu\mathsf{ret}.\ \langle (+) \,\|\, x \cdot y \cdot \mathsf{ret} \rangle$. By viewing ret as a statically bound variable it 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 **case** alternatives for free.

2.2.3 The scope of labels

There is one major restriction that we enforce to ensure that terms must always have a *unique* exit point by which they return their result, and so evaluating a term cannot cause a noticeable 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 restrictions of CPS [17], 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 **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:

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

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 both f and j, we can break the recursive bindings into two separate blocks with the correct scope, and place the binding for f first:

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

While we do not syntactically enforce this restriction, it 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 gives 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 **case** are disambiguated by selecting the first match, so the order of alternatives matters. For a non-recursive let we simply substitute, thus implementing call-by-name; implementing recursive let is only slightly harder, but we omit it here for simplicity. Note that the rule for continuation **let**s uses *structural substitution* [25], 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

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

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

Figure 2. Call-by-name operational semantics

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 with a reduction or equational theory is the difference between reducing anywhere or 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 [19] 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 **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 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**-

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

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

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

$$\begin{array}{rrl}
(\beta^{\rightarrow}) & \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{\mathit{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{\mathit{alt}} \\
(\mathbf{case}_{def}) & \langle \mathcal{H};\ \mathcal{J}, \mathcal{R};\ \left\langle W \,\middle\|\, \mathbf{case\,of}\ \overrightarrow{\mathit{alt}}\right\rangle\rangle \rightsquigarrow \langle \mathcal{H}, x = W;\ \mathcal{J}, \mathcal{R};\ c\rangle & x \to c \in \overrightarrow{\mathit{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};\ \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

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 [21, 29].

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 [2].

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. In particular, it is an extension (as well as a restriction) of the type system for the $\bar{\lambda}\mu\tilde{\mu}$-calculus [5, 12], and likewise we share the same unconventional notation for typing judgements to maintain a visible connection with the sequent calculus. In particular, the typing judgements for the different syntactic categories of Sequent Core are written with the following sequents:

- The type of commands 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 terms 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 continuations 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 scope 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, every typing rule becomes a rule of the sequent calculus. For example, $\to$L becomes the implication left rule

$$\frac{\Gamma \vdash \sigma \quad \Gamma, \tau \vdash \Delta}{\Gamma, \sigma \to \tau \vdash \Delta} \to\mathsf{L}$$

The only difference in the classic rule is that Δ appears in both premises, which comes from the fact that the sequent calculus was originally developed for *classical logic*, where as the pure functional programs we model correspond to *intuitionistic logic*. The same story holds for all the other left (L) and right (R) rules of Figure 4. Var and Ret are axioms for the initial sequents. Cut is an intuitionistic cut rule, and **let** gives a *multi-cut* rule.

Perhaps the most complex rules in Figure 4 are the ones for multiple-input continuations. It may seem a bit bizarre that the

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 4. Type System

polymorphism is pronounced "exists" instead of "forall," but luckily the above Curry-Howard reading helps explain what's going on. There are two instances of the Jump and Label rules that are helpful to consider. First is when we have exactly two monomorphic inputs:

$$\frac{\Gamma \vdash \tau_1 \quad \Gamma \vdash \tau_2}{\Gamma \vdash (\tau_1, \tau_2)}\;\mathsf{Jump} \qquad \frac{\Gamma, \tau_1, \tau_2 \vdash \Delta}{\Gamma, (\tau_1, \tau_2) \vdash \Delta}\;\mathsf{Label}$$

These are exactly the right and left rules for the (tensor) product type in the sequent calculus, illustrating the (tuple) product nature of multiple-input continuations and jumps. Second is when we have one polymorphic type quantified over one input:

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

These are exactly the right and left rules for existential types in the sequent calculus, which justifies the use of $\exists$ for the types of labels and jumps. Notice that the static scope of type variables uniformly gives the correct logical provisos for the universal quantifier in the $\forall$R rule and the existential quantifiers in the Label and TL rules. For example, the following command from Section 2.2.2

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

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

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

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 analogous restrictions captured here.

Also note 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 *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) lemmas.

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* $\left\langle K(\overrightarrow{\sigma}, \overrightarrow{v}) \,\middle\|\, \mathbf{case\,of}\;\overrightarrow{\mathit{alt}}\right\rangle$ *where neither* $K(\overrightarrow{b{:}\kappa}, \overrightarrow{x{:}\sigma}) \to c$ *nor* $x : \tau \to c$ *are in* $\overrightarrow{\mathit{alt}}$.
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 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.

$$S[\![\lambda x{:}\tau.e]\!] = \lambda x{:}\tau.S[\![e]\!] \qquad S[\![x]\!] = x$$
$$S[\![\Lambda a{:}\kappa.e]\!] = \Lambda a{:}\kappa.S[\![e]\!] \qquad S[\![K\ \vec{\sigma}\ \vec{e}]\!] = K(\vec{\sigma}, \overrightarrow{S[\![e]\!]})$$
$$S[\![\mathbf{let}\ bind\ \mathbf{in}\ e]\!] = \mu\mathsf{ret}.\ \mathbf{let}\ S[\![bind]\!]\ \mathbf{in}\ \langle S[\![e]\!] \parallel \mathsf{ret}\rangle$$
$$S[\![e\ e']\!] = \mu\mathsf{ret}.\ \langle S[\![e]\!] \parallel S[\![e']\!] \cdot \mathsf{ret}\rangle$$
$$S[\![e\ \tau]\!] = \mu\mathsf{ret}.\ \langle S[\![e]\!] \parallel \tau \cdot \mathsf{ret}\rangle$$
$$S\left[\!\!\left[\mathbf{case}\ e\ \mathbf{of}\ \overrightarrow{alt}\right]\!\!\right] = \mu\mathsf{ret}.\ \left\langle S[\![e]\!] \,\middle\|\, \mathbf{case\ of}\ \overrightarrow{S[\![alt]\!]}\right\rangle$$
$$S[\![x{:}\tau = e]\!] = x{:}\tau = S[\![e]\!]$$
$$S[\![\mathbf{rec}\ \{\overrightarrow{x{:}\tau = e}\}]\!] = \mathbf{rec}\left\{\overrightarrow{x{:}\tau = S[\![e]\!]}\right\}$$
$$S[\![x{:}\tau \to e]\!] = x{:}\tau \to \langle S[\![e]\!] \parallel \mathsf{ret}\rangle$$
$$S[\![K\ \overrightarrow{a{:}\kappa}\ \overrightarrow{x{:}\tau} \to e]\!] = K(\overrightarrow{a{:}\kappa}, \overrightarrow{x{:}\tau}) \to \langle S[\![e]\!] \parallel \mathsf{ret}\rangle$$

Figure 5. Definitional translation from Core to Sequent Core

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.

3.1 A definitional translation

Luckily, we can leverage the relationship between natural deduction and sequent calculus [5] to come up with a definitional translation from Sequent Core to Core, as shown in Figure 5. 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

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

instead of the direct $\mu\mathsf{ret}.\ \langle f \parallel 1\cdot 2\cdot 3\cdot\mathsf{ret}\rangle$. 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 [28] for the sequent calculus, S_a, which we achieve by aggressively performing μ reductions during translation as shown in Figure 6 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:

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

Core to Sequent Core

$$S_a[\![\lambda x{:}\tau.e]\!] = \lambda x{:}\tau.S_a[\![e]\!] \qquad S_a[\![x]\!] = x$$
$$S_a[\![\Lambda a{:}\kappa.e]\!] = \Lambda a{:}\kappa.S_a[\![e]\!] \qquad S_a[\![K\ \vec{\sigma}\ \vec{e}]\!] = K(\vec{\sigma}, \overrightarrow{S_a[\![e]\!]})$$
$$S_a[\![e]\!] = \mu\mathsf{ret}.S_a[\![e]\!]\ \mathsf{ret} \qquad \text{where } e \text{ is a computation}$$
$$S_a[\![\mathbf{let}\ bind\ \mathbf{in}\ e]\!]\ k = \mathbf{let}\ S_a[\![bind]\!]\ \mathbf{in}\ S_a[\![e]\!]\ k$$
$$S_a[\![e\ e']\!]\ k = S_a[\![e]\!]\ (S_a[\![e']\!] \cdot k)$$
$$S_a[\![e\ \tau]\!]\ k = S_a[\![e]\!]\ (\tau \cdot k)$$
$$S_a\left[\!\!\left[\mathbf{case}\ e\ \mathbf{of}\ \overrightarrow{alt}\right]\!\!\right]\ k = \mathbf{let}\ \overrightarrow{bind}\ \mathbf{in}\ S_a[\![e]\!]\ (\mathbf{case\ of}\ \overrightarrow{S_a[\![alt]\!]\ k'})$$
$$\text{where } (\overrightarrow{bind}, k') = \mathsf{shrink}(k)$$
$$S_a[\![e]\!]\ k = \langle S_a[\![e]\!] \parallel k\rangle \qquad \text{where } e \text{ is a value}$$
$$S_a[\![x{:}\tau = e]\!] = x{:}\tau = S_a[\![e]\!]$$
$$S_a[\![\mathbf{rec}\ \{\overrightarrow{x{:}\tau = e}\}]\!] = \mathbf{rec}\left\{\overrightarrow{x{:}\tau = S_a[\![e]\!]}\right\}$$
$$S_a[\![x{:}\tau \to e]\!]\ k = x{:}\tau \to S_a[\![e]\!]\ k$$
$$S_a[\![K\ \overrightarrow{a{:}\kappa}\ \overrightarrow{x{:}\tau} \to e]\!]\ k = K(\overrightarrow{a{:}\kappa}, \overrightarrow{x{:}\tau}) \to S_a[\![e]\!]\ k$$

Sequent Core to Core

$$D[\![\lambda x{:}\tau.v]\!] = \lambda x{:}\tau.D[\![v]\!] \qquad D[\![x]\!] = x$$
$$D[\![\Lambda a{:}\kappa.v]\!] = \Lambda a{:}\kappa.D[\![v]\!] \qquad D[\![K(\vec{\sigma}, \vec{v})]\!] = K\ \vec{\sigma}\ \overrightarrow{D[\![v]\!]}$$
$$D[\![\mu\mathsf{ret}.c]\!] = D[\![c]\!]$$
$$D[\![\mathbf{let}\ bind\ \mathbf{in}\ c]\!] = \mathbf{let}\ D[\![bind]\!]\ \mathbf{in}\ D[\![c]\!]$$
$$D[\![\langle v \parallel k\rangle]\!] = D[\![k]\!]\ [D[\![v]\!]]$$
$$D[\![\mathsf{jump}\ j\ \vec{\sigma}\ \vec{v}]\!] = j\ \vec{\sigma}\ \overrightarrow{D[\![v]\!]}$$
$$D[\![v \cdot k]\!] = D[\![k]\!]\ [\Box\ D[\![v]\!]] \qquad D[\![\mathsf{ret}]\!] = \Box$$
$$D[\![\sigma \cdot k]\!] = D[\![k]\!]\ [\Box\ \sigma] \qquad D\left[\!\!\left[\mathbf{case\ of}\ \overrightarrow{alt}\right]\!\!\right] = \mathbf{case}\ \Box\ \mathbf{of}\ \overrightarrow{D[\![alt]\!]}$$
$$D[\![x{:}\tau = v]\!] = (x{:}\tau = D[\![v]\!])$$
$$D[\![j{:}\exists\overrightarrow{a{:}\kappa}.(\vec{\tau}) = \tilde{\mu}[\overrightarrow{a{:}\kappa}, \overrightarrow{x{:}\tau}].c]\!] = (j{:}\forall\overrightarrow{a{:}\kappa}.\vec{\tau}\to\sigma = \Lambda\overrightarrow{a{:}\kappa}.\lambda\overrightarrow{x{:}\sigma}.D[\![c]\!])$$
$$\text{where } \mathsf{ret} : \sigma$$
$$D\left[\!\!\left[\mathbf{rec}\left\{\overrightarrow{bp}\right\}\right]\!\!\right] = \mathbf{rec}\left\{\overrightarrow{D[\![bp]\!]}\right\}$$
$$D[\![x{:}\sigma \to c]\!] = x{:}\sigma \to D[\![c]\!]$$
$$D[\![K(\overrightarrow{a{:}\kappa}, \overrightarrow{x{:}\sigma}) \to c]\!] = K\ \overrightarrow{a{:}\kappa}\ \overrightarrow{x{:}\sigma} \to D[\![c]\!]$$

Figure 6. Round-trip translations from Core to Sequent Core (S_a) and back to Core (D)

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 6.

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)**.** *$D\llbracket S_a\llbracket e\rrbracket\rrbracket \cong e$ and $S_a\llbracket D\llbracket v\rrbracket\rrbracket \cong v$.*

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 $\Gamma \vdash S_a\llbracket e\rrbracket : \tau$ in Sequent Core. If $\Gamma \vdash v : \tau$ in Sequent Core then $\Gamma \vdash D\llbracket v\rrbracket : \tau$ in Core.*

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 plug-in for the Glasgow Haskell Compiler (GHC), a mature, production-quality compiler for Haskell. In this section we reflect what we learned from this experience. The plug-in is available on GitHub.[2]

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-reduction and -expansion, 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 doing the following: 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 *SimplCont*) in the style of a zipper [14], 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 [35, 36]. 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* [26, 34]. Consider:

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

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

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

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

Notice how the outer boolean **case** will receive a *True* or *False* value, depending on the result of $(x\ \text{‘}mod\text{‘}\ 2)$. 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**:

$$
\begin{array}{lllll}
half = \lambda x.\ \textbf{case}\ x\ \text{‘}mod\text{‘}\ 2\ \textbf{of} \\
\quad 0 \to \textbf{case}\ True & \textbf{of} & True & \to & Just\,(x\ \text{‘}div\text{‘}\ 2) \\
 & & False & \to & Nothing \\
\quad \_ \to \textbf{case}\ False & \textbf{of} & True & \to & Just\,(x\ \text{‘}div\text{‘}\ 2) \\
 & & False & \to & Nothing
\end{array}
$$

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

$$
\begin{array}{lll}
half = \lambda x.\ \textbf{case}\ x\ \text{‘}mod\text{‘}\ 2\ \textbf{of} & 0 \to & Just\,(x\ \text{‘}div\text{‘}\ 2) \\
 & \_ \to & Nothing
\end{array}
$$

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.[3]

Like many compilers, including the original ANF implementation [9], GHC avoids excessive code duplication by abstracting large **case** alternatives into named functions that serve as join points. A typical result looks like this:

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

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: **case** $f\ x$ **of** $0 \to False\,;\ \_ \to True$. If *f* is inlined at this call site, another case-of-case transformation will occur, and after some further simplifications we get this:

$$
\begin{array}{llllll}
\textbf{let}\ j : Maybe\ Int \to Int = \lambda w.\ \ldots \\
\textbf{in case}\ g\ x\ \textbf{of} & Left\ y & \to \textbf{case}\ j\ (Just\ y) & \textbf{of} & 0 \to & False \\
 & & & & \_ \to & True \\
 & Right\ \_ & \to \textbf{case}\ j\ Nothing & \textbf{of} & 0 \to & False \\
 & & & & \_ \to & True
\end{array}
$$

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:

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

This time, j is represented by a labeled continuation accepting a *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 $\langle\ f \parallel x \cdot \textbf{case of}\ 0 \to False\,;\ \_ \to True\ \rangle$ followed by routine Sequent Core simplification instead gives us:

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

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

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

where the **case** e **of** $0 \to False\,;\ \_ \to True$ means to wrap every expression e that returns from j with the **case** analysis. This effect is extremely hard to achieve in Core as it stands, because join points are not distinguished and so the above substitution is not obvious, but 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* [17] 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, 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:

$$
\begin{array}{llll}
\textbf{let}\ f = \lambda y.\ \mu\mathsf{ret}.c \\
\textbf{in}\ \langle\ g \parallel x \cdot \textbf{case of} & A\ z & \to & \langle\ f \parallel z \cdot \mathsf{ret}\ \rangle \\
 & B\ \_ & \to & \langle\ True \parallel \mathsf{ret}\ \rangle \\
 & C & \to & \langle\ f \parallel True \cdot \mathsf{ret}\ \rangle\ \rangle
\end{array}
$$

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:

$$
\begin{array}{llll}
\textbf{let}\ j = \tilde{\mu}[y].\,c \\
\textbf{in}\ \langle\ g \parallel x \cdot \textbf{case of} & A\ z & \to & \mathsf{jump}\, j\ z \\
 & B\ \_ & \to & \langle\ True \parallel \mathsf{ret}\ \rangle \\
 & C & \to & \mathsf{jump}\, j\ True\ \rangle
\end{array}
$$

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

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.

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 **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 **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 **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 **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 [17], except that we only contify functions called with the ret continuation rather than any single continuation. So 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 *Nothing* if none satisfy it:

```
find :: (a→Bool) → [a] → Maybe a
find = λp.λxs. let go = λys.case ys of
                              []       → Nothing
                              (y:ys')  → case p y of
                                           True  → Just y
                                           False → go ys'
               in go xs
```

Here *go* is a join point, and our contification analysis can discover that fact.[4] Now suppose that *find* is called as:

```
f = λxs. case find even xs of Nothing → 0
                              Just x  → x + 1
```

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

```
f = λxs. let go = λys.case ys of
                        []      → Nothing
                        (y:ys') → case y `mod` 2 of
                                    0 → Just y
                                    _ → go ys'
         in  case go xs of Nothing → 0
                           Just x  → x + 1
```

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:

```
f = λxs. let go = λys.case ys of
                        []      → 0
                        (y:ys') → case y `mod` 2 of
                                    0 → y + 1
                                    _ → go ys'
         in  go xs
```

This code is much better: the allocation of the function closure for *go* and the intermediate value (*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* — it must provide the same number of arguments as the 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* [3]. However, it is unclear that this is useful often enough to warrant

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

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:

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

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.[5] 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 [5] or System L [23] for example. A primary difference is that in lieu of **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 **let**. Then, using the call-by-name semantics for Sequent Core, all $\tilde{\mu}$-abstractions can be *lifted*[6] to the top of a command, so every $\tilde{\mu}$-abstraction can be written as a **let**. Under a call-by-value semantics, $\tilde{\mu}$-abstractions play a more crucial role as noted by Curien and Herbelin [5], but in that case they are exactly the same as a default **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 **let** cannot be represented with $\tilde{\mu}$-abstractions alone gives **let** its primary role in Sequent Core.

The other difference 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 have) consider adding general existential types to the language. That way a multiple-input continuation $\tilde{\mu}[\vec{a}, \vec{x}].c$ might be interpreted as just a **case** continuation $\mathbf{case\,of}\;(\vec{a}, \vec{x}) \to c$, so that labels just refer to single-input continuations and jumps are just cut pairs $\langle(\vec{\sigma}, \vec{v}) \parallel j\rangle$. However, for this to represent the correct operational cost at run time, it is crucial that these existential tuples are *unboxed* [27], meaning that they are values of a truly positive type [23] 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 [42] interpretation of focalization.

6.2 CPS as an intermediate language

Though continuations had been actively researched for nearly a decade [30], the first use of CPS for compilation appeared in 1978, in the Rabbit compiler for Scheme [36]. Steele was interested in the connection between Scheme and the λ-calculus [39], 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. There have of course been CPS transforms given for call-by-name [13, 28] and call-by-need [24], but (to our knowledge) they have not been used in compilers before, so we would be 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 [26].

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 [15], which enable user optimizations like stream fusion [4]. 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" [32] CPS language. There is an essential trade-off in the expressive capabilities of rigid structure versus free abstraction [31], and the additional structure provided by call stacks enables more optimizations, like rewrite rules, by making continuations scrutable.

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

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

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

[6] This lifting can be done by the ς reductions of [40] and [22].

6.3 ANF as an intermediate language

In 1992, Sabry and Felleisen [33] 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 [28] with more reductions and hence further possible optimizations. Flanagan *et. al.* [9] 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 [17] 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 [23], 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) [6]. While it is known that SSA can be converted to CPS [16], 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 [20] 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:

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

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. In particular, the killer advantage of Sequent Core has turned out to be its treatment of join points, 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. 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. 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 [7], 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 cohesively 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. Besides 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.

References

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

[2] 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. . URL http://doi.acm.org/10.1145/199448.199507.

[3] 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. . URL http://dx.doi.org/10.1007/978-3-319-14675-1_3.

[4] 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. . URL http://doi.acm.org/10.1145/1291151.1291199.

[5] 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. . URL http://doi.acm.org/10.1145/351240.351262.

[6] 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. . URL http://doi.acm.org/10.1145/115372.115320.

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

[8] M. Felleisen and D. P. Friedman. *Control Operators, the SECD-Machine, and the λ-Calculus*. Indiana University, Computer Science Department, 1986.

[9] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, pages 237–247, 1993. . URL http://doi.acm.org/10.1145/155090.155113.

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

[11] 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. . URL http://dx.doi.org/10.1007/BFb0022247.

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

[13] 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. . URL http://dx.doi.org/10.1109/LICS.1997.614964.

[14] G. P. Huet. The zipper. *Journal of Functional Programming*, 7(5): 549–554, 1997.

[15] S. L. P. 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.

[16] R. Kelsey. A correspondence between continuation passing style and static single assignment form. In *Proceedings ACM SIGPLAN Workshop on Intermediate Representations (IR'95), San Francisco, CA, USA, January 22, 1995*, pages 13–23, 1995. . URL http://doi.acm.org/10.1145/202529.202532.

[17] A. Kennedy. Compiling with continuations, continued. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, pages 177–190, 2007. . URL http://doi.acm.org/10.1145/1291151.1291179.

[18] 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. . URL http://doi.acm.org/10.1145/12276.13333.

[19] J. Launchbury. A natural semantics for lazy evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*, pages 144–154, 1993. . URL http://doi.acm.org/10.1145/158511.158618.

[20] 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 2015, San Francisco, CA, USA, February 07 - 11, 2015*, pages 202–212, 2015. . URL http://dx.doi.org/10.1109/CGO.2015.7054200.

[21] 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.

[22] G. Munch-Maccagnoni. Focalisation and classical realisability. In *Computer Science Logic*, pages 409–423. Springer, 2009.

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

[24] C. Okasaki, P. Lee, and D. Tarditi. Call-by-need and continuation-passing style. *Lisp and Symbolic Computation*, 7(1):57–82, 1994.

[25] M. Parigot. Lambda-mu-calculus: An algorithmic interpretation of classical natural deduction. In *International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR92)*. Springer-Verlag, 1992.

[26] S. Peyton Jones and A. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1-3):3–47, Sept. 1998.

[27] 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.

[28] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975. .

[29] J.-C. Raoult and J. Vuillemin. Operational and Semantic Equivalence Between Recursive Programs. *Journal of the Association for Computing Machinery*, 27(4), October 1980.

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

[31] 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.

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

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

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

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

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

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

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

[39] 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.

[40] P. Wadler. Call-by-value is dual to call-by-name. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003*, pages 189–201, 2003. . URL `http://doi.acm.org/10.1145/944705.944723`.

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

[42] 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 7. 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.[7] 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.

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.

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

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

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

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}'') \\
&\quad \text{where} \quad (\mathcal{F}, \mathcal{C}) = A\,[\![v]\!] \\
&\quad\quad (\mathcal{F}', \mathcal{G}', \mathcal{C}') = (\mathcal{F}, \emptyset, \mathcal{C}) \oplus (\mathcal{F}_b, \mathcal{G}_b, \mathcal{C}_b) \\
&\quad\quad \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) \\
&\quad \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}''') \\
&\quad \text{where} \quad \overrightarrow{(\mathcal{F}, \mathcal{C})} = \overrightarrow{A\,[\![v]\!]} \\
&\quad\quad (\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) \\
&\quad\quad \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}') \\
&\quad \text{where} \quad \overrightarrow{(\mathcal{F}, \mathcal{G}, \mathcal{C})} = \overrightarrow{A\,[\![c]\!]} \\
&\quad\quad (\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 7. 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.