Compiling without continuations

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

Abstract

Many fields of study in compilers give rise to the concept of a join point—a place where different execution paths come together. While they have often been treated by representing them as functions or continuations, we believe it is time to study them in their own right. We show that adding them to a direct-style functional intermediate language allows new optimizations to be performed, including a functional version of loop-invariant code motion. Finally, we report on recent work on the Glasgow Haskell Compiler which added join points to the Core language.

1. Introduction

Consider this code, in a functional language:

\[
\text{if } (\text{if } e_1 \text{ then } e_2 \text{ else } e_3) \text{ then } e_4 \text{ else } e_5
\]

Many compilers will perform a commuting conversion \[13\], which naively would produce:

\[
\text{if } e_1 \text{ then } (\text{if } e_2 \text{ then } e_4 \text{ else } e_5) \text{ else } (\text{if } e_3 \text{ then } e_4 \text{ else } e_5)
\]

Commuting conversions are tremendously important in practice (Sec. 2), but there is a problem: the conversion duplicates \(e_4\) and \(e_5\). A natural countermeasure is to name the offending expressions and duplicate the names instead:

\[
\text{let } \{ j_4 () = e_4; j_5 () = e_5 \} \text{ in if } e_1 \text{ then (if } e_2 \text{ then } j_4 () \text{ else } j_5 () \text{) else (if } e_3 \text{ then } j_4 () \text{ else } j_5 () \text{)}
\]

We describe \(j_4\) and \(j_5\) 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 \(j_4\) and \(j_5\). 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 \(j_4\) 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 \(\beta\)-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. 3).

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 \(\lambda\)-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 conification \[16\], but it looks rather different in our setting.
- 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/fold fusion \[22\].
- 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.

Case-of-case transformation Consider these function definitions:

\[
isNothing :: \text{Maybe } a \to \text{Bool}
isNothing x = \text{case } x \text{ of Nothing }\to \text{True}
\]

\[
\text{Just } _\to \text{False}
\]
because you can think of them as places where control joins up

Compiling join points efficiently

Just x

Join point

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

null 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 
[1] -> 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 im-

mediately decompose it. We want to move the outer case into the

branches of the inner one, like this:

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

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 sim-

plified, 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 point

Joining conversions have a problem, though: they often duplicate the outer case. In our example that was OK, but what about

case (case (case v of { p1 -> e1; p2 -> e2 }) of

{ Nothing -> BIG1; Just x -> BIG2 }) of

{ Nothing -> j1 (); Just x -> j2 x }

where BIG1 and BIG2 are big expressions? We do not want to du-

plicate these large expressions, or we would risk bloating the size of

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 parameter.

Compiling join points efficiently

We call j1 and j2 join points because you can think of them as places where control joins up

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

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 analy-

sis 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 with exactly the right

number of arguments. (We leave the exact criteria for Sec.[2] 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].

Like-wise, the idea of generating different (and much more efficient)

code for non-escaping let bindings is well established in many

other compilers [15][23][28] as well as GHC.

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, yielding

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.
- 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 join points to make other 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 type annotations.

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

- A join binding that declares a join point. Each join point has a name, a list of type parameters, a list of value parameters, and a body.
- A jump expression that invokes a join point, passing all indicated arguments as well as an additional type argument (as discussed below).

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

#### Static semantics

The type system for System F$_J$ is given in Fig. 1 where typeof gives the type of a constructor and ctors gives the set of constructors for a datatype.

The typing judgement carries two environments, $\Gamma$ and $\Delta$, with $\Delta$ binding join points. The environment $\Delta$ 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 $\tau$ appears in $\Gamma$, and the symbols in $\Gamma$ are unique. Similarly, every label free in $e$ appears in $\Delta$.

To enforce that jumps are not used as side effects, $\Delta$ is reset in every premise for a subterm whose runtime context is not statically known. For example, consider $\text{join } j \ x = \text{RHS} \ \text{in } f (\text{jump } j \ True \ \text{Int})$. Here the context in which the jump is invoked is not statically known—in a lazy language it depends on how $f$ uses its argument—so it cannot be compiled to “adjust the stack and jump.” So $j$ is not a valid join point. We exclude such terms by resetting $\Delta$ to $\varepsilon$ when typechecking the argument in rule APP.

#### Terms

| $x$ | $\in$ | Term variables |
| $j$ | $\in$ | Label variables |
| $e$, $u$, $v$ | $::=$ | $x \ | \ f \ | \ \lambda a.e \ | \ e \ u$ |
| $\alpha \ e \ | \ e \ \varphi$ | $::=$ | $\Delta \not\vdash e : \tau$ Type polymorphism |
| $K \not\vdash \tau$ | $::=$ | Data construction |
| $\text{case } e \ of \ \alpha \varphi$ | $::=$ | Case analysis |
| $\text{let } v b \ in \ v$ | $::=$ | Let binding |
| $\text{join } j b \ in \ u$ | $::=$ | Join-point binding |
| $\text{jump } j \ \vartheta \ \tau$ | $::=$ | Jump |
| $\text{alt } ::=$ | $K \ x \ \vartheta \ a \ u$ | Case alternative |

#### Value bindings and join-point bindings

- $v b ::= x : \tau = e$ Non-recursive value
- $\text{rec } x : \tau = e$ Recursive values
- $j b ::= j \ \vartheta \ \tau \ = e$ Non-recursive join point
- $\text{rec } j \ \vartheta \ \tau$ Recursive join point

#### Types

| $a, b$ | $\in$ | Type variables |
| $\tau, \sigma, \varphi$ | $::=$ | $\alpha$ Variable |
| $T$ | $::=$ | Datatype |
| $\sigma \rightarrow \tau$ | $::=$ | Function type |
| $\tau \varphi$ | $::=$ | Application |
| $\gamma \alpha, \tau$ | $::=$ | Polymorphic type |

#### Frames, evaluation contexts, and stacks

| $F$ | $::=$ | $\emptyset v$ Applied function |
| $\square \tau$ | $::=$ | Instantiated polymorphism |
| $\text{case } \square \ of \ p \rightarrow \square$ | $::=$ | Case scrutinee |
| $\text{join } j b \ in \ \square$ | $::=$ | Join point |
| $E$ | $::=$ | $\square | F[E]$ Evaluation contexts |
| $s$ | $::=$ | $e | F : s$ Stacks |

#### Tail contexts

| $L$ | $::=$ | Empty unary context |
| $\text{case } e \ of \ p \rightarrow \ L$ | $::=$ | Case branches |
| $\text{let } v b \ in \ L$ | $::=$ | Body of let |
| $\text{join } j b \ \vartheta \ \tau \ = L \ in \ L'$ | $::=$ | Join point, body |
| $\text{rec } j b \ \vartheta \ \tau \ = L \ in \ L'$ | $::=$ | Rec join points, body |

#### Miscellaneous

| $C$ | $\in$ | General single-hole term contexts |
| $\Sigma$ | $::=$ | $\cdot \ | \ T : \sigma$ |
| $e$ | $::=$ | $\langle e ; s ; \Sigma \rangle$ Configuration |

### Figure 1: Syntax of System F$_J$.

Nevertheless, the typing of join points is a little bit more flexible than you might suspect. Consider this expression:

$$\text{join } j \ x = \text{RHS} \ \text{in } \begin{cases} \text{case } e \ of \ A \rightarrow \text{jump } j \ True \ C2C \\ B \rightarrow \text{jump } j \ False \ C2C \end{cases}$$

where $C2C = \text{Char} \rightarrow \text{Char}$. This is certainly well typed. A valid transformation is to move the application to $\lambda \ x'$ into both the body and the right hand side of the join, thus:

$$\begin{cases} \text{case } e \ of \ A \rightarrow \text{jump } j \ True \ C2C \\ B \rightarrow \text{jump } j \ False \ C2C \end{cases}$$

Now we can move the application into the branches:

$$\text{join } j \ x = \text{RHS} \lambda \ x'$$
in case of \( A \to (\text{jump } j \ True \ C2C) \) ‘\( x'’

\[ B \to (\text{jump } j \ False \ C2C) \) ‘\( x'’

\[ C \to (\lambda c.c) \) ‘\( x'’

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 is still well typed, as reflected by the fact that \( \Delta \) is not reset in the function part of an application (rule APP).

The types given to join points themselves deserve some attention. A join point that binds type variables \( \overline{\sigma} \) and value arguments of types \( \overline{\sigma} \) is given the type \( \forall \overline{\sigma}. \overline{\sigma} \to \forall r.r \) (rule JBIND). The return type indicated, namely \( \forall r.r \), is often written \( \perp \), and it indicates a non-returning function: a function which does not actually return can be safely given any return type. This is similar to how Haskell’s error function has type \( \forall a. \text{String} \to a \). We have merely moved the universal quantification to the end for consistency with the join syntax, which does not (and must not) bind this "return-type parameter."

So a join point’s type does not reflect the value of its body, and a jump can have any type whatsoever. 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

\[ \text{join } j = \text{"gotcha!"} \text{ in if } b \text{ then } \text{jump } j \text{ Int else 4} \]

because \( j \) returns a String but the body of the \text{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\( \_2 \) 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,

When we introduce the short axiom (Sec.\ref{sec:short-axiom}), 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.
If we naïvely inline context. Since describing the places where a term may return to its evaluation, the former but not the latter. For example:

We can also derive new axioms succinctly using tail contexts. For example, our commuting conversions as written do 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:

\[
E[L[\overline{x}; \tau]] = L[E[\overline{x}]]
\]

This can be derived from commute by first applying \( j\text{drop} \) and \( j\text{inline} \) 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, if every call to a given function is a tail call, 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 the jump rule will be empty).

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

The non-recursive version, contify, attempts to decompose the body of the let \( \text{i.e.,} \) 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 \( \rho \) 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 contifyrec, 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 to occur if some function \( f \) is polymorphic in its return type \([8]\).

Finding bindings to which contify or contifyrec will apply is not difficult. Our implementation is essentially a free-variable
(\lambda x : \sigma . e) \triangledown = \text{let } x : \sigma = e \text{ in } e \quad (\beta)
(\lambda a . e) \triangledown = e \{ \varphi/a \} \quad (\beta_r)
\text{let } \triangledown v b \triangledown \text{ in } C[v] = \text{let } \triangledown v b \triangledown \text{ in } C[v] \quad \text{if } (x : \sigma = v) \in v b \quad (\text{inline})
\text{let } \triangledown v b \triangledown \text{ in } e = e \quad (\text{drop})
\text{join } \triangledown j b \triangledown \text{ in } L[\triangledown \tau, \text{jump } j \triangledown \tau, e'] = \text{join } \triangledown j b \triangledown \text{ in } L[\triangledown \tau, \text{let } \overline{\sigma} = \varphi \triangledown \text{ in } u \{ \varphi/a \}, e'] \quad \text{if } (j \triangledown \overline{\sigma} \overline{\sigma} = u) \in j b \quad (\text{jinline})
\text{join } \triangledown j b \triangledown \text{ in } e = e \quad (\text{jdrop})
\text{case } K \triangledown \overline{\sigma} \text{ of } \overline{a} \overline{\tau} = \text{let } \overline{x} = \varphi \triangledown \text{ in } e \quad \text{if } (K \overline{\sigma} \overline{\sigma} \rightarrow e) \in \overline{\sigma} \overline{a} \overline{\tau} \quad (\text{case})
E'[\text{case } e \text{ of } K \triangledown \overline{\sigma} \rightarrow u] = \text{case } e \text{ of } K \triangledown \overline{\sigma} \rightarrow E[u] \quad (\text{casefloat})
E'[\text{let } \triangledown v b \triangledown \text{ in } e] = \text{let } \triangledown v b \triangledown \text{ in } E[e] \quad (\text{float})
E'[\text{join } \triangledown j b \triangledown \text{ in } e] = \text{join } E'[j b] \triangledown \text{ in } E[e] \quad (\text{jfloat})
E'[\text{jump } j \triangledown \tau \triangledown \tau'] = \text{jump } j \triangledown \tau \triangledown \tau' \quad (\text{abort})

Figure 4: Common optimizations for System FJ.

e = e'

let \phi = \Lambda \overline{\sigma} . \Lambda \overline{\tau} . \text{in } L[\overline{\sigma}] : \tau = \text{join } \phi \overline{\sigma} \overline{\tau} = u \text{ in } L[\text{tail}_\phi(u)] \quad (\text{contify})
\quad \text{if } \rho(\phi \overline{\sigma} \overline{\tau}) = \text{jump } j \phi \overline{\sigma} \overline{\tau} \tau
\quad \text{and } j \notin \text{fv}(L), u : \tau
let \text{rec } \psi = \Lambda \overline{\sigma} . \Lambda \overline{\tau} . L[\overline{\sigma}] \triangledown \text{ in } L'[\overline{\sigma}] : \tau = \text{join } \text{rec } \psi \overline{\sigma} \overline{\tau} = L[\text{tail}_\psi(u)] \text{ in } L'[\text{tail}_\psi(u)] \quad (\text{contifyrec})
\quad \text{if } \rho(\psi \overline{\sigma} \overline{\tau}) = \text{jump } j \psi \overline{\sigma} \overline{\tau} \tau
\quad \text{and } j \notin \text{fv}(L), f \notin \text{fv}(L'), L[\overline{u}] : \tau
\text{tail}_\psi(f \overline{\sigma} \overline{\tau}) \triangleq e \{ \sigma'/a \} \{ u/x \} \quad \text{if } \rho(f \overline{\sigma} \overline{\tau}) = e \text{ and } \text{dom}(\rho) \cap \text{fv}(\overline{u}) = \emptyset
\text{tail}_\psi(e) \triangleq e \quad \text{if } \text{dom}(\rho) \cap \text{fv}(e) = \emptyset
\text{tail}_\psi(e) \triangleq \text{undefined} \quad \text{otherwise}

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

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 \cite{11} or to Sec. 5 of \cite{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, a convenient point of comparison is the local CPS transformation in Moby \cite{23}, which produces mutually tail-recursive functions to improve code generation in much the same way GHC does. Note that Moby uses a direct-style intermediate representation, though its contification pass is expressed in terms of a CPS transform.

In essence, the final effect of Moby’s local CPS transform is to turn

\begin{verbatim}
let f x = ... in E[... f y ... f z ... ]
\end{verbatim}

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

\begin{verbatim}
let \{ j x = E[x] ; f x = j <rhs> \} in ... f y ... f z ...
\end{verbatim}

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

\begin{verbatim}
let f x = rhs in E[... f y ... f z ... ]
\end{verbatim}

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

\begin{verbatim}
E[let f x = rhs in ... f y ... f z ... ]
\end{verbatim}

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

\begin{verbatim}
E[join f x = rhs in ... jump f y \tau ... jump f z \tau ... ]
\end{verbatim}

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

\begin{verbatim}
join f x = E[rhs] in ... E[jump f y \tau ... E[jump f z \tau ... ]...
\end{verbatim}

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

\begin{verbatim}
join \{ j x = E[x] ; f x = j <rhs> \} in ... f y ... f z ...
\end{verbatim}

Thus we achieve the same result without any extra effort.

Naturally, contification is more routine and convenient in CPS-based compilers \cite{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 \cite{23} does so using labelled expressions), so it is only a matter of convenience, not feasibility.

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.

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).
Recursive join points  Consider this program, which finds the first element of a list that satisfies a predicate \( p \):

\[
\text{find} = \Lambda a. \lambda(p : a \rightarrow \text{Bool})(\text{xs} : [a]).
\]

\[
\text{let go } \text{xs} = \text{case } \text{go } \text{xs} \text{ of } x : \text{xs}' \rightarrow \text{if } p x \text{ then } \text{Just } x \\
\text{else } \text{go } \text{xs}' \\
\text{in go } \text{xs}
\]

Programmers quite often write loops like this, with a local definition for \( go \), perhaps to allow \( \text{find} \) to be inlined at a call site. Our first observation is this: \( go \) is a (recursive) join point! The conftication transformation of will identify a join point, and will transform the \text{let} to a \text{join}, and each call to \text{go} into a \text{jump}. Moreover, the transformed function is much more efficient because there is no longer a heap-allocated closure for \( go \).

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

\[
\text{any} = \Lambda a. \lambda(p : a \rightarrow \text{Bool})(\text{xs} : [a]).
\]

\[
\text{case } \text{find } p x s \text{ of Just } x \rightarrow \text{True} \\
\text{Nothing } \rightarrow \text{False}
\]

The call to \( \text{find} \) can be inlined:

\[
\text{any} = \Lambda a. \lambda(p : a \rightarrow \text{Bool})(\text{xs} : [a]).
\]

\[
\text{join } \text{go } \text{xs} = \text{case } \text{go } \text{xs} \text{ of } x : \text{xs}' \rightarrow \text{if } p x \text{ then } \text{Just } x \\
\text{else } \text{jump } \text{go } \text{xs}' \text{ (Maybe a)} \\
\text{in } \text{jump } \text{go } \text{xs} \text{ (Maybe a)} \text{ of } \text{Just } x \rightarrow \text{True} ; \text{Nothing } \rightarrow \text{False}
\]

Now, we have a call scrutinizing a \text{join} so we can apply axioms \text{f} from Figure 3. After some easy further transformations, we get

\[
\text{any} = \Lambda a. \lambda(p : a \rightarrow \text{Bool})(\text{xs} : [a]).
\]

\[
\text{join } \text{go } \text{xs} = \text{case } \text{go } \text{xs} \text{ of } x : \text{xs}' \rightarrow \text{if } p x \text{ then } \text{True} \\
\text{else } \text{jump } \text{go } \text{xs}' \text{ Bool} \\
\text{in } \text{jump } \text{go } \text{xs} \text{ Bool}
\]

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:

\[
\text{data Stream a where} \\
\text{MkStream } :: s \rightarrow (a \rightarrow \text{Step s a}) \rightarrow \text{Stream a}
\]

There are two competing approaches to the \text{Step} type. In unfold/destroy fusion, first described by Svenningsson [26], we have:

\[
\text{data Step s a } = \text{ Done } | \text{ Yield s a }
\]

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

Now 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:

\[
\text{next } s = \text{ case <incoming step> of } \\
\text{ Yield s'} a \rightarrow \text{<process element>} \\
\text{Done } \rightarrow \text{<process end of stream>}
\]

When composed together and inlined, the stepper functions become a nest of \text{cases}, each scrutinizing the output of the previous stepper. It is crucial for performance that each \text{Yield} or \text{Done} expression be matched to a \text{case}, much as we did with \text{Just} and \text{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.

As, this approach requires a recursive stepper function when implementing \text{filter}, which must loop over incoming elements until it finds a match. This breaks up the chain of \text{cases} by putting a loop in the way, much as our \text{any} above becomes a case on a loop. Hence until now, recursive stepper functions have been unfusible. Coupts et al. [6] suggested adding a Skip constructor to \text{Step}, thus:

\[
\text{data Step s a } = \text{ Done } | \text{ Yield s a } | \text{ Skip s}
\]

Now the stepper function can say to update the state and call again, obviating the need for a loop of its own. This makes \text{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 \text{zip} that consume two lists become more complicated and less efficient.

But with join points, just as with \text{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 \( F_\epsilon \)

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 \( e : e' : \tau \), then either:

1. The initial configuration \( (e; e') \) diverges, or
2. \( (e; e') \Rightarrow^* (A_1; e; \Sigma) \), for some store \( \Sigma \) 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 \( e \equiv e' \), if, given any stack \( s \) and store \( \Sigma \), either

- both \( (e; s; \Sigma) \) and \( (e'; s; \Sigma) \) diverge, or
- for some \( \Sigma_1, A_1, \Sigma_2, A_2, (e; s; \Sigma) \Rightarrow^* (A_1; e; \Sigma_1) \) and \( (e'; s; \Sigma) \Rightarrow^* (A_2; e; \Sigma_2) \).

The equational theory is sound with respect to observational equivalence:

Proposition 3. If \( e \equiv e' \), then \( e \equiv e' \).

Equivalence to System \( F_\epsilon \)  The best way to be sure that \( F_\epsilon \) can be implemented without any headaches is to show that it is equivalent to GHC’s existing System F-based language. This would suggest that the introduction of join points does not allow us to write any new programs, only to implement existing programs more efficiently. To prove the equivalence, we establish an \textit{erasure} procedure that removes all join points from an \( F_\epsilon \) term, leaving an equivalent System F term.

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

\[
\text{join } j = x + 1 \text{ in } \text{jump } j 1 \text{ (Int } \rightarrow \text{Int}) 2
\]

6 Note that \text{Stream} is an existential type, so as to abstract the internal state type \( s \) as an implementation detail of the stream.
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:

```latex
let f = \lambda x.x + 1 \text{ in } f \ 12
```

However, if we apply `abort` first:

```latex
join \ j \ x = x + 1 \ \text{in} \ \text{jump} \ j \ 1 \ \text{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:

```latex
join \ j \ x = x + 1 \ \text{in} \ \begin{cases} \text{case} \ b \ \text{of} \\
\quad \text{True} \rightarrow \text{jump} \ j \ 1 \ (\text{Int} \rightarrow \text{Int}) \\
\quad \text{False} \rightarrow \text{jump} \ j \ 3 \ (\text{Int} \rightarrow \text{Int})
\end{cases}
```

However, this can be handled by a commuting conversion:

```latex
join \ j \ x = x + 1 \ \text{in} \ \begin{cases} \text{case} \ b \ \text{of} \\
\quad \text{True} \rightarrow (\text{jump} \ j \ 1 \ (\text{Int} \rightarrow \text{Int})) \ 2 \\
\quad \text{False} \rightarrow (\text{jump} \ j \ 3 \ (\text{Int} \rightarrow \text{Int})) \ 2
\end{cases}
```

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

```latex
join \ j \ \bar{a} \ \bar{x} = u \ \text{in} \ L[ar{a}],
```

the terms $\bar{x}$ are in tail position for $j$.

The proof of Lemma[3] 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 $\Delta$ into both the scrutinee and the branches; note that case $\varphi \circ \alpha$ of $\delta \circ \alpha$ is an evaluation context and case $\varphi \circ p$ of $\overrightarrow{a}$ is a tail context. But $\varphi \circ \overrightarrow{a}$ is (in call-by-name) neither an evaluation context nor a tail context, and $\text{APP}$ does not propagate $\Delta$ into the argument.

Thus any expression can be written as:

$$L[E(L'[E'[\ldots[L(E[\ldots]])]]],$$

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 $\begin{cases} \text{case} \ b \ \text{of} \\
\quad \text{True} \rightarrow \text{jump} \ j \ 1 \ (\text{Int} \rightarrow \text{Int}) \\
\quad \text{False} \rightarrow \text{jump} \ j \ 3 \ (\text{Int} \rightarrow \text{Int})
\end{cases}$ to say that any expression can be written $L[ar{a}]$, where each $e_i$ is a leaf from the tree in $\begin{cases} \text{case} \ b \ \text{of} \\
\quad \text{True} \rightarrow \text{jump} \ j \ 1 \ (\text{Int} \rightarrow \text{Int}) \\
\quad \text{False} \rightarrow \text{jump} \ j \ 3 \ (\text{Int} \rightarrow \text{Int})
\end{cases}$.

To say a term in the above form is in commuting-normal form, i.e., 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 $F_1$ term $e$, there is a System $F$ term $e'$ such that $e' = e$.

### 7. Join points in practice

Join points are a way to introduce empirical code transformations at compile-time. In this section we report on our experience of doing so in GHC.

**Implementing join points in GHC.** We have implemented System $F_1$ as an extension to the Core language in GHC. Rather than adding two new data constructors for `join` and `jump` to the Core

\footnote{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.}

\footnote{ANF is simply commuting-normal form with named intermediate values.}

data type, we instead re-use ordinary `let-binding`s 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-binding`s. 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 $F_1$ according to the rules of Fig.2.

Second, we added a simple new conflation 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 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 (see 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[10]. 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

```latex
let j \ x y = \ldots \text{ in} \ j \ 1 \ 2,
```

the Float In pass wants to narrow $j$’s scope as much as possible: `(let j \ x y = \ldots \text{ 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-binding`s, 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. 9). So does performance improve? Table[1] presents benchmark data on allocations, collected from the standard spectral, real and shootout
Table 1: Benchmarks from the spectral, real, and shootout NoFib suites.

<table>
<thead>
<tr>
<th>spectral Program</th>
<th>Allocs</th>
<th>real Program</th>
<th>Allocs</th>
</tr>
</thead>
<tbody>
<tr>
<td>fibheaps</td>
<td>-1.1%</td>
<td>anna</td>
<td>+0.5%</td>
</tr>
<tr>
<td>ida</td>
<td>-1.4%</td>
<td>cacheprof</td>
<td>-0.5%</td>
</tr>
<tr>
<td>nucleic2</td>
<td>+0.2%</td>
<td>fem</td>
<td>+3.6%</td>
</tr>
<tr>
<td>para</td>
<td>-4.3%</td>
<td>gamteb</td>
<td>-1.4%</td>
</tr>
<tr>
<td>primetest</td>
<td>-3.6%</td>
<td>hpg</td>
<td>-2.1%</td>
</tr>
<tr>
<td>simple</td>
<td>-0.9%</td>
<td>parser</td>
<td>+1.2%</td>
</tr>
<tr>
<td>solid</td>
<td>-8.4%</td>
<td>rsa</td>
<td>-4.7%</td>
</tr>
<tr>
<td>transform</td>
<td>+1.1%</td>
<td>(18 others)</td>
<td></td>
</tr>
<tr>
<td>(45 others)</td>
<td></td>
<td>Min</td>
<td>-4.7%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Max</td>
<td>+3.6%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Geo. Mean</td>
<td>-0.2%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>shootout Program</th>
<th>Allocs</th>
</tr>
</thead>
<tbody>
<tr>
<td>k-nucleotide</td>
<td>-85.9%</td>
</tr>
<tr>
<td>n-body</td>
<td>-100.0%</td>
</tr>
<tr>
<td>spectral-norm</td>
<td>-0.8%</td>
</tr>
<tr>
<td>(5 others)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Min</td>
</tr>
<tr>
<td></td>
<td>Max</td>
</tr>
<tr>
<td></td>
<td>Geo. Mean</td>
</tr>
</tbody>
</table>

NoFib benchmark suite\(^7\) 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 recognised 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 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 give guaranteed 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 unfolded streams without sacrificing fusion. That knowledge in turn affects the way in which libraries are written: they can be smaller and faster.

\(^7\)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.

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 compiler intermediate language in which to perform optimization\(^1\).

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”\(^2\), 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\(^3\) 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:

\[
\text{letcont}\ k1\ x = \text{letcont}\ k2\ yv = f\ k\ x\ yv \quad \text{in} \quad g\ k2\ x
\]

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; since the point is to write the rules in the source language, this calls for an intermediate language that doesn’t make the same radical changes that CPS makes.

9. Related work

Join points and commuting conversions Join points have been around for a long time in practice\(^4\), 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\(^5\), and call-pattern specialization\(^6\).

\(^1\)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”\(^2\), 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\(^3\) 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:

\[
\text{letcont}\ k1\ x = \text{letcont}\ k2\ yv = f\ k\ x\ yv \quad \text{in} \quad g\ k2\ x
\]

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; since the point is to write the rules in the source language, this calls for an intermediate language that doesn’t make the same radical changes that CPS makes.

9. Related work

Join points and commuting conversions Join points have been around for a long time in practice\(^4\), 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\(^5\), and call-pattern specialization\(^6\).
Fluet and Weeks [11] describe MLton’s intermediate language, whose syntax is much like ours (only first-order). However, it requires that nondtail 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:

\[
\begin{align*}
\text{let } z &= e_4; \quad \text{let } j_2 y = e_3 \\
\text{in } e_4
\end{align*}
\]

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

\[
\begin{align*}
\text{let } j_2 y &= e_3 \\
\text{in } e_4
\end{align*}
\]

and then:

\[
\begin{align*}
\text{let } \{ j_1 z = e_4; \ j_2 y = e_3 \} \\
\text{in } e_4
\end{align*}
\]

They call \( j_1 \) a “useless function”: it is only applied to the result of \( j_2 \). It would be better to combine \( j_1 \) with \( j_2 \) 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 \( j_2 \) a join point, then the second step is instead

\[
\begin{align*}
\text{join } j_2 y &= \text{let } z = e_4; \ j_2 y = e_3 \\
\text{in } e_4
\end{align*}
\]

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.

\section*{10. Reflections}

Based on our experience in a mature compiler for a statically-typed functional language, the use of \( F_3 \) as an intermediate language seems very attractive. Compared to the baseline of System F, \( F_3 \) 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 \( F_3 \) 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. [5]).

\section*{References}


