Compiling without Continuations

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

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

Abstract

Many fields of study in compilers give rise to the concept of a *join point*—a place where different execution paths come together. Join points are often treated as functions or continuations, but we believe it is time to study them in their own right. We show that adding join points to a direct-style functional intermediate language is a simple but powerful change that allows new optimizations to be performed, including a significant improvement to list fusion. Finally, we report on recent work on adding join points to the intermediate language of the Glasgow Haskell Compiler.

CCS Concepts • **Software and its engineering** → **Compilers**; *Software performance*; *Formal language definitions*; Functional languages; Control structures; • **Theory of computation** → Equational logic and rewriting

Keywords Intermediate languages; CPS; ANF; Stream fusion; Haskell; GHC

1. Introduction

Consider this code, in a functional language:

```
if (if e1 then e2 else e3) then e4 else e5
```

Many compilers will perform a *commuting conversion* [13], which naïvely would produce:

```
if e1 then (if e2 then e4 else e5)
      else (if e3 then e4 else e5)
```

Commuting conversions are tremendously important in practice (Sec. 2), but there is a problem: the conversion duplicates `e4` and `e5`. A natural countermeasure is to name the offending expressions and duplicate the names instead:

```
let { j4 () = e4; j5 () = e5 }
in if e1 then (if e2 then j4 () else j5 ())
         else (if e3 then j4 () else j5 ())
```

PLDI'17, June 18–23, 2017, Barcelona, Spain
ACM. 978-1-4503-4988-8/17/06...$15.00
http://dx.doi.org/10.1145/3062341.3062380

We describe `j4` and `j5` as *join points*, because they say where execution of the two branches of the outer `if` joins up again. The duplication is gone, but a new problem has surfaced: the compiler may allocate closures for locally-defined functions like `j4` and `j5`. That is bad because allocation is expensive. And it is tantalizing because all we are doing here is encoding control flow: it is plain as a pikestaff that the "call" to `j4` should be no more than a jump, with no allocation anywhere. That's what a C compiler would do! Some code generators can cleverly eliminate the closures, but perhaps not if further transformations intervene.

The reader of Appel's inspirational book [1] may be thinking *"Just use continuation-passing style (CPS)!"* When expressed over CPS terms, many classic optimizations boil down to β-reduction (*i.e.,* function application), or arithmetic reductions, or variants thereof. And indeed it turns out that commuting conversions fall out rather naturally as well. But using CPS comes at a fairly heavy price: the intermediate language becomes more complicated, some transformations are harder or out of reach, and (unlike direct style) CPS commits to a particular evaluation order (Sec. 8).

Inspired by Flanagan *et al.* [10], the reader may now be thinking *"OK, just use administrative normal form (ANF)!"* That paper shows that many transformations achievable in CPS are equally accessible in direct style. ANF allows an optimizer to *exploit* CPS technology without needing to *implement* it. The motto is: *Think in CPS; work in direct style.*

But alas, a subsequent paper by Kennedy shows that there remain transformations that are inaccessible in ANF but fall out naturally in CPS [16]. So the obvious question is this: could we extend ANF in some way, to get all the goodness of direct style *and* the benefits of CPS? In this paper we say "yes!", making the following contributions:

- We describe a modest extension to a direct-style λ-calculus intermediate language, namely adding join points (Sec. 3). We give the syntax, type system, and operational semantics, together with optimising transformations.
- We describe how to infer which ordinary bindings are in fact join points (Sec. 4). In a CPS setting this analysis is called *contification* [16], but it looks rather different here.

- We show that join points can be recursive, and that recursive join points open up a new and entirely unexpected (to us) optimization opportunity for fusion (Sec. 5). In particular, this insight fully resolves a long-standing tension between two competing approaches to fusion, namely stream fusion [6] and unfold/destroy fusion [26].
- We give some metatheory in Sec. 6, including type soundness and correctness of the optimizing transformations. We show the safety of adding jumps as a control effect by establishing an equivalence with System F.
- We demonstrate that our approach works at scale, in a state-of-the-art optimizing compiler for Haskell, GHC (Sec. 7). As hoped, adding join points turned out to be a very modest change, despite GHC's scale and complexity. Like any optimization, it does not make every program go faster, but it has a dramatic effect on some.

Overall, adding join points to ANF has an extremely good power-to-weight ratio, and we strongly recommend it to any direct-style compiler. Our title is somewhat tongue-in-cheek, but we now know of no optimizing transformation that is accessible to a CPS compiler but not to a direct-style one.

2. Motivation and Key Ideas

We review compilation techniques for commuting conversions, to expose the challenge that we tackle in this paper. For the sake of concreteness we describe the way things work in GHC. However, we believe that the whole paper is equally applicable to a call-by-value language.

The Case-of-Case Transformation Consider:

```
isNothing :: Maybe a -> Bool
isNothing x = case x of Nothing -> True
                        Just _  -> False

mHead :: [a] -> Maybe a
mHead ps = case ps of []    -> Nothing
                      (p:_) -> Just p

null :: [a] -> Bool
null as = isNothing (mHead as)
```

Here `null`[1] is a simple composition of the library functions `isNothing` and `mHead`. When the optimizer works on `null`, it will inline both `isNothing` and `mHead` to yield:

```
null as = case (case as of []    -> Nothing
                           (p:_) -> Just p) of
            { Nothing -> True; Just _ -> False }
```

Executed directly, this would be terribly inefficient; if the argument list is non-empty we would allocate a result `Just p` only to immediately decompose it. We want to move the outer case into the branches of the inner one, like this:

```
null as = case as of
            []  -> case Nothing of Nothing -> True
                                   Just z  -> False
            p:_ -> case Just p  of Nothing -> True
                                   Just _  -> False
```

[1] Haskell's standard `null` function returns whether a list is empty.

This is a commuting conversion, specifically the *case-of-case transformation*. In this example, it now happens that both inner `case` expressions scrutinize a data constructor, so they can be simplified, yielding

```
null as = case as of { [] -> True; _:_ -> False }
```

which is exactly the code we would have written for `null` from scratch.

GHC does a tremendous amount of inlining, including across modules or even packages, so commuting conversions like this are very important in practice: they are the key that unlocks a cascade of further optimizations.

Join Points Commuting conversions have a problem, though: *they often duplicate the outer* `case`. In our example that was OK, but what about

```
case (case v of { p1 -> e1; p2 -> e2 }) of
   { Nothing -> BIG1; Just x  -> BIG2 }
```

where `BIG1` and `BIG2` are big expressions? Duplicating these large expressions would risk bloating the compiled code, perhaps exponentially when `case` expressions are deeply nested [17]. It is easy to avoid this duplication by first introducing an auxiliary `let` binding:

```
let { j1 () = BIG1; j2 x  = BIG2 } in
case (case v of { p1 -> e1; p2 -> e2 }) of
  { Nothing -> j1 (); Just x  -> j2 x }
```

Now we can move the outer `case` expression into the arms of the inner `case`, without duplicating `BIG1` or `BIG2`, thus:

```
let { j1 () = BIG1; j2 x  = BIG2 } in
case v of
  p1 -> case e1 of Nothing -> j1 ()
                   Just x  -> j2 x
  p2 -> case e2 of Nothing -> j1 ()
                   Just x  -> j2 x
```

Notice that `j2` takes as its parameter the variable bound by the pattern `Just x`, whereas `j1` has no parameters[2].

Compiling Join Points Efficiently We call `j1` and `j2` *join points* because you can think of them as places where control joins up again, but so far they are perfectly ordinary `let`-bound functions, and as such they will be allocated as closures in the heap. But that's ridiculous: all that is happening here is control flow splitting and joining up again. A C compiler would generate a jump to a label, not a call to a heap-allocated function closure!

So, right before code generation, GHC performs a simple analysis to identify bindings that can be compiled as join points. This identifies `let`-bound functions that will never be captured in a closure or thunk, and will only be tail-called

[2] The dummy unit parameter is not necessary in a lazy language, but it is in a call-by-value language.

with exactly the right number of arguments. (We leave the exact criteria for Sec. 4.) These join-point bindings do not allocate anything; instead a tail call to a join point simply adjusts the stack and jumps to the code for the join point.

The case-of-case transformation, including the idea of using `let` bindings to avoid duplication, is very old; for example, both are features of Steele's Rabbit compiler for Scheme [24]. In Rabbit the transformation is limited to booleans, but the discussion above shows that it generalizes very naturally to arbitrary data types. In this more general form, it has been part of GHC for decades [19]. Likewise, the idea of generating more efficient code for non-escaping `let` bindings is well established in many other compilers [15, 23, 27], dating back to Rabbit and its `BIND-ANALYZE` routine [24].

Preserving and Exploiting Join Points So far so good, but there is a serious problem with recognizing join points only in the back end of the compiler. Consider this expression:

```
case (let j x = BIG in
      case v of { A -> j 1; B -> j 2; C -> True })
  of { True  -> False; False -> True }
```

Here `j` is a join point. Now suppose we do case-of-case on this expression. Treating the binding for `j` as an ordinary `let` binding (as GHC does today), we move the outer `case` past the `let`, and duplicate it into the branches of the inner `case`:

```
let j x = BIG in
case v of
  A -> case j 1  of { True -> False; False -> True }
  B -> case j 2  of { True -> False; False -> True }
  C -> case True of { True -> False; False -> True }
```

The third branch simplifies nicely, but the first two do not. There are two distinct problems:

1. The binding for `j` is no longer a join point (it is not tail-called), so the super-efficient code generation strategy does not apply, and the compiler will allocate a closure for `j` at runtime. This happens in practice: we have cases in which GHC's optimizer actually *increases* allocation because it inadvertently destroys a join point.
2. Even worse, the two copies of the outer `case` now scrutinize an uninformative call like `(j 1)`. So the extra code bloat from duplicating the outer `case` is entirely wasted. And it's a huge lost opportunity, as we shall see.

So *it is not enough to generate efficient code for join points; we must identify, preserve, and exploit them*. In our example, if the optimizer *knew* that the binding for `j` is a join point, it could exploit that knowledge to transform our original expression like this:

```
let j x = case BIG of True  -> False
                      False -> True
in case v of
  A -> j 1
  B -> j 2
  C -> case True of { True -> False; False -> True }
```

This is much, much better than our previous attempt:

- The outer `case` has moved into the right-hand side of the join point, so it now scrutinizes `BIG`. That's good, because `BIG` might be a data constructor or a `case` expression (which would expose another case-of-case opportunity). So the outer `case` now scrutinizes the actual result of the expression, rather than an uninformative join-point call. That solves problem (2).
- The `A` and `B` branches do not mention the outer `case`, because it has moved into the join point itself. So `j` is still tail-called and remains an efficiently-compiled join point. That solves problem (1).
- The outer `case` still scrutinizes the branches that do not finish with a join point call, e.g. the `C` branch.

The Key Idea Thus motivated, in the rest of this paper we explore the following very simple idea:

- Distinguish certain `let` bindings as *join-point bindings*, and their (tail-)call sites as *jumps*. This, by itself, is not new; see Section 9.
- Adjust the case-of-case transformation to take account of join-point bindings and jumps.
- In all the other transformations carried out by the compiler, ensure that join points remain join points.

Our key innovation is that, by recognising join points as a language construct, we both *preserve* join points through subsequent transformations and *exploit* them to make those transformations more effective. Next, we formalize this approach; subsequent sections develop the consequences.

3. System F_J: Join Points and Jumps

We now formalize the intuitions developed so far by describing System F_J, a small intermediate language with join points. F_J is an extension of GHC's Core intermediate language [19]. We omit existentials, GADTs, and coercions [25], since they are largely orthogonal to join points.

Syntax System F_J is a simple λ-calculus language in the style of System F, with **let** expressions, data type constructors, and **case** expressions; its syntax is given in Fig. 1. System F_J is an explicitly-typed language, so all binders are typed, but in our presentation we will often drop the types.

The join-point extension is highlighted in the figure and consists of two new syntactic constructs:

- A **join** binding declares a (possibly-recursive) join point. Each join point has a name, a list of type parameters, a list of value parameters, and a body.
- A **jump** expression invokes a join point, passing all indicated arguments as well as an additional result-type argument (as discussed shortly, under "The type of a join point").

Although we use curried syntax for **jump**s, join points are *polyadic*; partial application is not allowed.

Static Semantics The type system for System F_J is given in Fig. 2, where typeof gives the type of a constructor and ctors gives the set of constructors for a datatype.

Terms

x	$\in$	Term variables	
j	$\in$	Label variables	
e, u, v	$::=$	$x \mid l \mid \lambda x{:}\sigma.e \mid e\,u$	
	$\mid$	$\Lambda a.e \mid e\,\varphi$	Type polymorphism
	$\mid$	$K\,\vec{\varphi}\,\vec{e}$	Data construction
	$\mid$	$\textbf{case}\ e\ \textbf{of}\ \overrightarrow{alt}$	Case analysis
	$\mid$	$\textbf{let}\ vb\ \textbf{in}\ v$	Let binding
	$\mid$	$\textbf{join}\ jb\ \textbf{in}\ u$	Join-point binding
	$\mid$	$\textbf{jump}\ j\ \vec{\varphi}\ \vec{e}\ \tau$	Jump
alt	$::=$	$K\ \overrightarrow{x{:}\sigma} \to u$	Case alternative

Value bindings and join-point bindings

vb	$::=$	$x{:}\tau = e$	Non-recursive value
	$\mid$	$\textbf{rec}\ \overrightarrow{x{:}\tau = e}$	Recursive values
jb	$::=$	$j\ \vec{a}\ \overrightarrow{x{:}\sigma} = e$	Non-recursive join point
	$\mid$	$\textbf{rec}\ \overrightarrow{j\ \vec{a}\ \overrightarrow{x{:}\sigma} = e}$	Recursive join points

Answers

A	$::=$	$\lambda x{:}\sigma.e \mid \Lambda a.e \mid K\,\vec{\varphi}\,\vec{v}$

Types

a, b	$\in$	Type variables	
τ, σ, φ	$::=$	a	Variable
	$\mid$	T	Datatype
	$\mid$	$\sigma \to \tau$	Function type
	$\mid$	$\tau\,\varphi$	Application
	$\mid$	$\forall a.\,\tau$	Polymorphic type

Frames, evaluation contexts, and stacks

F	$::=$	$\square\, v$	Applied function
	$\mid$	$\square\, \tau$	Instantiated polymorphism
	$\mid$	$\textbf{case}\ \square\ \textbf{of}\ \overrightarrow{p \to u}$	Case scrutinee
	$\mid$	$\textbf{join}\ jb\ \textbf{in}\ \square$	Join point
E	$::=$	$\square \mid F[E]$	Evaluation contexts
s	$::=$	$\varepsilon \mid F : s$	Stacks

Tail contexts

L	$::=$	$\square$	Empty unary context
	$\mid$	$\textbf{case}\ e\ \textbf{of}\ \overrightarrow{p \to L}$	Case branches
	$\mid$	$\textbf{let}\ vb\ \textbf{in}\ L$	Body of let
	$\mid$	$\textbf{join}\ j\ \vec{a}\ \overrightarrow{x{:}\sigma} = L\ \textbf{in}\ L'$	Join point, body
	$\mid$	$\textbf{join rec}\ \overrightarrow{j\ \vec{a}\ \overrightarrow{x{:}\sigma} = L}\ \textbf{in}\ L'$	Rec join points, body

Miscellaneous

C	$\in$	General single-hole term contexts	
Σ	$::=$	$\cdot \mid \Sigma, x{:}\sigma = v$	Heap
c	$::=$	$\langle e;\ s;\ \Sigma\rangle$	Configuration

Figure 1: Syntax of System F_J.

The typing judgement carries two environments, Γ and Δ, with Δ binding join points. The environment Δ is extended by a **join** (rules JBIND and RJBIND) and consulted at a **jump**. Note that we rely on scoping conventions in some places: if $\Gamma;\ \Delta \vdash e : \tau$, then every variable (type or term) free in e or τ appears in Γ, and the symbols in Γ are unique. Similarly, every label free in e appears in Δ.

For jumps to truly be compiled as mere jumps, they must not occur in any subterm whose evaluation will take place in some unknown context. Otherwise, the jump would leave the stack in an unknown state. We enforce this invariant by resetting Δ to ε in every premise for such a subterm, for example in the premise for the body e in rule ABS.

The Type of a Join Point The type given to a join point deserves some attention. A join point that binds type variables $\vec{a}$ and value arguments of types $\vec{\sigma}$ is given the type $\forall\vec{a}.\,\vec{\sigma} \to \forall r.\,r$ (rule JBIND). Dually, a **jump** applies a join point to some type arguments (to instantiate $\vec{a}$), some value arguments (to saturate the $\vec{\sigma}$), and a final type argument (to instantiate r) that specifies the type returned by the **jump**. We put the universal quantification of r at the end to indicate that the argument types $\vec{\sigma}$ do not (and must not[3]) mention this "return-type parameter." Indeed, when we introduce the *abort* axiom (Sec. 3), it will need to change this type argument arbitrarily, which it can only safely do if the type is never actually used in the other parameters.

So a join point's result type $\forall r$ does not reflect the value of its body. What then keeps a join point from returning arbitrary values? It is the JBIND rule (or its recursive variant) that checks the right-hand side of the join point, making sure it is the same as that of the entire **join** expression. Thus we cannot have

$$\textbf{join}\ j = \texttt{"Gotcha!"}\ \textbf{in}\ \textbf{if}\ b\ \textbf{then}\ \textbf{jump}\ j\ \mathit{Int}\ \textbf{else}\ 4$$

because j returns a *String* but the body of the **join** returns an *Int*. In short, the burden of typechecking has moved: whereas a function can be declared to return any type but can only be invoked in certain contexts, a join point can be invoked in any context but can only return a certain type.

Finally, the reader may wonder why join points are polymorphic (apart from the result type). In F_J as presented here, we could manage with monomorphic join points, but they become absolutely necessary when we add data constructors that bind existential type variables. We omitted existentials from this paper for simplicity, but they are very important in practice and GHC certainly supports them.

Managing Δ The typing of join points is a little bit more flexible than you might suspect. Consider this expression:

$$\left(\begin{array}{l} \textbf{join}\ j\ x = \text{RHS} \\ \textbf{in}\ \textbf{case}\ v\ \textbf{of}\ A \to \textbf{jump}\ j\ \mathit{True}\ \mathit{C2C} \\ \qquad\qquad\quad B \to \textbf{jump}\ j\ \mathit{False}\ \mathit{C2C} \\ \qquad\qquad\quad C \to \lambda c.c \end{array}\right)\ \texttt{'x'}$$

where $\mathit{C2C} = \mathit{Char} \to \mathit{Char}$. This is certainly well typed. A valid transformation is to move the application to `'x'` into

$$\boxed{\Gamma;\,\Delta \vdash e : \tau}$$

$$\dfrac{(x{:}\tau) \in \Gamma}{\Gamma;\,\Delta \vdash x : \tau}\ \textsc{Var} \qquad \dfrac{\mathsf{typeof}(K) = \forall \vec{a}\,.\,\vec{\sigma} \to T\,\vec{a} \quad \overrightarrow{\Gamma;\,\varepsilon \vdash u : \sigma\{\varphi/a\}}}{\Gamma;\,\Delta \vdash K\ \vec{\varphi}\ \vec{u} : T\,\vec{\varphi}}\ \textsc{Con} \qquad \dfrac{\Gamma, (x{:}\sigma);\,\varepsilon \vdash e : \tau}{\Gamma;\,\Delta \vdash \lambda(x{:}\sigma).e : \sigma \to \tau}\ \textsc{Abs} \qquad \dfrac{\Gamma, a;\,\varepsilon \vdash e : \tau}{\Gamma;\,\Delta \vdash \Lambda a.e : \forall a.\,\tau}\ \textsc{TAbs}$$

$$\dfrac{\Gamma;\,\Delta \vdash e : \sigma \to \tau \quad \Gamma;\,\varepsilon \vdash u : \sigma}{\Gamma;\,\Delta \vdash e\,u : \tau}\ \textsc{App} \qquad \dfrac{\Gamma;\,\Delta \vdash e : \forall a.\,\tau}{\Gamma;\,\Delta \vdash e\,\varphi : \tau\{\varphi/a\}}\ \textsc{TApp} \qquad \dfrac{(j{:}\forall \vec{a}\,.\,\vec{\sigma} \to \forall r.\,r) \in \Delta \quad \overrightarrow{\Gamma;\,\varepsilon \vdash u : \sigma\{\overrightarrow{\varphi/a}\}}}{\Gamma;\,\Delta \vdash \mathbf{jump}\,j\ \vec{\varphi}\ \vec{u}\ \tau : \tau}\ \textsc{Jump}$$

$$\dfrac{\Gamma;\,\varepsilon \vdash u : \sigma \quad \Gamma, x{:}\sigma;\,\Delta \vdash e : \tau}{\Gamma;\,\Delta \vdash \mathbf{let}\,x{:}\sigma = u\,\mathbf{in}\,e : \tau}\ \textsc{VBind} \qquad \dfrac{\overrightarrow{\Gamma, \overrightarrow{x{:}\sigma};\,\varepsilon \vdash u : \sigma} \quad \Gamma, \overrightarrow{x{:}\sigma};\,\Delta \vdash e : \tau}{\Gamma;\,\Delta \vdash \mathbf{let\,rec}\,\overrightarrow{x{:}\sigma = u}\,\mathbf{in}\,e : \tau}\ \textsc{RVBind}$$

$$\dfrac{\Gamma, \vec{a}, \overrightarrow{x{:}\sigma};\,\Delta \vdash u : \tau \quad \Gamma;\,\Delta, (j{:}\forall \vec{a}\,.\,\vec{\sigma} \to \forall r.\,r) \vdash e : \tau}{\Gamma;\,\Delta \vdash \mathbf{join}\,j\ \vec{a}\ \overrightarrow{x{:}\sigma} = u\,\mathbf{in}\,e : \tau}\ \textsc{JBind}$$

$$\dfrac{\overrightarrow{\Gamma, \vec{a}, \overrightarrow{x{:}\sigma};\,\Delta, \overrightarrow{j{:}\forall \vec{a}\,.\,\vec{\sigma} \to \forall r.\,r} \vdash u : \tau} \quad \Gamma;\,\Delta, \overrightarrow{j{:}\forall \vec{a}\,.\,\vec{\sigma} \to \forall r.\,r} \vdash e : \tau}{\Gamma;\,\Delta \vdash \mathbf{join\,rec}\,\overrightarrow{j\ \vec{a}\ \overrightarrow{x{:}\sigma} = u}\,\mathbf{in}\,e : \tau}\ \textsc{RJBind}$$

$$\dfrac{\Gamma;\,\Delta \vdash e : T\,\vec{\varphi} \quad \overrightarrow{\mathsf{typeof}(K) = \forall \vec{a}\,.\,\vec{\sigma} \to T\,\vec{a}} \quad \overrightarrow{\vec{\nu} = \vec{\sigma}\{\overrightarrow{\varphi/a}\}} \quad \overrightarrow{\Gamma, \overrightarrow{x{:}\nu};\,\Delta \vdash u : \tau} \quad \mathsf{ctors}(T) = \{\vec{K}\}}{\Gamma;\,\Delta \vdash \mathbf{case}\,e\,\mathbf{of}\,\overrightarrow{K\ \overrightarrow{x{:}\nu} \to u} : \tau}\ \textsc{Case}$$

Figure 2: Type system for System F$_J$.

both the body *and* the right-hand side of the **join**, thus:

$$\begin{array}{l} \mathbf{join}\,j\,x = \mathrm{RHS}\ \texttt{'x'} \\ \mathbf{in} \begin{pmatrix} \mathbf{case}\,v\,\mathbf{of}\ A \to \mathbf{jump}\,j\ \mathit{True}\,C2C \\ \qquad\quad B \to \mathbf{jump}\,j\ \mathit{False}\,C2C \\ \qquad\quad C \to \lambda c.c \end{pmatrix} \texttt{'x'} \end{array}$$

Now we can move the application into the branches:

$$\begin{array}{l} \mathbf{join}\,j\,x = \mathrm{RHS}\ \texttt{'x'} \\ \mathbf{in\,case}\,v\,\mathbf{of}\ A \to (\mathbf{jump}\,j\ \mathit{True}\,C2C)\ \texttt{'x'} \\ \qquad\qquad\quad B \to (\mathbf{jump}\,j\ \mathit{False}\,C2C)\ \texttt{'x'} \\ \qquad\qquad\quad C \to (\lambda c.c)\ \texttt{'x'} \end{array}$$

Should this be well typed? The jumps to j are not exactly tail calls, but they can (and indeed must) discard their context—here the application to `'x'`—and resume execution at j. We will see shortly how this program can be further transformed to remove the redundant applications to `'x'`, but the point here is that this intermediate program should be well typed. That point is reflected in the typing rules by the fact that Δ is *not* reset in the function part of an application (rule App), or in the scrutinee of a **case** (rule Case).

Operational Semantics We give System F$_J$ an operational semantics (Fig. 3) in the style of an abstract machine. A *configuration* of the machine is a triple $\langle e;\ s;\ \Sigma\rangle$ consisting of an expression e which is the current focus of execution; a *stack* s representing the current evaluation context (including join-point bindings); and a *heap* Σ of value bindings. The stack is a list of *frames*, each of which is an argument to apply, a case analysis to perform, or a bound join point (or recursive group). Each frame is moved to the stack via the *push* rule. Since we define evaluation contexts by composing frames (hence $F[E]$ in Fig. 1), the rule has a simple form. Most of the

$$\boxed{\langle e;\ s;\ \Sigma\rangle \mapsto \langle e';\ s';\ \Sigma'\rangle}$$

$$\begin{array}{rcll} \langle F[e];\ s;\ \Sigma\rangle & \mapsto & \langle e;\ F : s;\ \Sigma\rangle & (\mathit{push}) \\ \langle \lambda x.e;\ \Box\,v : s;\ \Sigma\rangle & \mapsto & \langle e;\ s;\ \Sigma, x = v\rangle & (\beta) \\ \langle \Lambda a.e;\ \Box\,\varphi : s;\ \Sigma\rangle & \mapsto & \langle e\{\varphi/a\};\ s;\ \Sigma\rangle & (\beta_\tau) \\ \langle \mathbf{let}\,\mathit{vb}\,\mathbf{in}\,e;\ s;\ \Sigma\rangle & \mapsto & \langle e;\ s;\ \Sigma, \mathit{vb}\rangle & (\mathit{bind}) \\ \langle x;\ s;\ \Sigma[x = v]\rangle & \mapsto & \langle v;\ s;\ \Sigma[x = v]\rangle & (\mathit{look}) \\ \left\langle \begin{array}{c} K\,\vec{\varphi}\,\vec{v}; \\ \mathbf{case}\,\Box\,\mathbf{of}\ \overrightarrow{\mathit{alt}} : s; \\ \Sigma \end{array} \right\rangle & \mapsto & \langle u;\ s;\ \Sigma, \overrightarrow{x = v}\rangle & (\mathit{case}) \\ & & \text{if } (K\ \vec{x} \to u) \in \overrightarrow{\mathit{alt}} & \\ \left\langle \begin{array}{c} \mathbf{jump}\,j\ \vec{\varphi}\ \vec{v}\ \tau; \\ s' \mathbin{+\!\!+} (\mathbf{join}\,\mathit{jb}\,\mathbf{in}\,\Box : s); \\ \Sigma \end{array} \right\rangle & \mapsto & \left\langle \begin{array}{c} u\{\overrightarrow{\varphi/a}\}; \\ \mathbf{join}\,\mathit{jb}\,\mathbf{in}\,\Box : s; \\ \Sigma, \overrightarrow{x = v} \end{array} \right\rangle & (\mathit{jump}) \\ & & \text{if } (j\ \vec{a}\ \vec{x} = u) \in \mathit{jb} & \\ \left\langle \begin{array}{c} A; \\ \mathbf{join}\,\mathit{jb}\,\mathbf{in}\,\Box : s; \\ \Sigma \end{array} \right\rangle & \mapsto & \langle A;\ s;\ \Sigma\rangle & (\mathit{ans}) \end{array}$$

Figure 3: Call-by-name operational semantics for System F$_J$.

rules are quite conventional. We describe only call-by-name evaluation here, as rule *look* shows; switching to call-by-need by pushing an update frame is absolutely standard.

Note that only *value* bindings are put in the heap. Join points are stack-allocated in a frame: they represent mere code blocks, not first-class function closures. As expected, a jump throws away its context (the *jump* rule); it does so by popping all the frames from the stack to the binding (as usual, $\mathbin{+\!\!+}$ stands for the concatenation of two stacks):

$$\left\langle \begin{array}{l} \mathbf{join}\,j\,x = x \\ \mathbf{in\,case}\,(\mathbf{jump}\,j\,2\,(\mathit{Int} \to \mathit{Bool}))\,3\,\mathbf{of}\ldots;\ \varepsilon;\ \varepsilon \end{array} \right\rangle$$

$$\mapsto^\star \left\langle \begin{matrix} \mathbf{jump}\, j\, 2\, (\mathit{Int} \to \mathit{Bool}); \\ \square\, 3 : \mathbf{case}\, \square\, \mathbf{of} \ldots : \mathbf{join}\, j\, x = x\, \mathbf{in}\, \square : \varepsilon; \\ \varepsilon \end{matrix} \right\rangle$$
$$\mapsto \langle x;\, \mathbf{join}\, j\, x = x\, \mathbf{in}\, \square : \varepsilon;\ x = 2 \rangle$$

Here three frames are pushed onto the stack: the join-point binding, the case analysis, and finally the application of the jump to 3. Then the jump is evaluated, popping the latter two frames, replacing the term with the one from the join point, and binding the argument.

The ans rule removes a join-point binding from the context once an answer A (see Fig. 1) is computed; note that a well-typed answer cannot contain a jump, so at that point the binding must be dead code. Continuing our example:

$$\langle x;\, \mathbf{join}\, j\, x = x\, \mathbf{in}\, \square : \varepsilon;\ x = 2 \rangle \mapsto^\star \langle 2;\, \varepsilon;\ x = 2 \rangle$$

Optimizing Transformations The operational semantics operates on closed configurations. An optimizing compiler, by contrast, must transform *open* terms. To describe possible optimizations, then, we separately develop a sound *equational theory* (Fig. 4), which lays down the "rules of the game" by which the optimizer is allowed to work. It is up to the optimizer to determine how to apply the rules to rewrite code. All the axioms carry the usual implicit scoping restrictions to avoid free-variable capture. We spell out the side conditions in *drop* and *jdrop* because these actually restrict when the rule can be applied rather than merely ensuring hygiene. (In these side conditions, the bv function stands for "bound variables"; for instance, $\mathrm{bv}(x{:}\sigma = e) = \{x\}$.)

The β, β_τ, and *case* axioms are analogues of the similarly-named rules in the operational semantics. Since there is no heap, β and *case* create **let** expressions instead. Compile-time substitution, or *inlining*, is performed for values by *inline* and for join points by *jinline*. If a binding is inlined exhaustively, it becomes dead code and can be eliminated by the *drop* or *jdrop* axiom. Values may be substituted anywhere[4], which we indicate using a general single-hole context C in *inline*. Inlining of join points is a bit more delicate. A jump indicates *both* that we should execute the join point *and* that we should throw out the evaluation context up to the join point's declaration. Simply copying the body accomplishes the former but not the latter. For example:

$$\mathbf{join}\, j\, (x : \mathit{Int}) = x + 1\, \mathbf{in}\, (\mathbf{jump}\, j\, 2\, (\mathit{Int} \to \mathit{Int}))\, 3$$

If we naïvely inline j here, we end up with the ill-typed term:

$$\mathbf{join}\, j\, (x : \mathit{Int}) = x + 1\, \mathbf{in}\, (2 + 1)\, 3$$

Inlining is safe, however, if the jump is a tail call, since then there is no extra evaluation context to throw away. To specify the allowable places to inline a join point, then, we use a syntactic notion called a *tail context.* A tail context L (see Fig. 1) is a multi-hole context describing the places where a term may return to its evaluation context. Since $\square\, 3$ is not a tail context, the *jinline* axiom fails for the above term.

[4] For brevity, we have omitted rules allowing inlining a recursive definition into the definition itself (or another definition in the same recursive group).

The *casefloat*, *float*, *jfloat*, and *jfloat*$_{rec}$ axioms perform commuting conversions. The former two are conventional, but *jfloat* and *jfloat*$_{rec}$ exploit the new join-point construct to perform exactly the transformation we needed in Sec. 2 to avoid destroying a join point. The only difference between the two is that *jfloat*$_{rec}$ acts on a recursive binding; the operation performed is the same.

Consider again the example at the beginning of Sec. 2. With our new syntax, we can write it as:

$$\begin{array}{l} \mathbf{case} \left(\begin{array}{l} \mathbf{join}\, j\, x = \mathrm{BIG} \\ \mathbf{in\, case}\, v\, \mathbf{of}\, A \to \mathbf{jump}\, j\, 1\, \mathit{Bool} \\ \qquad\qquad\quad B \to \mathbf{jump}\, j\, 2\, \mathit{Bool} \\ \qquad\qquad\quad C \to \mathit{True} \end{array} \right) \mathbf{of} \\ \quad \{\mathit{True} \to \mathit{False}; \mathit{False} \to \mathit{True}\} \end{array}$$

We can use *jfloat* to move the outer **case** both into the right-hand side of the **join** binding and into its body; use *casefloat* to move the outer **case** into the branches of the inner **case**; use abort to discard the outer **case** where it scrutinizes a **jump**; and use *case* to simplify the C alternative. The result is just what we want:

$$\begin{array}{l} \mathbf{join}\, j\, x = \mathbf{case}\, \mathrm{BIG}\, \mathbf{of}\, \{\mathit{True} \to \mathit{False}; \mathit{False} \to \mathit{True}\} \\ \mathbf{in\, case}\, v\, \mathbf{of}\, A \to \mathbf{jump}\, j\, 1\, \mathit{Bool} \\ \qquad\qquad\quad B \to \mathbf{jump}\, j\, 2\, \mathit{Bool} \\ \qquad\qquad\quad C \to \mathit{False} \end{array}$$

The** commute **Axiom The left-hand sides of axioms *float*, *jfloat*, *jfloat*$_{rec}$, and *casefloat* enumerate the forms of a tail context L (Figure 1). So the four axioms are all instances of a single equivalent form:

$$E[L[\overrightarrow{e}]] = L[\overrightarrow{E[e]}] \quad (\mathit{commute})$$

This rule *commute* moves the evaluation context E into *each* hole of the tail context L.[5]

We can also derive new axioms succinctly using tail contexts. For example, our commuting conversions as written risk quite a bit of code duplication by copying E arbitrarily many times (into each branch of a **case** and each join point). Of course, in a real implementation, we would prefer not to do this, so instead we might use a different axiom:

$$E[L[\overrightarrow{e}] : \tau] = \mathbf{join}\, j\, x = E[x]\, \mathbf{in}\, L[\overrightarrow{\mathbf{jump}\, j\, e\, \tau}]$$

This can be derived from *commute* by first applying *jdrop* and *jinline* backward.

4. Contification: Inferring Join Points

Not all join points originate from commuting conversions. Though the source language doesn't have join points or jumps, many **let**-bound functions can be converted to join points without changing the meaning of the program. In particular,

[5] Since L has a hole wherever something is returned to E, *commute* "substitutes" the latter into the places where it is invoked. In fact, from a CPS standpoint, *commute* (in concert with abort) is *precisely* a substitution operation.

$$\boxed{e = e'}$$

$$\begin{array}{rclll}
(\lambda x{:}\sigma.e)\,v & = & \mathbf{let}\,x{:}\sigma = v\,\mathbf{in}\,e & & (\beta)\\
(\Lambda a.e)\,\varphi & = & e\{\varphi/a\} & & (\beta_\tau)\\
\mathbf{let}\,\mathit{vb}\,\mathbf{in}\,C[x] & = & \mathbf{let}\,\mathit{vb}\,\mathbf{in}\,C[v] & \text{if } (x{:}\sigma = v) \in \mathit{vb} & (\mathit{inline})\\
\mathbf{let}\,\mathit{vb}\,\mathbf{in}\,e & = & e & \text{if } \mathrm{bv}(\mathit{vb}) \cap \mathrm{fv}(e) = \emptyset & (\mathit{drop})\\
\mathbf{join}\,\mathit{jb}\,\mathbf{in}\,L[\vec{e}, \mathbf{jump}\,j\,\vec{\varphi}\,\vec{v}\,\tau, \vec{e'}] & = & \mathbf{join}\,\mathit{jb}\,\mathbf{in}\,L[\vec{e}, \mathbf{let}\,\overrightarrow{x{:}\sigma = v}\,\mathbf{in}\,u\{\overrightarrow{\varphi/a}\}, \vec{e'}] & \text{if } (j\,\vec{a}\,\overrightarrow{x{:}\sigma} = u) \in \mathit{jb} & (\mathit{jinline})\\
\mathbf{join}\,\mathit{jb}\,\mathbf{in}\,e & = & e & \text{if } \mathrm{bv}(\mathit{jb}) \cap \mathrm{fv}(e) = \emptyset & (\mathit{jdrop})\\
\mathbf{case}\,K\,\vec{\varphi}\,\vec{v}\,\mathbf{of}\,\overrightarrow{\mathit{alt}} & = & \mathbf{let}\,\overrightarrow{x{:}\sigma = v}\,\mathbf{in}\,e & \text{if } (K\,\overrightarrow{x{:}\sigma} \to e) \in \overrightarrow{\mathit{alt}} & (\mathit{case})\\
E[\mathbf{case}\,e\,\mathbf{of}\,\overrightarrow{K\,\vec{x} \to u}] & = & \mathbf{case}\,e\,\mathbf{of}\,\overrightarrow{K\,\vec{x} \to E[u]} & & (\mathit{casefloat})\\
E[\mathbf{let}\,\mathit{vb}\,\mathbf{in}\,e] & = & \mathbf{let}\,\mathit{vb}\,\mathbf{in}\,E[e] & & (\mathit{float})\\
E[\mathbf{join}\,j\,\vec{a}\,\vec{x} = u\,\mathbf{in}\,e] & = & \mathbf{join}\,j\,\vec{a}\,\vec{x} = E[u]\,\mathbf{in}\,E[e] & & (\mathit{jfloat})\\
E[\mathbf{join\,rec}\,\overrightarrow{j\,\vec{a}\,\vec{x} = u}\,\mathbf{in}\,e] & = & \mathbf{join\,rec}\,\overrightarrow{j\,\vec{a}\,\vec{x} = E[u]}\,\mathbf{in}\,E[e] & & (\mathit{jfloat}_{\mathit{rec}})\\
E[\mathbf{jump}\,j\,\vec{\varphi}\,\vec{e}\,\tau] : \tau' & = & \mathbf{jump}\,j\,\vec{\varphi}\,\vec{e}\,\tau' & & (\mathit{abort})
\end{array}$$

Figure 4: Common optimizations for System F_J.

$$\boxed{e = e'}$$

$$\begin{array}{rcll}
\mathbf{let}\,f = \Lambda\vec{a}.\lambda\vec{x}.u\,\mathbf{in}\,L[\vec{e}] : \tau & = & \mathbf{join}\,j\,\vec{a}\,\vec{x} = u\,\mathbf{in}\,L[\overrightarrow{\mathsf{tail}_\rho(e)}] & (\mathit{contify})\\
& & \quad \text{if } \rho(f\;\vec{a}\;\vec{x}) = \mathbf{jump}\,j\,\vec{a}\,\vec{x}\,\tau & \\
& & \quad \text{and } f \notin \mathrm{fv}(L), u : \tau & \\
\mathbf{let\,rec}\,\overrightarrow{f = \Lambda\vec{a}.\lambda\vec{x}.L[\vec{u}]}\,\mathbf{in}\,L'[\vec{e}] : \tau & = & \mathbf{join\,rec}\,\overrightarrow{j\,\vec{a}\,\vec{x} = L[\overrightarrow{\mathsf{tail}_\rho(u)}]}\,\mathbf{in}\,L'[\overrightarrow{\mathsf{tail}_\rho(e)}] & (\mathit{contify}_{\mathit{rec}})\\
& & \quad \text{if } \overrightarrow{\rho(f\;\vec{a}\;\vec{x}) = \mathbf{jump}\,j\,\vec{a}\,\vec{x}\,\tau} & \\
& & \quad \text{and } \overrightarrow{f \notin \mathrm{fv}(\vec{L})}, f \notin \mathrm{fv}(L'), \overrightarrow{L[\vec{u}] : \tau} &
\end{array}$$

$$\begin{array}{rcll}
\mathsf{tail}_\rho(f\;\vec{\sigma}\;\vec{u}) & \triangleq & e\{\overrightarrow{\sigma/a}\}\{\overrightarrow{u/x}\} & \text{if } \rho(f\;\vec{a}\;\vec{x}) = e \text{ and } \mathrm{dom}(\rho) \cap \mathrm{fv}(\vec{u}) = \emptyset\\
\mathsf{tail}_\rho(e) & \triangleq & e & \text{if } \mathrm{dom}(\rho) \cap \mathrm{fv}(e) = \emptyset\\
\mathsf{tail}_\rho(e) & \triangleq & \text{undefined} & \text{otherwise}
\end{array}$$

Figure 5: Contification as a source-to-source transformation.

if *every* call to a given function is a saturated tail call (i.e. appears only in an L-context), and we turn the calls into jumps, then whenever one of the jumps is executed, there will be nothing to drop from the evaluation context (the s' in *jump* will be empty).

The process is a form of *contification* [16] (or *continuation demotion*), which we formalize in Fig. 5, where $\mathrm{fv}(e)$ means the set of free variables of e (and similarly $\mathrm{fv}(L)$ for tail contexts), and $\mathrm{dom}(\rho)$ means the domain of the environment ρ (to be described shortly).

The non-recursive version, *contify*, attempts to decompose the body of the **let** (*i.e.,* the scope of f) into a tail context L and its arguments, where the arguments contain all the occurrences of f, then attempts to run the special partial function tail on each argument to the tail context. This function will only succeed if there are no non-tail calls to f.

The tail function takes an environment ρ mapping applications of contifiable variables f to jumps to corresponding join points j. For each expression that matches the form of a saturated call to such an f, then, tail turns the call into a jump to its j, *provided* that none of the arguments to the function contains a free occurrence of a variable being contified—an occurrence in argument position is disallowed by the typing rules. For any other expression, tail changes nothing but does check that no variable being contified appears; otherwise, tail fails, causing the *contify* axiom not to match.

There is one last proviso in the *contify* and $\mathit{contify}_{\mathit{rec}}$ axioms, which is that the body of each function to be contified must have the same type as the body of the **let**. This can fail if some function f is polymorphic in its return type [8].

Finding bindings to which *contify* or $\mathit{contify}_{\mathit{rec}}$ will apply is not difficult. Our implementation is essentially a free-variable analysis that also tracks whether each free variable has appeared *only* in the holes of tail contexts. This is much simpler than previous contification algorithms because we *only look for tail calls*. We invite the reader to compare to [11] or to Sec. 5 of [16], which both allow for more general calls to be dealt with. Yet we claim that, in concert with the Simplifier and the Float In pass, our algorithm covers most of the same ground.

To demonstrate, consider the *local CPS transformation* in Moby [23], which produces mutually tail-recursive functions to improve code generation in much the same way GHC does. Moby uses a direct-style intermediate representation, but its contification pass is expressed in terms of a CPS transform, which turns

```
let f x = ... in E[... f y ... f z ...]
```

(where the calls to `f` are tail calls within `E`) into

```
let { j x = E[x]; f x = j <rhs> }
in ...f y...f z...
```

where the tail calls to `f` are now compiled as jumps. Note that `f` now matches the *contify* axiom, but it did not before due to the E in the way. Nonetheless, our extended GHC achieves the same effect, only in stages. Starting with:

$$\mathbf{let}\, f\, x = \text{rhs}\,\mathbf{in}\, E[\ldots f\, y \ldots f\, z \ldots]$$

First, applying *float* from right to left floats f inward:

$$E[\mathbf{let}\, f\, x = \text{rhs}\,\mathbf{in}\, \ldots f\, y \ldots f\, z \ldots]$$

Next, *contify* applies, since the calls to f are now tail calls:

$$E[\mathbf{join}\, f\, x = \text{rhs}\,\mathbf{in}\, \ldots \mathbf{jump}\, f\, y\, \tau \ldots \mathbf{jump}\, f\, z\, \tau \ldots]$$

And now *jfloat* pushes E into the join point f and the body:

$$\mathbf{join}\, f\, x = E[\text{rhs}]\,\mathbf{in}\, \ldots E[\mathbf{jump}\, f\, y\, \tau] \ldots E[\mathbf{jump}\, f\, z\, \tau] \ldots]$$

From here, *abort* removes E from the jumps, and we can abstract E by running *jdrop* and *jinline* backward:

$$\mathbf{join}\, \{j\, x = E[x]; f\, x = \mathbf{jump}\, j\, \text{rhs}\, \tau\}\,\mathbf{in} \ldots f\, y \ldots f\, z \ldots$$

Thus we achieve the same result without any extra effort[6].

Naturally, contification is more routine and convenient in CPS-based compilers [11, 16]. The ability to handle an intervening context comes nearly "for free" since contexts already have names. Notably, it is still possible to name contexts in direct style (the Moby paper [23] does so using labelled expressions), so it is *only* a matter of convenience.

5. Recursive Join Points and Fusion

We have mentioned, without stressing the point, that join points can be recursive. We have also shown that it is rather easy to identify let-bindings that can be re-expressed (more efficiently) as join points. To our complete surprise, we discovered that the combination of these two features allowed us to solve a long-standing problem with stream fusion.

Recursive Join Points Consider this program, which finds the first element of a list that satisfies a predicate p:

$$\begin{array}{l} \mathit{find} = \Lambda a.\lambda(p : a \to \mathit{Bool})(xs : [a]). \\ \quad \mathbf{let}\, \mathit{go}\ xs = \mathbf{case}\, xs\, \mathbf{of} \\ \qquad\qquad x : xs' \to \mathbf{if}\, p\ x\, \mathbf{then}\, \mathit{Just}\ x \\ \qquad\qquad\qquad\qquad\quad \mathbf{else}\ \ \mathit{go}\ xs' \\ \qquad\qquad [\,] \qquad\ \to \mathit{Nothing} \\ \quad \mathbf{in}\, \mathit{go}\ xs_0 \end{array}$$

Programmers quite often write loops like this, with a local definition for go, perhaps to allow *find* to be inlined at a call site. Our first observation is this: go is a (recursive) join point! The contification transformation of will identify go as a join point, and will transform the **let** (which allocates) to a **join** (which does not), and each call to go into an efficient **jump**.

But it gets better! Because go is a join point, it can participate in a commuting conversion. Suppose, for example, that *find* is called from any like this:

$$\begin{array}{l} \mathit{any} = \Lambda a.\lambda(p : a \to \mathit{Bool})(xs : [a]). \\ \qquad \mathbf{case}\, \mathit{find}\, p\, xs\, \mathbf{of}\ \mathit{Just}\, \_ \quad \to \mathit{True} \\ \qquad\qquad\qquad\qquad\quad \mathit{Nothing} \to \mathit{False} \end{array}$$

The call to *find* can be inlined:

$$\begin{array}{l} \mathit{any} = \Lambda a.\lambda(p : a \to \mathit{Bool})(xs : [a]). \\ \quad \mathbf{case} \left(\begin{array}{l} \mathbf{join}\, \mathit{go}\ xs = \mathbf{case}\, xs\, \mathbf{of} \\ \quad x : xs' \to \mathbf{if}\, p\, x\, \mathbf{then}\, \mathit{Just}\ x \\ \qquad\qquad\qquad \mathbf{else}\ \ \mathbf{jump}\, \mathit{go}\ xs'\ (\mathit{Maybe}\ a) \\ \quad [\,] \qquad \to \mathit{Nothing} \\ \mathbf{in}\, \mathbf{jump}\, \mathit{go}\ xs\ (\mathit{Maybe}\ a) \end{array} \right) \mathbf{of} \\ \quad \{\mathit{Just}\, \_ \to \mathit{True}; \mathit{Nothing} \to \mathit{False}\} \end{array}$$

Now, we have a **case** scrutinizing a **join** so we can apply axiom *jfloat* from Figure 4. After some easy further transformations, we get

$$\begin{array}{l} \mathit{any} = \Lambda a.\lambda(p : a \to \mathit{Bool})(xs : [a]). \\ \quad \mathbf{join}\, \mathit{go}\ xs = \mathbf{case}\, xs\, \mathbf{of} \\ \qquad x : xs' \to \mathbf{if}\, p\, x\, \mathbf{then}\, \mathit{True} \\ \qquad\qquad\qquad\quad \mathbf{else}\ \ \mathbf{jump}\, \mathit{go}\ xs'\ \mathit{Bool} \\ \qquad [\,] \qquad \to \mathit{False} \\ \quad \mathbf{in}\, \mathbf{jump}\, \mathit{go}\ xs\ \mathit{Bool} \end{array}$$

Look carefully at what has happened here: the consumer (any) of a recursive loop (go) has moved *all the way to the return point of the loop*, so that we were able to cancel the **case** in the consumer with the data constructor returned at the conclusion of the loop.

Stream Fusion It turns out that this new ability to move a consumer all the way to the return points of a tail-recursive loop has direct implications for a very widely used transformation: stream fusion. The key idea of stream fusion is to represent a list (or array, or other sequence) by a pair of a *state* and a *stepper function*, thus:[7]

```
data Stream a where
  MkStream :: s -> (s -> Step s a) -> Stream a
```

There are two competing approaches to the `Step` type. In unfold/destroy fusion (Svenningsson [26]), we have:

```
data Step s a = Done | Yield s a
```

Hence a stepper function takes an incoming state and either yields an element and a new state or signals the end. Now

[6] The parts of this sequence not specifically to do with join points were already implemented before in GHC: The Float In pass applies *float* in reverse, and the Simplifier regularly creates join points to share evaluation contexts (except that previously they were ordinary **let** bindings).

[7] Note that `Stream` is an existential type, so as to abstract the internal state type `s` as an implementation detail of the stream.

a pipeline of list processors can be rewritten as a pipeline of stepper functions, each of which produces and consumes elements one by one. A typical stepper function for a stream transformer looks like:

```
  next s = case <incoming step> of
             Yield s' a -> <process element>
             Done       -> <process end of stream>
```

When composed together and inlined, the stepper functions become a nest of cases, each scrutinizing the output of the previous stepper. It is crucial for performance that each Yield or Done expression be matched to a case, much as we did with Just and Nothing in the example that began Sec. 2. Fortunately, case-of-case and the other commuting conversions that GHC performs are usually up to the task.

Alas, this approach requires a recursive stepper function when implementing filter, which must loop over incoming elements until it finds a match. This breaks up the chain of cases by putting a loop in the way, much as our *any* above becomes a **case** on a loop. Hence until now, recursive stepper functions have been un-fusible. Coutts *et al.* [6] suggested adding a Skip construtor to Step, thus:

```
data Step s a = Done | Yield s a | Skip s
```

Now the stepper function can say to update the state and call again, obviating the need for a loop of its own. This makes filter fusible, but it complicates everything else! Everything gets three cases instead of two, leading to more code and more runtime tests; and functions like zip that consume two lists become more complicated and less efficient.

But with join points, just as with *any*, Svenningsson's original Skip-less approach fuses just fine! Result: simpler code, less of it, and faster to execute. It's a straight win.

6. Metatheory of F_J

Proofs can be found in the extended version of this paper[8].

Correctness and Type Safety The way to "run" a program on our abstract machine is to initialize the machine with an empty stack and an empty store. Type safety, then, says that once we start the machine, the program either runs forever or successfully returns an answer.

Proposition 1 (Type safety). *If $\varepsilon; \varepsilon \vdash e : \tau$, then either:*
1. *The initial configuration $\langle e;\ \varepsilon;\ \varepsilon\rangle$ diverges, or*
2. *$\langle e;\ \varepsilon;\ \varepsilon\rangle \mapsto^\star \langle A;\ \varepsilon;\ \Sigma\rangle$, for some store Σ and answer A.*

To establish the correctness of our rewriting axioms, we first define a notion of *observational equivalence*.

Definition 2. *Two terms e and e' are* observationally equivalent*, written $e \cong e'$, if, given any context C, $\langle C[e];\ \varepsilon;\ \varepsilon\rangle$ diverges if and only if $\langle C[e'];\ \varepsilon;\ \varepsilon\rangle$ diverges.*

The equational theory is sound with respect to $\cong$:

[8] https://www.microsoft.com/en-us/research/publication/compiling-without-continuations/

Proposition 3. *If $e = e'$, then $e \cong e'$.*

Equivalence to System F The best way to be sure that F_J can be implemented without headaches is to show that it is equivalent to GHC's existing System F-based language. This would suggest that join points do not allow us to write any *new* programs, only to implement existing programs more efficiently. To prove the equivalence, we establish an *erasure* procedure that removes all join points from an F_J term, leaving an equivalent System F term.

To erase the join points, we want to apply the *contify* axiom (or its recursive variant) from right to left. However, we cannot necessarily do so immediately for each join point, since *contify* only applies when all invocations are in tail position. For example, we cannot de-contify j here:

$$\mathbf{join}\, j\, x = x + 1\, \mathbf{in}\, (\mathbf{jump}\, j\, 1\, (\mathit{Int} \to \mathit{Int}))\, 2$$

Simply rewriting the join point as a function and the jump as a function call would change the meaning of the program—in fact, it would not even be well-typed:

$$\mathbf{let}\, f = \lambda x.x + 1\, \mathbf{in}\, f\, 1\, 2$$

However, if we apply *abort* first:

$$\mathbf{join}\, j\, x = x + 1\, \mathbf{in}\, \mathbf{jump}\, j\, 1\, \mathit{Int}$$

Now the jump is a tail call, so *contify* applies.

The *abort* axiom is not enough on its own, since the jump may be buried inside a tail context:

$$\mathbf{join}\, j\, x = x + 1\, \mathbf{in} \begin{pmatrix} \mathbf{case}\, b\, \mathbf{of} \\ \quad \mathit{True} \;\to \mathbf{jump}\, j\, 1\, (\mathit{Int} \to \mathit{Int}) \\ \quad \mathit{False} \to \mathbf{jump}\, j\, 3\, (\mathit{Int} \to \mathit{Int}) \end{pmatrix} 2$$

However, this can be handled by a commuting conversion:

$$\begin{aligned} &\mathbf{join}\, j\, x = x + 1\, \mathbf{in}\, \mathbf{case}\, b\, \mathbf{of} \\ &\qquad\qquad \mathit{True} \;\to (\mathbf{jump}\, j\, 1\, (\mathit{Int} \to \mathit{Int}))\, 2 \\ &\qquad\qquad \mathit{False} \to (\mathbf{jump}\, j\, 3\, (\mathit{Int} \to \mathit{Int}))\, 2 \end{aligned}$$

And now *abort* applies twice and j can be de-contified.

Lemma 4. *For any well-typed term e, there is an e' such that $e' = e$ and every jump in e' is in tail position.*

By "tail position," we mean one of the holes in a tail context that starts with the binding for the join point being called. In other words, given a term

$$\mathbf{join}\, j\, \overrightarrow{a}\, \overrightarrow{x} = u\, \mathbf{in}\, L[\overrightarrow{e}],$$

the terms $\overrightarrow{e}$ are in tail position for j.

The proof of Lemma 4 relies on the observation that the places in a term that may contain free occurrences of labels are precisely those appearing in the hole of either an evaluation or a tail context. For example, the CASE typing rule propagates Δ into both the scrutinee and the branches; note that $\mathbf{case}\, \square\, \mathbf{of}\, \overrightarrow{alt}$ is an evaluation context and $\mathbf{case}\, e\, \mathbf{of}\, \overrightarrow{p \to \square}$ is a tail context. But $e\, \square$ is (in call-by-name) neither an evaluation context *nor* a tail context, and APP does *not* propagate Δ into the argument.

Thus *any* expression can be written as:

$$L[\overrightarrow{E[\overrightarrow{L'[E'[\ldots[L^{(n)}[\overrightarrow{E^{(n)}[e]}]]\ldots]]}]}], \qquad (1)$$

which is to say a tree of tail contexts alternating with evaluation contexts, where all free occurrences of join points are at the leaves. By iterating $commute$ and $abort$, we can flatten the tree, rewriting (1) to say that any expression can be written $L[\overrightarrow{e}]$, where each e_i is a leaf from the tree in (1). In this form, no e_i can be expressed as $E[L[\ldots]]$ for nontrivial, non-binding[9] E and nontrivial L, and every jump to a free occurrence of a label is some e_i. Let us say a term in the above form is in *commuting-normal form*. (Note that ANF is simply commuting-normal form with named intermediate values.) By $commute$ and $abort$, every term has a commuting-normal form, and by construction, every jump in a commuting-normal form is a tail call. Thus *every label can be decontified*, and we have:

Theorem 5 (Erasure)**.** *For any closed, well-typed* F_J *term* e*, there is a System F term* e' *such that* $e' = e$*.*

7. Join Points in Practice

Is is one thing to define a calculus, but quite another to use it in a full-scale optimising compiler. In this section we report on our experience of doing so in GHC.

Implementing Join Points in GHC We have implemented System F_J as an extension to the Core language in GHC. As a representation choice, instead of adding two new data constructors for **join** and **jump** to the Core data type, we instead re-use ordinary **let**-bindings and function applications, distinguishing join points only by a flag *on the identifier itself*.

Thus, with no code changes, GHC treats join-point identifiers identically to other identifiers, and join-point bindings identically to ordinary **let** bindings. This is extremely convenient in practice. For example, all the code that deals with dropping dead bindings, inlining a binding that occurs just once, inlining a binding whose right-hand side is small, and so on, all works automatically for join points too.

With the modified Core language in hand, we had three tasks. First, GHC has an internal typechecker, called Core Lint, that (optionally) checks the type-correctness of the intermediate program after each pass. We augmented Core Lint for F_J according to the rules of Fig. 2.

Second, we added a simple new contification analysis to identify let-bindings that can be converted into join points (see Sec. 4). Since the analysis is simple, we run it frequently, whenever the so-called occurrence analyzer runs.

Finally, the new Core Lint forensically identified several existing Core-to-Core passes that were "destroying" join points (see Sec. 2). Destroying a join point de-optimizes the program, so it is wonderful now to have a way to nail such problems at their source. Moreover, once Core Lint flagged a problem, it was never difficult to alter the Core-to-Core transformation to make it preserve join points. Here are some of the specifics about particular passes:

The Simplifier is a sort of partial evaluator responsible for many local transformations, including commuting conversions and inlining [19]. The Simplifier is implemented as a tail-recursive traversal that builds up a representation of the evaluation context as it goes; as such, implementing the *jfloat* and $abort$ axioms (Sec. 3) requires only two new behaviors:

- (*jfloat*) When traversing a join-point binding, copy the evaluation context into the right-hand side.
- ($abort$) When traversing a jump, throw away the evaluation context.

The Float Out pass moves **let** bindings outwards [20]. Moving a **join** binding outwards, however, risks destroying the join point, so we modified Float Out to leave **join** bindings alone in most cases.

The Float In pass moves **let** bindings inwards. It too can destroy join points by un-saturating them. For example, given `let j x y = ... in j 1 2`, the Float In pass wants to narrow j's scope as much as possible: `(let j x y = ... in j) 1 2`. We modified Float In so that it never un-saturates a join point.

Strictness analysis is as useful for join points as it is for ordinary **let** bindings, so it is convenient that **join** bindings are, by default, treated identically to ordinary **let** bindings. In GHC, the results of strictness analysis are exploited by the so-called worker/wrapper transform [12, 19]. We needed to modify this transform so that the generated worker and wrapper are both join points. We found that GHC's *constructed product result* (CPR) analysis [3] caused the wrapper to invoke the worker inside a **case** expression, thus preventing the worker from being a join point. We simply disable CPR analysis for join points; it turns out that the commuting conversions for join points do a better job anyway.

Benchmarks The reason for adding join points is to improve performance; expressiveness is unchanged (Sec. 6). So does performance improve? Table 1 presents benchmark data on allocations, collected from the standard `spectral`, `real` and `shootout` NoFib benchmark suites[10]. We ran the tests on our modified GHC branch, and compared them to the GHC baseline to which our modifications were applied. Remember, the baseline compiler *already* recognises join points in the back end and compiles them efficiently (Sec. 2); the performance changes here come from preserving and exploiting join points during optimization.

We report only heap allocations because they are a repeatable proxy for runtime; the latter is much harder to measure

[9] A **join** can be treated as either an evaluation context or a tail context; using $commute$ to push a **join** inward is not necessarily helpful, however.

[10] The `imaginary` suite had no interesting cases. We believe this is because join points tend to show up only in fairly large functions, and the `imaginary` tests are all micro-benchmarks.

`spectral`

Program	Allocs
fibheaps	-1.1%
ida	-1.4%
nucleic2	+0.2%
para	-4.3%
primetest	-3.6%
simple	-0.9%
solid	-8.4%
sphere	-3.3%
transform	+1.1%
(45 others)	
Min	-8.4%
Max	+1.1%
Geo. Mean	-0.4%

`real`

Program	Allocs
anna	+0.5%
cacheprof	-0.5%
fem	+3.6%
gamteb	-1.4%
hpg	-2.1%
parser	+1.2%
rsa	-4.7%
(18 others)	
Min	-4.7%
Max	+3.6%
Geo. Mean	-0.2%

`shootout`

Program	Allocs
k-nucleotide	-85.9%
n-body	-100.0%
spectral-norm	-0.8%
(5 others)	
Min	-100.0%
Max	+0.0%
Geo. Mean	n/a

Table 1: Benchmarks from the `spectral`, `real`, and `shootout` NoFib suites.

reliably. All tests omitted from the tables had an improvement in allocations, but less than 0.3%.

There are some startling figures: using join points eliminated *all* allocations in `n-body` and 85.9% in `k-nucleotide`. We caution that these are highly atypical programs, already hand-crafted to run fast. Still, it seems that our work may make it easier for performance-hungry authors to squeeze more performance out of their inner loops.

The complex interaction between inlining and other transformations makes it impossible to *guarantee* improvements. For example, improving a function f might make it small enough to inline into g, but this may cause g to become too *large* to inline elsewhere, and that in turn may lose the optimization opportunities previously exposed by inlining g. GHC's approach is heuristic, aiming to make losses unlikely, but they do occur, including a 1.1% increase in allocations in `spectral/transform` and a 3.6% increase in `real/fem`.

Beyond Benchmarks These benchmarks show modest but fairly consistent improvements for existing, unmodified programs. But we believe that the systematic addition of join points may have a more significant effect on programming patterns. Our discussion of fusion in Sec. 5 is a case in point: with join points we can use skip-less unfoldr/destroy streams without sacrificing fusion. That knowledge in turn affects the way libraries are written: they can be smaller and faster.

Moreover, the transformation pipeline becomes more robust. In GHC today, if a "join point" is inlined we get good fusion behavior, but if its size grows to exceed the (arbitrary) inlining threshold, suddenly behavior becomes much worse. An innocuous change in the source program can lead to a big change in execution time. That step-change problem disappears when we formally add join points.

8. Why Not Use Continuation-Passing Style?

Our join points are, of course, nothing more than continuations, albeit second-class continuations that do not escape, and thus can be implemented efficiently. So why not just use CPS? Kennedy's work makes a convincing argument for CPS as a language in which to perform optimization [16].

There are many similarities between Kennedy's work and ours. Notably, Kennedy distinguishes ordinary bindings (**let**) from continuation bindings (**letcont**), just as we distinguish ordinary bindings from join points (**join**); similarly, he distinguishes continuation invocations (i.e. jumps) from ordinary function calls, and we follow suit. But there are a number of reasons to prefer direct style, if possible:

- Direct style is, well, more direct. Programs are simply easier to understand, and the compiler's optimizations are easier to follow. Although it sounds superficial, in practice it is a significant advantage of direct style; for example Haskell programmers often pore over the GHC's Core dumps of their programs.
- The translation into CPS encodes a particular order of evaluation, whereas direct style does not. That dramatically inhibits code-motion transformations. For example, GHC does a great deal of "**let** floating" [20], in which a **let** binding is floated outwards or inwards, which is valid for pure (effect-free) bindings. This becomes harder or impossible in CPS, where the order of evaluation is prescribed.
  Fixing the order of evaluation is a particular issue when compiling a call-by-need language, since the known call-by-need CPS transform [18] is quite involved.
- Some transformations are much harder in CPS. For example, consider common sub-expression elimination (CSE). In `f (g x) (g x)`, the common sub-expression is easy to see. But it is much harder to find in the CPS version:
```
letcont k1 xv = letcont k2 yv = f k xv yv
                in g k2 x
in g k1 x
```
- GHC makes extensive use of user-written rewrite rules as optimizing transformations [22]. For example, stream fusion relies on the following rule, which states that turning a stream into a list and back does nothing [6]:
```
{-# RULES "stream/unstream"
    forall s. stream (unstream s) = s #-}
```
  In CPS, these nested function applications are more difficult to spot. Also, rule matching is simply easier to reason about when the rules are written in more-or-less the same syntax as the intermediate language, but CPS makes radical changes compared to the source language.

9. Related Work

Join Points and Commuting Conversions Join points have been around for a long time in practice [27], but they have lacked a formal treatment until now. By introducing join points at the level at which common optimizations are applied, we're able to exploit them more fully. For example, stream fusion as discussed in Sec. 5 depends on several algorithms working in concert, including commuting conversions, inlining, user-specified rewrite rules [22], and call-pattern specialization [21].

Fluet and Weeks [11] describe MLton's intermediate language, whose syntax is much like ours (only first-order). However, it requires that non-tail calls be written so as to pass the result to a named continuation (what we would call a join point). As the authors note, however, this is only a minor syntactic change from passing the continuation as a parameter, and so the language has more in common with CPS than with direct style.

Commuting conversions are also discussed by Benton *et al.* in a call-by-value setting [4]. Consider:

```
let z = let y = case a of { A -> e1; B -> e2 }
        in e3
in e4
```

They show how to apply commuting conversions from the inside outward, creating functions to share code, getting:

```
let z = let j2 y = e3
        in case a of { A -> j2 e1; B -> j2 e2 }
in e4
```

and then:

```
let { j1 z = e4; j2 y = e3 }
in case a of { A -> j1 (j2 e1); B -> j1 (j2 e2) }
```

They call `j1` a "useless function": it is only applied to the result of `j2`. It would be better to combine `j1` with `j2` to save a function call. Their solution is to be careful about the order of commuting conversions, since the problem does not occur if one goes from the outside inward instead. However, with join points, the order does not matter! If we make `j2` a join point, then the second step is instead

```
join j2 y = let z = e3 in e4
in case a of { A -> j2 e1; B -> j2 e2 }
```

which is the same result one gets starting from the outside. So our approach is more robust to the order in which transformations are applied.

SSA The majority of current commercial and open-source compilers (including GCC, LLVM, Mozilla JavaScript) and compiler frameworks use the Static Single Assignment (SSA) form [7], which for an assembly-like language means that variables are assigned only once. If a variable might have different values, it is defined by a ϕ-node, which chooses a value depending on control flow. This makes data flow explicit, which helps to simplify some optimizations.

As it happens, SSA is inter-derivable with CPS [2] or ANF [5]. Code blocks in SSA become mutually-recursive continuations in CPS or functions in ANF, and ϕ-nodes indicate the parameters at the different call sites. In fact, in ANF, the functions representing blocks are always tail-called, so adding join points to ANF gives a closer correspondence to SSA code—functions correspond to functions and join points correspond to blocks. Indeed the Swift Intermediate Language SIL appears to have adopted the idea of "basic blocks with arguments" instead of ϕ-nodes [14].

Sequent Calculus Our previous work [8] showed how to define an intermediate language, called Sequent Core, which sits in between direct style and CPS. Sequent Core disentangles the concepts of "context" and "evaluation order"—contexts are invaluable, but Haskell has no fixed evaluation order, a fact which GHC exploits ruthlessly. The inspiration for our language's design came from logic, namely the *sequent calculus*. The sequent calculus is the twin brother of *natural deduction*, the foundation of direct-style representations. In this paper, we use Sequent Core as our inspiration much as Flanagan *et al.* [10] used CPS, with a new motto: *Think in sequent calculus; work in λ-calculus.*

Relation to a Language with Control Since F_J has a notion of control, it becomes natural to relate it to known control theories such as the one developed to reason about **callcc** in Scheme [9]. In fact, our language can encode $\mathbf{callcc}\, v$ as $\mathbf{join}\, j\, x = x\, \mathbf{in}\, [\![v]\!]\, (\lambda y.\, \mathbf{jump}\, j\, y)$. By design, this encoding does not type in our system since the continuation variable j is free in a lambda-abstraction. This has repercussions on the semantics: join points can no longer be saved in the stack but need to be stored in the heap, which is precisely what is needed to implement **callcc**.

10. Reflections

Based on our experience in a mature compiler for a statically-typed functional language, the use of F_J as an intermediate language seems very attractive. Compared to the baseline of System F, F_J is a rather small change; other transformations are barely affected; the new commuting conversions are valuable in practice; and they make the transformation pipeline more robust.

Although we have presented F_J as a lazy language, *everything in this paper applies equally to a call-by-value language*. All one needs to do is to change the evaluation context, the notion of what is substitutable, and a few typing rules (as described in Sec. 6).

Acknowledgments

We gratefully acknowledge helpful feedback from Arjan Boeijink, Joachim Breitner, Gabor Greif, Andrew Kennedy, Oleg Kiselyov, Simon Marlow, and Carter Schonwald. This material is based upon work supported by the National Science Foundation under Grant No. CCF-1423617.

References

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

[2] A. W. Appel. SSA is functional programming. *SIGPLAN Notices*, 33(4):17–20, 1998.

[3] C. A. Baker-Finch, K. Glynn, and S. L. Peyton Jones. Constructed product result analysis for Haskell. *J. Funct. Program.*, 14(2):211–245, 2004.

[4] N. Benton, A. Kennedy, S. Lindley, and C. V. Russo. Shrinking reductions in SML.NET. In *Implementation and Application of Functional Languages, 16th International Workshop, IFL 2004, Lübeck, Germany, September 8-10, 2004, Revised Selected Papers*, pages 142–159, 2004.

[5] M. M. T. Chakravarty, G. Keller, and P. Zadarnowski. A functional perspective on SSA optimisation algorithms. *Electr. Notes Theor. Comput. Sci.*, 82(2):347–361, 2003.

[6] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: from lists to streams to nothing at all. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'07)*, Freiburg, Germany, Oct. 2007. ACM.

[7] 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.

[8] P. Downen, L. Maurer, Z. M. Ariola, and S. Peyton Jones. Sequent calculus as a compiler intermediate language. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 74–88, 2016.

[9] M. Felleisen and R. Hieb. A revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

[10] 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.

[11] M. Fluet and S. Weeks. Contification using dominators. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001.*, pages 2–13, 2001.

[12] A. Gill and G. Hutton. The worker/wrapper transformation. *J. Funct. Program.*, 19(2):227–251, 2009.

[13] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and types*, volume 7. Cambridge University Press Cambridge, 1989.

[14] J. Groff and C. Lattner. Swift intermediate language. LLVM Developers Meeting `http://www.llvm.org/devmtg/2015-10/`, 2015.

[15] A. Keep, A. Hearn, and R. Dybvig. Optimizing closures in O(0) time. In *Annual Workshop on Scheme and Functional Programming*. ACM, 2012.

[16] 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.

[17] S. Lindley. *Normalisation by evaluation in the compilation of typed functional programming languages*. PhD thesis, University of Edinburgh, College of Science and Engineering, School of Informatics, 2005.

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

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

[20] S. Peyton Jones, W. Partain, and A. Santos. Let-floating: moving bindings to give faster programs. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, Philadelphia, May 1996. ACM.

[21] S. L. Peyton Jones. Call-pattern specialisation for Haskell programs. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, pages 327–337, 2007.

[22] S. L. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In R. Hinze, editor, *2001 Haskell Workshop*. ACM, September 2001.

[23] J. H. Reppy. Optimizing nested loops using local CPS conversion. *Higher-Order and Symbolic Computation*, 15(2-3): 161–180, 2002.

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

[25] M. Sulzmann, M. Chakravarty, S. P. Jones, and K. Donnelly. System F with type equality coercions. In *ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'07)*, pages 53–66. ACM, January 2007. ISBN 1-59593-393-X.

[26] J. Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002.*, pages 124–132, 2002.

[27] A. P. Tolmach and D. Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *J. Funct. Program.*, 8(4):367–412, 1998.