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.

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

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

1. Introduction

Consider this code, in a functional language:

\[
\text{if (if e1 then e2 else e3) then e4 else e5}
\]

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

\[
\text{if e1 then (if e2 then e4 else e5) else (if e3 then e4 else e5)}
\]

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:

\[
\text{let \{ j4 () = e4; j5 () = e5 \} in if e1 then (if e2 then j4 () else j5 ()) else (if e3 then j4 () else j5 ()}
\]

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 \(\beta\)-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. [5]).

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 contification [10], 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:

\[ \text{isNothing} :: \text{Maybe } a \rightarrow \text{Bool} \]

\[ \text{isNothing} \; x = \text{case } x \; \text{of Nothing} \rightarrow \text{True} \]

\[ \text{Just } \; \_ \rightarrow \text{False} \]

\[ \text{mHead} :: [a] \rightarrow \text{Maybe } a \]

\[ \text{mHead} \; ps = \text{case } ps \; \text{of} \; \text{[]} \rightarrow \text{Nothing} \]

\[ (p:_) \rightarrow \text{Just } p \]

null :: [a] \rightarrow \text{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 \text{[]} \rightarrow \text{Nothing} \quad (p:_) \rightarrow \text{Just } p \) of

\{ Nothing \rightarrow \text{True}; \text{Just } \_ \rightarrow \text{False} \}

Executed directly, this would be terribly inefficient; if the argument list is non-empty we would allocate a result \text{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

\[ \text{[]} \rightarrow \text{case Nothing of Nothing} \rightarrow \text{True} \]

\[ \text{Just } z \rightarrow \text{False} \]

\[ p:_ \rightarrow \text{case } \text{Just } p \; \text{of Nothing} \rightarrow \text{True} \]

\[ \text{Just } \_ \rightarrow \text{False} \]

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 simplified, yielding

null as = case as of \{ \text{[]} \rightarrow \text{True}; \_:_ \rightarrow \text{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

\[ \text{case } \text{case } v \; \text{of} \{ p1 \rightarrow e1; p2 \rightarrow e2 \} \; \text{of} \]

\{ Nothing \rightarrow \text{BIG1}; \text{Just } x \rightarrow \text{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 \text{let} binding:

\[ \text{let} \; \{ j1 () = \text{BIG1}; j2 \; x = \text{BIG2} \} \; \text{in} \]

\[ \text{case } \text{case } v \; \text{of} \{ p1 \rightarrow e1; p2 \rightarrow e2 \} \; \text{of} \]

\{ Nothing \rightarrow j1 (); \text{Just } x \rightarrow j2 \; x \}

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

\[ \text{let} \; \{ j1 () = \text{BIG1}; j2 \; x = \text{BIG2} \} \; \text{in} \]

\[ \text{case } v \; \text{of} \]

\[ p1 \rightarrow \text{case } e1 \; \text{of Nothing} \rightarrow j1 () \]

\[ \text{Just } x \rightarrow j2 \; x \]

\[ p2 \rightarrow \text{case } e2 \; \text{of Nothing} \rightarrow j1 () \]

\[ \text{Just } x \rightarrow j2 \; x \]

Notice that \text{j2} takes as its parameter the variable bound by the pattern \text{Just } x, whereas \text{j1} has no parameter\(^{2}\).

Compiling Join Points Efficiently  We call \text{j1} and \text{j2} join points because you can think of them as places where control joins up again, but so far they are perfectly ordinary \text{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

\(^{2}\) The dummy unit parameter is not necessary in a lazy language, but it is in a call-by-value language.
points. This identifies \texttt{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. 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 \texttt{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 \texttt{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:

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

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

```haskell
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 \texttt{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 \texttt{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 \texttt{case} now scrutinize an uninformative call like \texttt{(j 1)}. So the extra code bloat from duplicating the outer \texttt{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 \texttt{j} is a join point, it could exploit that knowledge to transform our original expression like this:

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

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

- Distinguish certain \texttt{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 $\lambda$-calculus language in the style of System F, with \texttt{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 jumps, join points are polyadic; partial application is not allowed.

3

2017/4/20
Static Semantics  The type system for System $F_J$ is given in Fig. 2 where \texttt{typeof} gives the type of a constructor and \texttt{ctor}s 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 $\texttt{join}$ (rules $\text{JBIND}$ and $\text{RJBIND}$) and consulted at a $\texttt{jump}$. 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 $\Delta$.

For $\texttt{jump}$s 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 $\Delta$ to $e$ in every premise for such a subterm, for example in the premise for the body $e$ in rule $\text{ABS}$.

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 $\vec{\tau}$ is given the type $\forall \vec{a}. \vec{\tau} \rightarrow \forall r. r$ (rule $\text{JBIND}$). Dually, a $\texttt{jump}$ applies a join point to some type arguments (to instantiate $\vec{a}$), some value arguments (to saturate the $\vec{\tau}$), and a final type argument (to instantiate $r$) that specifies the type returned by the $\texttt{jump}$. We put the universal quantification of $r$ at the end to indicate that the argument types $\vec{\tau}$ do not (and must not) mention this “return-type parameter.” Indeed, when we introduce the $\texttt{abort}$ 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 $\text{JBIND}$ 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 $\texttt{join}$ expression. Thus we cannot have

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

because $j$ returns a $\texttt{String}$ but the body of the $\texttt{join}$ returns an $\texttt{Int}$. 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.

\begin{figure}[h]
\centering
\begin{lstlisting}[language=System F]
Terms
\[x \in \text{Term variables}\]
\[j \in \text{Label variables}\]
\[e, u, v ::= x | l | \lambda x. e | e u | \Lambda a. e | e \varphi | K \vec{\tau} \vec{\sigma} | \text{Data construction}\]
\[| \text{case e of } \vec{a} | \text{Case analysis}\]
\[| \text{let vb in u} | \text{Let binding}\]
\[| \text{join } j b \text{ in } u | \text{Join-point binding}\]
\[| \text{jump } j \vec{\tau} \vec{\sigma} \tau \rho | \text{Jump}\]
\[alt ::= K \vec{\tau} \vec{\sigma} \rightarrow u | \text{Case alternative}\]

Value bindings and join-point bindings
\[vb ::= x = e | \text{Non-recursive value}\]
\[| \text{rec } x : \tau = e | \text{Recursive values}\]
\[jb ::= j \vec{a} = e | \text{Non-recursive join point}\]
\[| \text{rec } j \vec{a} = e | \text{Recursive join points}\]

Types
\[a, b \in \text{Type variables}\]
\[\tau, \sigma, \varphi ::= a | \text{Variable}\]
\[| T | \text{Datatype}\]
\[| \sigma \rightarrow \tau | \text{Function type}\]
\[| \varphi \in \sigma | \text{Application}\]
\[| \forall a. \tau | \text{Polymorphic type}\]

Frames, evaluation contexts, and stacks
\[F ::= \Box \nu | \text{Applied function}\]
\[| \Box \tau | \text{Instantiated polymorphism}\]
\[| \text{case } \Box \text{ of } \vec{p} \rightarrow \vec{a} | \text{Case scrutiny}\]
\[| \text{join } j b \text{ in } \Box | \text{Join point}\]
\[E ::= \Box | F[E] | \text{Evaluation contexts}\]
\[s ::= \varepsilon | F : s | \text{Stacks}\]

Tail contexts
\[L ::= \Box | \text{Empty unary context}\]
\[| \text{case } e \text{ of } \vec{p} \rightarrow L | \text{Case branches}\]
\[| \text{let vb in } L | \text{Body of let}\]
\[| \text{join } j \vec{a} = e \vec{\sigma} = L \text{ in } L' | \text{Join point, body}\]
\[| \text{join rec } j \vec{a} = e \vec{\sigma} = L \text{ in } L' | \text{Rec join points, body}\]

Miscellaneous
\[C ::= \Box | \text{General single-hole term contexts}\]
\[\Sigma ::= \Box | \text{Heap}\]
\[e ::= (e; s; \Sigma) | \text{Configuration}\]
\end{lstlisting}
\caption{Syntax of System $F_J$.}
\end{figure}
Managing Δ. The typing of join points is a little bit more flexible than you might suspect. Consider this expression:

\[
\text{join } j \ x = \text{RHS}
\]

\[
\text{in case } v \\text{ of } A \rightarrow \text{jump } j \ \text{True C2C}
\]

\[
B \rightarrow \text{jump } j \ \text{False C2C}
\]

\[
C \rightarrow \lambda \ c . c
\]

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

\[
\text{join } j \ x = \text{RHS } 'x'
\]

\[
\text{in case } v \\text{ of } A \rightarrow \text{jump } j \ \text{True C2C}
\]

\[
B \rightarrow \text{jump } j \ \text{False C2C}
\]

\[
C \rightarrow \lambda \ c . c
\]

Now we can move the application into the branches:

\[
\text{join } j \ x = \text{RHS } 'x'
\]

\[
\text{in case } v \\text{ of } A \rightarrow (\text{jump } j \ \text{True C2C}) 'x'
\]

\[
B \rightarrow (\text{jump } j \ \text{False C2C}) 'x'
\]

\[
C \rightarrow (\lambda \ c . c) 'x'
\]

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 should be well typed. That point is reflected in the typing rules by the fact that \(\Delta\) is not reset in the function part of an application (rule \(\text{APP}\)), or in the scrutinee of a case (rule \(\text{CASE}\)).

Operational Semantics. We give System \(FJ\) an operational semantics (Fig. 3) in the style of an abstract machine. A configuration of the machine is a triple \((e; \ s; \ \Sigma)\) 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 \(\Sigma\) 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 \(\text{push}\) rule. Since we define evaluation contexts by composing frames (hence \(F[E]\) in Fig. 1), the rule has a simple form. Most of the
The operational semantics

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 optimizing compiler to determine how to apply the rules to rewrite code. All the axioms carry the usual implicit skipping restrictions to avoid free-variable capture. We spell out the side conditions, the bv function stands for “bound variables”; for instance, \( \text{bv}(x: \sigma = e) = \{x\} \).

The \( \beta, \beta_r, \) and case axioms are analogous of the similarly-named rules in the operational semantics. Since there is no heap, \( \beta \) 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:

\[
\text{join } j (x : \text{Int}) = x + 1 \text{ in } (j\text{jump } j 2 (\text{Int} \rightarrow \text{Int})) 3
\]

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

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

\[
\text{join } j (x : \text{Int}) = x + 1 \text{ in } (2 + 1) 3
\]

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.

The casefloat, float, jfloat, and jfloatrec axioms perform commuting conversions. The former two are conventional, but \( j\text{float} \) and \( j\text{floatrec} \) exploit the new join-point construct to perform exactly the transformation we needed in Sec. 2,\(^5\) to avoid destroying a join point. The only difference between the two is that \( j\text{floatrec} \) 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:

\[
\text{case } \left( \begin{array}{l}
\text{join } j x = \text{BIG} \\
\text{in case } v \text{ of } A \rightarrow \text{jump } j 1 \text{ Bool} \\
B \rightarrow \text{jump } j 2 \text{ Bool} \\
C \rightarrow \text{True}
\end{array} \right)
\]

\{ True \rightarrow False; False \rightarrow True \}

We can use \( j\text{float} \) 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 \( \text{jump} \); and use case to simplify the \( C \) alternative. The result is just what we want:

\[
\text{join } j x = \text{BIG} \text{ of } \{ \text{True} \rightarrow \text{False}; \text{False} \rightarrow \text{True} \}
\]

in case \( v \text{ of } A \rightarrow \text{jump } j 1 \text{ Bool} \\
B \rightarrow \text{jump } j 2 \text{ Bool} \\
C \rightarrow \text{False}
\]

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

\[
E[L[e]] = E[e] \quad (\text{commute})
\]

This rule commute moves the evaluation context \( E \) into each hole of the tail context \( L \).

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:

\[
E[L[e] : \tau] = \text{join } j x = E[x] \text{ in } L[j\text{jump } j e \tau]
\]

\(^5\)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.
\[
\begin{align*}
(\lambda : x : \sigma : e) \ s & = \ \text{let} \ x : \sigma = \ s \ \text{in} \ e \tag{\beta} \\
(\Lambda : a : e) \ \varphi & = e(\varphi/a) \tag{\beta_s} \\
\text{let} \ v b \ \text{in} \ C[x] & = \ \text{let} \ v b \ \text{in} \ C[v] \tag{inline} \\
\text{let} \ v b \ \text{in} \ e & = \ e \tag{drop} \\
\text{join} \ j b \ \text{in} \ L[\varphi, \ \text{jump} \ j \ \varphi \ \varphi \ \tau, \ e] & = \ \text{join} \ j b \ \text{in} \ L[\varphi, \ \text{let} \ x : \sigma = \ v \ \text{in} \ u(\varphi/a), \ e'] \tag{jinline} \\
\text{join} \ j b \ \text{in} \ e & = \ e \tag{jdrop} \\
\text{case} \ K \ \varphi \ \varphi \ \text{of} \ a l l & = \ \text{let} \ x : \sigma = \ v \ \text{in} \ e \tag{case} \\
E[\text{case} \ e \ \text{of} \ K \ \varphi \ \varphi \rightarrow u] & = \ \text{case} \ e \ \text{of} \ K \ \varphi \ \varphi \rightarrow E[u] \tag{casefloat} \\
E[\text{let} \ v b \ \text{in} \ e] & = \ \text{let} \ v b \ \text{in} E[e] \tag{float} \\
E[\text{join} \ j b \ \varphi \ \varphi \rightarrow u \ \text{in} \ e] & = \ \text{join} \ j b \ \varphi \ \varphi \rightarrow E[u] \ \text{in} \ E[e] \tag{jfloat} \\
E[\text{join} \ \text{rec} \ j b \ \varphi \ \varphi \rightarrow u \ \text{in} \ e] & = \ \text{join} \ \text{rec} \ j b \ \varphi \ \varphi \rightarrow E[u] \ \text{in} \ E[e] \tag{jfloat_{\text{rec}}} \\
E[\text{jump} \ j \ \varphi \ \varphi \rightarrow \tau'] & = \ \text{jump} \ j \ \varphi \ \varphi \rightarrow \tau' \tag{abort}
\end{align*}
\]

Figure 4: Common optimizations for System \( F_j \).

This can be derived from \textit{commute} by first applying \textit{jdrop} and \textit{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 \texttt{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 saturated tail call (i.e. appears only in an \textit{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 \textit{jump} will be empty).

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

The non-recursive version, \textit{contify}, attempts to decompose the body of the \texttt{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 \( \rho \) 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 \textit{contify} axiom not to match.

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

Finding bindings to which \textit{contify} or \textit{contify}_{\text{rec}} will apply is not difficult. Our implementation is essentially a free-variable analysis that also tracks whether each free variable has appeared \textit{only} in the holes of tail contexts. This is much simpler than previous contification algorithms because we \textit{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

\[
\text{let} \ f \ x = \ldots \ \text{in} \ E[\ldots \ f \ y \ldots \ f \ z \ldots]
\]

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

\[
\text{let} \ \{ \ j \ x = E[x]; \ f \ x = j <\text{rhs}> \} \ \text{in} \ \ldots \ f \ y \ldots \ f \ z \ldots
\]

where the tail calls to \( f \) are now compiled as jumps. Note that \( f \) now matches the \textit{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:

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

First, applying \textit{float} from right to left floats \( f \) inward:

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

Next, \textit{contify} applies, since the calls to \( f \) are now tail calls:

\[
E[\text{join} \ f \ x = \text{rhs} \ \text{in} \ \ldots \ \text{jump} \ f \ y \tau \ldots \ \text{jump} \ f \ z \tau \ldots]
\]
And now \texttt{jfloat} pushes $E$ into the join point $f$ and the body:

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

From here, \texttt{abort} removes $E$ from the jumps, and we can abstract $E$ by running \texttt{jdrop} and \texttt{jinline} backward:

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

Thus we achieve the same result without any extra effort\footnote{The parts of this sequence not specifically to do with join points were already implemented before in GHC: The Float In pass applies \texttt{float} in reverse, and the Simplifier regularly creates join points to share evaluation contexts (except that previously they were ordinary \texttt{let} bindings).}

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.

\section{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 \texttt{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. We have also shown that it is rather easy to identify \texttt{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.

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

\begin{align*}
\text{find} & = \Lambda a. \lambda (p : a \rightarrow \text{Bool}) (xs : [a]). \\
\text{let } go & \quad \text{case } x \quad \text{of} \\
\quad x &: x' & \text{if } p x \text{ then } \text{Just } x \quad \text{else } go x' \\
\quad [] & \rightarrow \text{Nothing} \\
\text{in } go & \quad x:0
\end{align*}

Programmers quite often write loops like this, with a local definition for \texttt{go}, perhaps to allow \texttt{find} to be included at a call site. Our first observation is: \texttt{go} is a (recursive) join point!

The contification transformation of will identify \texttt{go} as a join point, and will transform the \texttt{let} (which allocates) to a \texttt{join} (which does not), and each call to \texttt{go} to into an efficient \texttt{jump}.

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

\begin{align*}
\text{any} & = \Lambda a. \lambda (p : a \rightarrow \text{Bool}) (xs : [a]). \\
\text{case } & \quad \text{find } p x s \text{ of } \text{Just } - \rightarrow \text{True} \\
& \text{Nothing } \rightarrow \text{False}
\end{align*}

The call to \texttt{find} can be included:

\begin{align*}
\text{any} & = \Lambda a. \lambda (p : a \rightarrow \text{Bool}) (xs : [a]). \\
\text{case } & \quad \text{go } x s \text{ of } \\
& \quad x : x' \rightarrow \text{if } p x \text{ then } \text{Just } x \\
& \quad \text{else } \text{jump } go x s' (\text{Maybe } a) \\
& \quad \text{in } \text{jump } go x s (\text{Maybe } a) \\
& \text{Just } - \rightarrow \text{True; Nothing } \rightarrow \text{False}
\end{align*}

Now, we have a case scrutinizing a \texttt{join} so we can apply axiom \texttt{jfloat} from Figure\cite{4}. After some easy further transformations, we get

\begin{align*}
\text{any} & = \Lambda a. \lambda (p : a \rightarrow \text{Bool}) (xs : [a]). \\
\text{join } & \quad \text{go } x s \text{ of } \\
\quad x : x' \rightarrow \text{if } p x \text{ then } \text{True} \\
& \quad \text{else } \text{jump } go x s' \text{ Bool} \\
& \quad \text{in } \text{jump } go x s \text{ Bool} \\
& \text{False}
\end{align*}

Look carefully at what has happened here: the consumer (\texttt{any}) of a recursive loop (\texttt{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.

\textbf{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 \textit{state} and a \textit{stepper function}, thus\footnote{The parts of this sequence not specifically to do with join points were already implemented before in GHC: The Float In pass applies \texttt{float} in reverse, and the Simplifier regularly creates join points to share evaluation contexts (except that previously they were ordinary \texttt{let} bindings).}
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 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:

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

When composed together and inlined, the stepper functions filter when implementing \texttt{Just}, 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 \texttt{any} above becomes a \texttt{case} on a loop. Hence until now, recursive stepper functions have been un-fusible. Coutts \textit{et al.} [6] suggested adding a \texttt{Skip} constructor to \texttt{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! Everything gets three cases instead of two, leading to more code and more runtime tests; and functions like \texttt{zip} that consume two lists become more complicated and less efficient.

But with join points, just as with \texttt{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 \texttt{F}_J

Proofs can be found in the extended version of this paper².

**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.

\textbf{Proposition 1} (Type safety). If \(\varepsilon; \varepsilon \vdash e : \tau\), then either:

\(1. \text{The initial configuration } \langle e; \varepsilon; \varepsilon \rangle \text{ diverges, or} \)
\(2. \langle e; \varepsilon; \varepsilon \rangle \vdash^* \langle A; \varepsilon; \Sigma \rangle, \text{ for some store } \Sigma \text{ and answer } A. \)

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

\textbf{Definition 2.} Two terms \(e\) and \(e'\) are observationally equivalent, written \(e \equiv e'\), 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 \(\equiv\):

\textbf{Proposition 3.} If \(e = e'\), then \(e \equiv e'\).

**Equivalence to System \texttt{F}** The best way to be sure that \texttt{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 \textit{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 \(F_J\) term, leaving an equivalent System \texttt{F} term.

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

\[
\text{let } f = \lambda x. x + 1 \text{ in } f 1 2
\]

However, if we apply \texttt{abort} first:

\[
\text{join } j x = x + 1 \text{ in } \text{jump } j 1 \text{ Int}
\]

Now the jump is a tail call, so \texttt{contify} applies.

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

\[
\text{join } j x = x + 1 \text{ in case } b \text{ of}
\]
\[
\begin{cases}
\text{True} & \to \text{jump } j 1 \text{ (Int } \to \text{ Int}) \\
\text{False} & \to \text{jump } j 3 \text{ (Int } \to \text{ Int})
\end{cases}
\]

However, this can be handled by a commuting conversion:

\[
\text{join } j x = x + 1 \text{ in case } b \text{ of}
\]
\[
\begin{cases}
\text{True} & \to (\text{jump } j 1 \text{ (Int } \to \text{ Int}) \text{) } 2 \\
\text{False} & \to (\text{jump } j 3 \text{ (Int } \to \text{ Int}) \text{) } 2
\end{cases}
\]

And now \texttt{abort} applies twice and \(j\) can be de-\texttt{contifyed}.

\textbf{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

\[
\text{join } j \boxed{\varepsilon} \boxed{x} = u \text{ in } L[\varepsilon],
\]
the terms $\overline{e}$ 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 $\square$ of all is an evaluation context and case $e$ of $p \rightarrow \square$ 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 $\Delta$ into the argument.

Thus any expression can be written as:

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

(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 $L[\overline{e}]$, 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 non-trivial, non-binding $E$ and non-trivial $L$, 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 conification analysis to identify let-bindings that can be converted into 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 $j x y = \ldots$ in $j 12$, the Float In pass wants to narrow $j$’s scope as much as possible: (let $j x y = \ldots$ in $j) 12$. 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
We caution that these are highly atypical programs, already compressed to fit into this space. The tests are all micro-benchmarks.

| spectral                       | real
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>Allocs</td>
</tr>
<tr>
<td>fibheaps</td>
<td>-1.1%</td>
</tr>
<tr>
<td>ida</td>
<td>-1.4%</td>
</tr>
<tr>
<td>nucleic2</td>
<td>+0.2%</td>
</tr>
<tr>
<td>para</td>
<td>-4.3%</td>
</tr>
<tr>
<td>primetest</td>
<td>-3.6%</td>
</tr>
<tr>
<td>simple</td>
<td>-0.9%</td>
</tr>
<tr>
<td>solid</td>
<td>-8.4%</td>
</tr>
<tr>
<td>sphere</td>
<td>-3.3%</td>
</tr>
<tr>
<td>transform</td>
<td>+1.1%</td>
</tr>
<tr>
<td>(45 others)</td>
<td></td>
</tr>
<tr>
<td>Geo. Mean</td>
<td>-0.4%</td>
</tr>
</tbody>
</table>

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

and shootout NoFib benchmark suite. 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 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:

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.
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:

\[
\begin{align*}
\text{let } z &= \text{let } y = \text{case } a \text{ of } \{ A \to e_1; B \to e_2 \} \text{ in } e_3 \\
\text{in } e_4
\end{align*}
\]

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

\[
\begin{align*}
\text{let } z &= \text{let } j_2 y = e_3 \text{ in case } a \text{ of } \{ A \to j_2 e_1; B \to j_2 e_2 \} \\
\text{in } e_4
\end{align*}
\]

and then:

\[
\begin{align*}
\text{let } \{ j_1 z = e_4; j_2 y = e_3 \} \text{ in case } a \text{ of } \{ A \to j_1 (j_2 e_1); B \to j_1 (j_2 e_2) \}
\end{align*}
\]

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

\[
\begin{align*}
\text{join } j_2 y &= \text{let } z = e_3 \text{ in } e_4 \\
\text{in case } a \text{ of } \{ A \to j_2 e_1; B \to j_2 e_2 \}
\end{align*}
\]

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.

9. Related Work

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 callcc \( v \) as

\[
\begin{align*}
\text{let } x &= x \in [v] (\lambda y. \text{jump } j y).
\end{align*}
\]

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 valu-
able 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


A. Proof of type safety

We did not give a type system for configurations, so the off-the-shelf proof of progress and preservation is not quite applicable. However, we can adapt easily enough by annotating each configuration with a well-typed term that corresponds to it. Write \( \langle e / s; \Sigma; e \rangle \) (or \( \langle e / c \rangle \)) for an annotated configuration. We will need to track the connection between \( e \) and \( c \), for which we need a few tools. Let \( B \) be a binding context, that is, series of let bindings surrounding a hole. Then write \( \llbracket B \rrbracket \) for the store containing those same bindings (but with recursive groups flattened). Also, \( \llbracket E \rrbracket \) translates the evaluation context \( E \) to a stack (which is of course just another syntax for the same structure). Then let \( \sim \) relate terms to configurations such that

\[
\llbracket B[e] \rrbracket \sim \langle e; \llbracket E \rrbracket; \llbracket B \rrbracket \rangle.
\]

Now, write

\[
\langle e/c \rangle : \tau
\]

when \( e \sim c \) and \( \epsilon \vdash e : \tau \), and write

\[
\langle e/c \rangle \mapsto \langle e'/c' \rangle
\]

if \( c \mapsto c' \).

It will also be convenient to consider the types of binding contexts. Let

\[
\Gamma \vdash B : \Gamma'
\]

denote that \( B \) binds the symbols in \( \Gamma' \).

We need a few utilities before we tackle the proof.

**Proposition 6** (Substitution). 1. If \( \Gamma, x : \sigma; \Delta \vdash e : \tau \) and \( \Gamma; \Delta \vdash v : \sigma \), then \( \Gamma; \Delta \vdash e[v/x] : \tau \).

2. If \( \Gamma, a; \Delta \vdash e : \tau \), then \( \Gamma; \Delta \vdash e[\sigma/a] : \tau[\sigma/a] \).

**Lemma 7.** If \( \Gamma; \Delta \vdash \llbracket B[e] \rrbracket : \tau \) and variables bound by \( B \) aren’t free in \( E \), then \( \Gamma; \Delta \vdash \llbracket B[\tau] \rrbracket \).

**Proof.** By induction on \( E \) and then \( B \). \( \square \)

**Lemma 8.** \( \Gamma \vdash B[e] : \tau \) if and only if there exists \( \Gamma' \) with \( \Gamma, \Gamma' \vdash e : \tau \) and \( \Gamma' \vdash B : \Gamma' \).

**Proof.** By induction on \( B \). \( \square \)

Now we are ready:

**Lemma 9** (Progress and preservation). If \( \langle e/c \rangle : \tau \), then either

1. \( c \equiv \langle A; \epsilon; \Sigma \rangle \), where \( A \) is an answer, or

2. \( \langle e/c \rangle \mapsto \langle e'/c' \rangle \) for some \( e' \) and \( c' \) with \( \langle e'/c' \rangle : \tau \).

**Proof.** Since \( e \sim c \), we have \( e \equiv \llbracket B[e_0] \rrbracket \) and \( c \equiv \langle e_0; \llbracket E \rrbracket; \llbracket B \rrbracket \rangle \) for some \( B \), \( E \), and \( e_0 \). Proceed by case analysis on \( e_0 \).

- For \( e_0 \equiv F[e_1] \), where \( F \) is any frame, the push rule applies. From the term’s perspective, push does nothing, since it only shuffles part of the configuration around, so we can take \( e' \sim e \) and \( e' : \tau \) holds by assumption.

\[
\llbracket B[e_0] \rrbracket \equiv \llbracket B[e_1] \rrbracket \sim \langle F[\llbracket e_1 \rrbracket]; \llbracket E \rrbracket; \llbracket B \rrbracket \rangle
\]

\[
\mapsto \langle e_1; F; \llbracket E \rrbracket; \llbracket B \rrbracket \rangle
\]

\[
\mapsto \llbracket B[e_1] \rrbracket : \tau
\]

- For \( e_0 \equiv \text{let } v b \text{ in } e_1 \), the bind rule applies, and we finish with Lemmas 7 and 8.

\[
\llbracket B[e_0] \rrbracket \equiv \llbracket B[\text{let } v b \text{ in } e_1] \rrbracket
\]

\[
\mapsto \langle \text{let } v b \text{ in } e_1; \llbracket E \rrbracket; \llbracket B \rrbracket \rangle
\]

\[
\mapsto \langle e_1; \llbracket E \rrbracket; \llbracket B \rrbracket; \llbracket v b \rrbracket \rangle
\]

\[
\mapsto \llbracket B[\text{let } v b \text{ in } E[e_1]] \rrbracket : \tau \text{ (by Lemma 7, 8)}
\]

- For \( e_0 \equiv \text{jump } j \varphi \triangleright \tau' \), note that there must be a matching join in \( s \) (provable by induction on \( E \)). In other words,

\[
E \equiv E_1[\text{join } j b \text{ in } E_2[\text{jump } j \varphi \triangleright \tau']] \]

where some \( j \varphi \triangleright \varphi' \varphi \triangleright \tau' \). Then the body \( u \) must have the same type as \( E_2[\text{jump } j \varphi \triangleright \tau'] \), and we finish by Prop. 1.

\[
\llbracket B[e_0] \rrbracket \equiv \llbracket B[E_1[\text{jump } j b \text{ in } E_2[\text{jump } j \varphi \triangleright \tau']]] \quad \text{jump } j \varphi \triangleright \tau';
\]

\[
\mapsto \langle \llbracket E_1 \rrbracket \oplus (\text{join } j \varphi \triangleright \varphi' \varphi \triangleright \tau') ; \llbracket B \rrbracket \rangle
\]

\[
\mapsto \langle \llbracket E_1 \rrbracket \oplus \text{join } j \varphi \triangleright \varphi' \varphi \triangleright \tau' \rangle ; \llbracket B \rrbracket
\]

\[
\mapsto \llbracket B[\text{join } j \varphi \triangleright \varphi' \varphi \triangleright \tau'] \rangle ; \llbracket E_2 \rrbracket
\]

\[
\mapsto \llbracket B[\text{join } j \varphi \triangleright \varphi' \varphi \triangleright \tau'] \rangle ; \llbracket E_1 \rrbracket
\]

\[
\mapsto \llbracket B[\text{join } j \varphi \triangleright \varphi' \varphi \triangleright \tau'] \rangle ; \llbracket E_2 \rrbracket
\]

\[
\mapsto \llbracket B[\text{join } j \varphi \triangleright \varphi' \varphi \triangleright \tau'] \rangle ; \llbracket E_2 \rrbracket
\]

\[
\mapsto \llbracket B[\text{join } j \varphi \triangleright \varphi' \varphi \triangleright \tau'] \rangle ; \llbracket E_2 \rrbracket
\]

\[
\mapsto \llbracket B[\text{join } j \varphi \triangleright \varphi' \varphi \triangleright \tau'] \rangle ; \llbracket E_2 \rrbracket
\]

\[
\mapsto \llbracket B[\text{join } j \varphi \triangleright \varphi' \varphi \triangleright \tau'] \rangle ; \llbracket E_2 \rrbracket
\]

\[
\mapsto \llbracket B[\text{join } j \varphi \triangleright \varphi' \varphi \triangleright \tau'] \rangle ; \llbracket E_2 \rrbracket
\]

\[
\mapsto \llbracket B[\text{join } j \varphi \triangleright \varphi' \varphi \triangleright \tau'] \rangle ; \llbracket E_2 \rrbracket
\]

\[
\mapsto \llbracket B[\text{join } j \varphi \triangleright \varphi' \varphi \triangleright \tau'] \rangle ; \llbracket E_2 \rrbracket
\]

\[
\mapsto \llbracket B[\text{join } j \varphi \triangleright \varphi' \varphi \triangleright \tau'] \rangle ; \llbracket E_2 \rrbracket
\]

\[
\mapsto \llbracket B[\text{join } j \varphi \triangleright \varphi' \varphi \triangleright \tau'] \rangle ; \llbracket E_2 \rrbracket
\]

This development assumes a non-recursive join point; nothing changes for a recursive one.

- For \( e_0 \equiv A \), examine \( E \):

  - If \( E \equiv \square \), we are done (case 1).

- If \( E \equiv E'[\text{join } j b \text{ in } \square] \), then ans applies. The reduct typechecks by a standard strengthening lemma, since no label can appear free in an answer.

\[
\llbracket B[e_0] \rrbracket \equiv \llbracket B[E'[\text{join } j b \text{ in } \square]] \rrbracket ; \rrbracket e_0 \rrbracket
\]

\[
\equiv \llbracket B[E'[\text{join } j b \text{ in } \square]] \rrbracket
\]

\[
\mapsto \langle \llbracket E' \rrbracket ; \llbracket \text{join } j b \text{ in } \square \rrbracket ; \llbracket B \rrbracket \rangle
\]

\[
\mapsto \langle \llbracket E' \rrbracket ; \llbracket \text{join } j b \text{ in } \square \rrbracket ; \llbracket B \rrbracket \rangle
\]

\[
\mapsto \llbracket B[e_0] \rrbracket : \tau
\]

- Otherwise, the outermost frame must be of the correct form according to the type of \( e_0 \), so one of \( \beta, \beta_\tau \), or case applies. In each case we finish with either Lemma 7 or Prop. 5.
• For $e_0 \equiv x$, by Lemma 8 we must have that $B : \Gamma$ and because $x \in \text{dom} \, \Gamma$, bind applies; then the RHS of $x$ has the same type that $x$ typechecked under,

$$B[E[e_0]] \equiv B[E[x]] \sim \langle x; \mathbf{[}[E]; \mathbf{[}[B]\rangle \rangle \rightarrow \langle u; \mathbf{[}[E]; \mathbf{[}[B]\rangle \rangle \sim B[E[u]] : \tau$$

\[ \square \]

**Proof of Thm.** Generalize from the initial configuration to any $\langle e;/c \rangle : \tau$, since clearly $e \sim \langle e; e; e \rangle$ and hence $\langle e;/e; e; e \rangle : \tau$. Proceed by coinduction. By Lemma 9 either $c$ is an answer configuration (proving case 2) or $\langle e;/c \rangle \rightarrow \langle e'//c' \rangle$ where $\langle e'//c' \rangle : \tau$. This may proceed forever, proving case 1, or else eventually there must be an answer.

\[ \square \]

### B. Proof of soundness of equational axioms

We reuse the notation from App. A.

The first order of business is to reconcile the two semantics—the operational semantics relates configurations, but the rewrite rules relate terms. Thus we create an alternative operational semantics that relates terms (see Fig. 5). Note that this operational semantics is not strictly deterministic because there may be several bind redexes within a term. The bind rule can be made deterministic by either favoring the outermost bind redex over all others, or delaying bind reduction until no other rules apply. However, doing so is not necessary for our goal of proving soundness of the equational theory (and in fact it makes the task considerably more difficult without out gain), and we already have a deterministic operational reduction relation based on configurations.

To formulate the correspondence, we divide the rules into external and internal categories. For the original configuration-based semantics, the push and bind rules are internal and the rest are external; for the new one, only bind is internal (there is no push).

Our $\sim$ relation from last section can now be more generally characterized for the looser operational semantics which uses terms of the form $S[e]$ as

$$e \sim c \text{ iff } \langle e; e; e \rangle \rightarrow \tau e.$$  

which effectively divides $S$ into a $B$ and $E$ which are translated as before. From this understanding, $\sim$ is now a bisimulation.

**Proposition 10.** If $e \sim c$, then:

1. If $e \rightarrow^* \rightarrow_e e'$, then $c \rightarrow^* \rightarrow_e e'$ with $e' \sim c'$.
2. If $c \rightarrow^* \rightarrow_e e'$, then $e \rightarrow^* \rightarrow_e e'$ with $e' \sim c'$.

**Proof.** Before we demonstrate the bisimulation, observe that for every $S$, there are (unique) $\Sigma$ and $s$ such that $\langle S[e]; e; e \rangle \rightarrow^* \langle e; s; \Sigma \rangle$, which follows by induction on $S$. Additionally, if $\langle e; e; e \rangle \rightarrow^* \langle e'; s; \Sigma \rangle$ then there is some $S$ such that $e \equiv S[e'] \rightarrow^* \langle \Sigma \[ \mathbf{[}[S]\[ e'] \] \rangle$ (where $\mathbf{[}[\Sigma]$ and $\mathbf{[}[s]$ is the reverse translation from heaps to binding contexts and from stacks to evaluation contexts), which follows by induction on $\rightarrow^*$ and performing bind reductions as necessary.

1. First, we show that if $e \rightarrow \rightarrow_e e'$ then $c \rightarrow \rightarrow_e c'$ with $e' \sim c'$. In particular, there is only the one internal reduction $\text{bind} (S[F:\text{let } v b \text{ in } e]) \rightarrow S[\text{let } v b \text{ in } F[e]]$. Before the bind reduction, we have for some $\Sigma$ and $s$

$$\langle S[F:\text{let } v b \text{ in } e]; \varepsilon; e \rangle \rightarrow \tau \langle F[\text{let } v b \text{ in } e]; s; \Sigma \rangle \rightarrow \tau \langle \text{let } v b \text{ in } e; F; s; \Sigma \rangle \rightarrow \tau \langle e; F; s; \Sigma; v b \rangle$$

whereas after the bind reduction, we have

$$\langle S[\text{let } v b \text{ in } F[e]]; \varepsilon; e \rangle \rightarrow \tau \langle \text{let } v b \text{ in } F[e]; s; \Sigma \rangle \rightarrow \tau \langle F[e]; s; \varepsilon; v b \rangle \rightarrow \tau \langle e; F; s; \Sigma; v b \rangle$$

Either $c \rightarrow^* \langle e; F; s; \Sigma; v b \rangle$—in which case we can take $c' \equiv \langle e; F; s; \Sigma; v b \rangle$—or $\rightarrow^* \langle e; F; s; \Sigma; v b \rangle$—in which case, we can take $c' \equiv c$.

Second, we show that if $e \rightarrow \rightarrow_e e'$ then $c \rightarrow \rightarrow_e c'$ with $e' \sim c'$ by cases on the possible reductions for $e \rightarrow \rightarrow_e e'$. Each case is similar to $\beta (S[(\lambda x:\sigma.e) u] \rightarrow S[\text{let } x:\sigma = u \text{ in } e])$. Before $\beta$ reduction, we have for some $\Sigma$ and $s$

$$\langle S[(\lambda x:\sigma.e) u]; \varepsilon; e \rangle \rightarrow \tau \langle (\lambda x:\sigma.e) u; s; \Sigma \rangle \rightarrow \tau \langle \lambda x:\sigma.e; \varepsilon; u; s; \Sigma \rangle \rightarrow \tau \langle e; s; \Sigma; x:\sigma = u \rangle$$

whereas after the $\beta$ reduction, we have

$$\langle S[\text{let } x:\sigma = u \text{ in } e]; \varepsilon; e \rangle \rightarrow \tau \langle \text{let } x:\sigma = u \text{ in } e; s; \Sigma \rangle \rightarrow \tau \langle e; s; \Sigma; x:\sigma = u \rangle$$

Since $\langle \lambda x:\sigma.e; \varepsilon; u; s; \Sigma \rangle \not\rightarrow^* \lambda x:\sigma.e; \varepsilon; u; s; \Sigma$, we can take $\lambda x:\sigma.e; \varepsilon; u; s; \Sigma$.

Therefore, the result follows by induction on $\rightarrow^*$ using the above two facts.

2. First, we show that if $c \rightarrow^* c'$ then $c \sim c'$. Since $e \sim c$ we know $e \rightarrow^* c \rightarrow^* c'$ by definition, and so $e \sim c'$.

Second, we show that if $c \rightarrow^* c'$ then $c \rightarrow^* c'$ with $c' \sim c'$ by cases on the possible reductions for $c \rightarrow^* c'$. Each case is similar to $\beta ((\lambda x:\sigma.u; \varepsilon; v : s; \Sigma) \rightarrow (u; s; \Sigma; x:\sigma = u))$. Because $e \sim (\lambda x:\sigma.u; \varepsilon; v : s; \Sigma)$ we have $e \rightarrow^* \langle \Sigma [[\mathbf{[}[s]\[ \mathbf{[}[\lambda x:\sigma.u]\[ v \] \] \] \rangle \sim (u; s; \Sigma; x:\sigma = u)$

Therefore, the result follows by induction on $\rightarrow^*$ using the above two facts.

\[ \square \]
We can now restate observational equivalence in terms of standard reductions on terms.

**Proposition 11.** \( e \equiv e' \) if and only if for all \( C \), \( C[e] \) diverges if and only if \( C[e'] \) diverges.

**Proof.** First, we show that given \( e \sim c \), \( e \) diverges if and only if \( e \) diverges. Note that the internal reductions of both operational semantics are strongly normalizing. Therefore, any infinite reduction sequence of \( e \) contains an infinite number of external reductions, and similarly for \( c \). The correspondence of divergence then follows from the bisimulation in Lemma 10.

Now, suppose that \( e \equiv e' \). From the above, if \( C[e] \) diverges according to the term-based operational semantics, then

- \( \langle C[e]; e; e \rangle \) diverges because \( C[e] \sim (C[e]; e; e) \),
- \( \langle C[e']; e; e \rangle \) diverges because \( e \equiv e' \), and
- \( C[e'] \) diverges because \( C[e'] \sim (C[e']; e; e) \).

So for all \( C \), \( C[e] \) diverges if and only if \( C[e'] \) diverges. Going the other way, suppose that for all \( C \), \( C[e] \) diverges if and only if \( C[e'] \) diverges. Similarly, if \( (C[e]; e; e) \) diverges, then

- \( C[e] \) diverges because \( C[e] \sim (C[e]; e; e) \),
- \( C[e'] \) diverges because \( C[e] \) diverges, and
- \( (C[e']; e; e) \) because \( C[e'] \sim (C[e']; e; e) \).

So \( e \equiv e' \).

Now that we have our footing, we demonstrate Theorem 2 via two common properties of reduction relations: *confluence* and *standardization*. Let \( \rightarrow \) be defined by the rules for \( \beta \)-read as-is from left to right and \( \rightarrow \) be the compatible closure of \( \rightarrow \). See Fig. 7 for the precise definitions, and note the presence of two extra rules \(*\text{contifydrop} \text{ and contifydrop}_{\text{rec}} \)* which are variants of \(*\text{contify} \text{ and contify}_{\text{rec}} \)* as well as the \(*\text{letcomm} \)* which commutes let bindings. The two extra contification rules do not extend the reduction theory (i.e., \( \rightarrow \)), the reflexive-transitive closure of \( \rightarrow \)), since they are simulated by \(*\text{contify} \text{ and contify}_{\text{rec}} \) with the help of \(*\text{jdrop} \), however they will be helpful bigger-step-reductions for the purposes of demonstrating standardization.

Confluence can be shown directly using the available powerful methodologies, in particular the *decreasing diagrams* technique for establishing confluence.

**Theorem 12** (Confluence). *If \( e \rightarrow^{*} e_{1} \) and \( e \rightarrow^{*} e_{2} \) then \( e_{1} \rightarrow^{*} e' \) and \( e_{2} \rightarrow^{*} e' \) for some \( e' \).*

**Proof.** Confluence follows by a decreasing diagrams argument using the following ordering among the reduction rules:

\[
\text{float} < \text{case} \text{float} < \text{jfloat} < \beta < \beta_{\text{r}} < \text{case} < \text{inline} < \text{jinlinecomm} < \text{letcomm} < \text{contify} < \text{contify}_{\text{rec}}
\]

Note that we do not need to consider \(*\text{contifydrop} \text{ and contifydrop}_{\text{rec}} \)* since they are simulated by the other reduction rules and thus are subsumed by them in the reflexive-transitive closure.
\begin{align*}
\frac{e \rightarrow e', e \rightarrow e'}{C[e] \rightarrow C[e']}
\end{align*}

\[ e \rightarrow e', e \rightarrow e' \]

\[ e \rightarrow e', e \rightarrow e' \]

... reduction relation. Additionally, we will replace the \textit{jinlinedrop} reduction rule with the following more general \textit{jinlinecomm} rule where \( \overline{E[L]} \) is a composition of many evaluation and tail contexts:

\[ \begin{align*}
\text{let rec } f & = \lambda \overline{\alpha} \cdot \lambda \overline{x} \cdot \overline{L[u]} \in \text{\textit{contifydrop}} \\
& \quad \text{let rec } j \overline{\alpha} \overline{x} & = \overline{E[u]}\in \text{\textit{contifydrop}} \\
& \quad \text{let rec } f & = \lambda \overline{\alpha} \cdot \lambda \overline{x} \cdot \overline{L[u]} \in \text{\textit{contifydrop}}
\end{align*} \]

which does not change the reflexive-transitive reduction relation since \textit{jinlinedrop} is simulated by the \textit{jinlinedrop} rule where \( \overline{E[L]} \) is a composition of many evaluation and tail contexts:

\[ \begin{align*}
\text{let rec } j \overline{\alpha} \overline{x} & = \overline{E[u]}\in \text{\textit{contifydrop}} \\
& \quad \text{let rec } f & = \lambda \overline{\alpha} \cdot \lambda \overline{x} \cdot \overline{L[u]} \in \text{\textit{contifydrop}}
\end{align*} \]

This is decreasing because \textit{float} < \textit{case}.  

- The \textit{case}-\textit{casefloat} critical-pair:

\[ \begin{align*}
E[\textit{case } K_i \overline{\alpha} \overline{u} \text{ of } \overline{K_i \overline{x}} & \rightarrow e_i^*] \rightarrow \textit{case}\ E[\textit{let } \overline{x} \overline{\sigma} = \overline{u} \text{ in } e_i] \\
E[\textit{case } K_i \overline{\alpha} \overline{u} \text{ of } \overline{K_i \overline{x}} & \rightarrow e_i] \rightarrow \textit{case}\ E[\textit{let } \overline{x} \overline{\sigma} = \overline{u} \text{ in } e_i]
\end{align*} \]

...
joins to
\[\text{join } E'[jb] \]
\[\in \overrightarrow{L}[E'[\overrightarrow{E}[e]]], \text{ let } x:x = v \in E'[u(\varphi/a), E'[\overrightarrow{E}[e']]]\]
as follows:
\[E'[\text{jump } j \varphi x \overrightarrow{v} \tau, E'[jb]]\]
\[\rightarrow_{\text{float, casefloat, jfloat}} E'[jb] \]
\[\in \overrightarrow{L}[E'[\overrightarrow{E}[e]]], \text{ let } x:x = v \in E'[u(\varphi/a), E'[\overrightarrow{E}[e']]]\]
\[E'[\text{jump } j \varphi x \overrightarrow{v} \tau, E'[jb]]\]
\[\rightarrow_{\text{jinlinecomm}} \]
\[\in \overrightarrow{L}[E'[\overrightarrow{E}[e]]], \text{ let } x:x = v \in E'[u(\varphi/a), E'[\overrightarrow{E}[e']]]\]
which is decreasing because \(\text{float} < \text{casefloat} < \text{jfloat} < \text{jinlinecomm}\).

• The \text{inline-contify} critical pair:

\[\text{let } f = \Lambda \overrightarrow{a}. \Lambda \overrightarrow{x}. u \in L[\overrightarrow{e}, f \overrightarrow{v} \overrightarrow{v}, e']\]
\[\rightarrow_{\text{inline}} \text{let } f = \Lambda \overrightarrow{a}. \Lambda \overrightarrow{x}. u \in L[\overrightarrow{e}, (\Lambda \overrightarrow{a}. \Lambda \overrightarrow{x}. u) \overrightarrow{v} \overrightarrow{v}, e']\]
\[\text{let } f = \Lambda \overrightarrow{a}. \Lambda \overrightarrow{x}. u \in L[\overrightarrow{e}, \Lambda \overrightarrow{a}. \Lambda \overrightarrow{x}. u \overrightarrow{v} \overrightarrow{v}, e']\]
\[\rightarrow_{\text{contify}} \text{join } j \overrightarrow{d} \overrightarrow{d} = u \]
\[\text{in } L[\text{tail}_p(e), \text{jump } j \overrightarrow{d} \overrightarrow{v} \tau, \text{tail}_p(e')]\]
joins to
\[\text{join } j \overrightarrow{d} \overrightarrow{d} = u \in L[\text{tail}_p(e), \text{let } x = v \in u(\varphi/a), \text{tail}_p(e')]\]
as follows:
\[\text{let } f = \Lambda \overrightarrow{a}. \Lambda \overrightarrow{x}. u \in L[\overrightarrow{e}, (\Lambda \overrightarrow{a}. \Lambda \overrightarrow{x}. u) \overrightarrow{v} \overrightarrow{v}, e']\]
\[\rightarrow_{\beta} \text{let } f = \Lambda \overrightarrow{a}. \Lambda \overrightarrow{x}. u \in L[\overrightarrow{e}, \Lambda \overrightarrow{a}. \Lambda \overrightarrow{x}. u \overrightarrow{v} \overrightarrow{v}, e']\]
\[\text{let } f = \Lambda \overrightarrow{a}. \Lambda \overrightarrow{x}. u \in L[\overrightarrow{e}, \Lambda \overrightarrow{a}. \Lambda \overrightarrow{x}. u \overrightarrow{v} \overrightarrow{v}, e']\]
\[\rightarrow_{\text{contify}} \text{join } j \overrightarrow{d} \overrightarrow{d} = u \]
\[\text{in } L[\text{tail}_p(e), \text{let } x = v \in u(\varphi/a), \text{tail}_p(e')]\]
\[\text{join } j \overrightarrow{d} \overrightarrow{d} = u \in L[\text{tail}_p(e), \text{jump } j \overrightarrow{d} \overrightarrow{v} \tau, \text{tail}_p(e')]\]
\[\rightarrow_{\text{jinlinecomm}} \text{join } j \overrightarrow{d} \overrightarrow{d} = u \]
\[\text{in } L[\text{tail}_p(e), \text{let } x = v \in u(\varphi/a), \text{tail}_p(e')]\]
which is decreasing because \(\text{float} < \beta < \beta_\tau < \text{inline} \text{ and jinlinecomm} < \text{contify}\).

• The first \text{letcomm-contify} critical pair:

\[\text{let } vb \text{ in let } f = \Lambda \overrightarrow{a}. \Lambda \overrightarrow{x}. u \in L[\overrightarrow{e}]\]
\[\rightarrow_{\text{letcomm}} \text{let } f = \Lambda \overrightarrow{a}. \Lambda \overrightarrow{x}. u \text{ in let } vb \text{ in } L[\overrightarrow{e}]\]
\[\text{let } vb \text{ in let } f = \Lambda \overrightarrow{a}. \Lambda \overrightarrow{x}. u \in L[\overrightarrow{e}]\]
\[\rightarrow_{\text{contify}} \text{let } vb \text{ in join } j \overrightarrow{d} \overrightarrow{d} = u \in L[\text{tail}_p(e)]\]
joins to \(\text{let } vb \text{ in join } j \overrightarrow{d} \overrightarrow{d} = u \in L[\text{tail}_p(e)]\) as follows:
\[\text{let } f = \Lambda \overrightarrow{a}. \Lambda \overrightarrow{x}. u \in L[\overrightarrow{e}]\]
\[\rightarrow_{\text{contify}} \text{let } vb \text{ in join } j \overrightarrow{d} \overrightarrow{d} = u \text{ in let } vb \text{ in } L[\text{tail}_p(e)]\]
\[\rightarrow_{\text{float}} \text{let } vb \text{ in join } j \overrightarrow{d} \overrightarrow{d} = u \in L[\text{tail}_p(e)]\]
which is decreasing because \(\text{float} < \text{letcomm} < \text{contify}\).

The second \text{letcomm-contify} critical pair:
\[\text{let } f = \Lambda \overrightarrow{a}. \Lambda \overrightarrow{x}. u \in L[\overrightarrow{e}]\]
\[\rightarrow_{\text{letcomm}} \text{let } vb \text{ in let } f = \Lambda \overrightarrow{a}. \Lambda \overrightarrow{x}. u \in L[\overrightarrow{e}]\]
\[\text{let } f = \Lambda \overrightarrow{a}. \Lambda \overrightarrow{x}. u \in L[\overrightarrow{e}]\]
\[\rightarrow_{\text{contify}} \text{let } vb \text{ in join } j \overrightarrow{d} \overrightarrow{d} = u \in L[\text{tail}_p(e)]\]
joins to \(\text{let } vb \text{ in join } j \overrightarrow{d} \overrightarrow{d} = u \in L[\text{tail}_p(e)]\) as follows:
\[\text{let } f = \Lambda \overrightarrow{a}. \Lambda \overrightarrow{x}. u \in L[\overrightarrow{e}]\]
\[\rightarrow_{\text{contify}} \text{let } vb \text{ in join } j \overrightarrow{d} \overrightarrow{d} = u \in L[\text{tail}_p(e)]\]
\[\text{let } vb \text{ in join } j \overrightarrow{d} \overrightarrow{d} = u \in L[\text{tail}_p(e)]\]
\[\rightarrow_{\text{float}} \text{let } vb \text{ in join } j \overrightarrow{d} \overrightarrow{d} = u \in L[\text{tail}_p(e)]\]
which is decreasing because \(\text{float} < \text{contify}\).

• The \text{letcomm-contifyrec} critical pairs join similarly to the previous pair, which is also decreasing because \(\text{float} < \text{letcomm} < \text{contify}\).

Standardization—which states that for any result (i.e., a term without a standard redex written \(e \not\rightarrow\)) reachable by the general \(\rightarrow\), there is a similar expanded result that is reachable by the standard \(\Rightarrow\) relation—is harder. We will take the approach of postponing all non-standard reductions after standard ones. Let \(\leftrightarrow_B\) be defined as any \(\rightarrow\) that is not standard, \(\Rightarrow\) be any \(\Rightarrow\) that is not standard, and \(\Rightarrow_B\) be any \(\leftrightarrow\) within a \(B\) context, as shown in \(\boxtimes\) Sorting a reduction sequence to put the standard reductions first works as a standardization technique since non-standard reduction only relates results to other results.

Lemma 13. 1. If \(e_1 \rightarrow e_2\) and \(e_1 \not\rightarrow\) then \(e_2 \not\rightarrow_B e_2 \not\rightarrow\).
2. If \(e_1 \leftrightarrow_B e_2\) and \(e_1 \not\rightarrow\) then \(e_2 \not\rightarrow\), In other words, if \(e_1 \not\rightarrow\) and \(e_2 \not\rightarrow\), then \(e_1 \not\rightarrow\).
Proof. • The only (non-standard) reduction that can create a standard redex are the contify family, which have the ability to create a bind redex by surrounding a let binding with a join binding. However, in this case any let bindings within the newly-created join point can be floated out with the bind reduction.
• The first fact holds because non-standard reductions cannot destroy standard redexes. The second fact follows from the first, since it is impossible to have a term \( e_1 \mapsto e'_2 \), such that \( e_1 \mapsto e_2 \).

We now seek to postpone non-standard reductions after standard ones. In general, this task is rather difficult to approach directly due to the possibility of duplicating reductions (which defeats attempts to generalize the single-step process to multiple steps), so first we only consider the subset of non-standard reductions which occur within the eye of a chain of let bindings, as described by the \( \mapsto_B \) reduction relation. These specific non-standard reductions can be postponed without duplicating the non-standard reduction, but at the cost of duplication standard reductions and introducing many uses of letcomm (denoted by \( =_{letcomm} \), since it is inherently symmetric).

Lemma 14. If \( e \mapsto_B e_1 \mapsto e' \) then \( e \mapsto e' =_{letcomm} e'_2 \mapsto_B e' \) for some \( e_2 \) and \( e'_2 \).

Proof. By cases on the possible reductions \( e \mapsto_B e_1 \mapsto e' \). When the two reductions are non-overlapping they commute immediately in one step, as

\[
B[\text{let } vb \text{ in } e] \mapsto_B B[e] \mapsto_B B[e'] \\
B[\text{let } vb \text{ in } e] \mapsto_B B[\text{let } vb \text{ in } e'] \mapsto_B B[e']
\]

for example. The other cases where the two reductions overlap in such a way that necessitates a more complex commutation are as follows.

For casefloat followed by case, we have:

\[
B \left[ E[\text{case } K, \varphi \# of K, \xi; \varphi \mapsto e_i'] \right] \\
\mapsto_B B \left[ \text{case } K, \varphi \# of K, \xi; \varphi \mapsto E[e_i'] \right] \\
\quad \mapsto_B B \left[ \text{let } \xi; \varphi = \# in E[e_i] \right] \\
\quad \mapsto_B B \left[ E[\text{case } K, \varphi \# of K, \xi; \varphi \mapsto e_i'] \right] \\
\quad \mapsto_B B \left[ E[\text{let } \xi; \varphi = \# in E[e_i]] \right] \\
\quad \mapsto_B B \left[ \text{let } \xi; \varphi = \# in E[e_i] \right]
\]

For jfloat followed by jump, we have

\[
B \left[ E[\text{join } j \# in E'[\text{jump } j \# \varphi \mapsto \varphi]] \right] \\
\quad \mapsto_B B \left[ \text{join } E[jb] \text{ in } E'[E'[\text{jump } j \# \varphi \mapsto \varphi]] \right] \\
\quad \mapsto_B B \left[ \text{let } \xi; \varphi = \# in \text{join } E[jb] \text{ in } E[e[\varphi/a]] \right]
\]

For contify followed by bind, we have:

\[
B \left[ \text{let } f = \Lambda x. \lambda \varphi. u \text{ in let } vb \text{ in } L[\varphi] \right] \\
\quad \mapsto_B B \left[ \text{let } j \# \varphi = u \text{ in let } vb \text{ in } L[\text{tail}_p(e)] \right] \\
\quad \quad \text{where } \rho(f \varphi) = \text{jump } j \# \varphi \tau \\
\quad \quad \mapsto_B B \left[ \text{let } vb \text{ in } \text{join } j \# \varphi = u \text{ in } L[\text{tail}_p(e)] \right]
\]

For contify followed by jump, we have:

\[
B \left[ \text{let } f = \Lambda x. \lambda \varphi. u \text{ in } L[\varphi, \varphi \mapsto \varphi', \varphi'] : \tau \right] \\
\quad \mapsto_B B \left[ \text{join } \text{jump } j \# \varphi = u \text{ in } \text{let } vb \text{ in } L[\text{tail}_p(e)] \right] \\
\quad \quad \text{where } \rho(f \varphi) = \text{jump } j \# \varphi \tau \\
\quad \quad \quad \mapsto_B B \left[ \text{let } j \# \varphi = u \text{ in } L[\text{tail}_p(e)] \right]
\]

For contifydroprec followed by bind, we have:

\[
B \left[ \text{let } f = \Lambda x. \lambda \varphi. u \text{ in } L[\varphi] \right] \\
\quad \mapsto_B B \left[ \text{let } f = \Lambda x. \lambda \varphi. u \text{ in } L[\varphi, \varphi] \right] \\
\quad \quad \text{where } \rho(f \varphi) = \text{jump } j \# \varphi \tau \\
\quad \quad \quad \mapsto_B B \left[ \text{let } j \# \varphi = u \text{ in } L[\text{tail}_p(e)] \right]
\]

For contifydroprec followed by jump, we have:

\[
B \left[ \text{let } f = \Lambda x. \lambda \varphi. u \text{ in } L[\varphi, \varphi \mapsto \varphi', \varphi'] : \tau \right] \\
\quad \mapsto_B B \left[ \text{join } \text{jump } j \# \varphi = u \text{ in } L[\text{tail}_p(e)] \right] \\
\quad \quad \text{where } \rho(f \varphi) = \text{jump } j \# \varphi \tau \\
\quad \quad \quad \mapsto_B B \left[ \text{let } j \# \varphi = u \text{ in } L[\text{tail}_p(e)] \right]
\]
The two cases involving \( \text{contify}_{\text{rec}} \) (\( \text{contify}_{\text{rec}} \) followed by \( \text{bind} \), and \( \text{contify}_{\text{rec}} \) followed by \( \text{jump} \)) are special cases of the above two cases involving \( \text{contify}_{\text{drop}} \).

The reliance on \( \text{letcomm} \) does not cause us trouble, however, since it is a simple enough reduction that \( \equiv_{\text{letcomm}} \) can be postponed after standard reduction.

**Lemma 15.** If \( e \equiv_{\text{letcomm}} e_1 \mapsto e' \) then \( e \mapsto e_2 \equiv_{\text{letcomm}} e' \) for some \( e_2 \).

**Proof.** Since \( e \equiv_{\text{letcomm}} e_1 \), then \( e \) and \( e_1 \) are the same up to permuted let bindings. The fact that there is an \( e_2 \) such that \( e \mapsto e_2 \equiv_{\text{letcomm}} e' \) follows by cases on \( e_1 \mapsto e \) and permuting let bindings afterward as necessary.

Both of these postponements can be combined together to form the (composable) postponement of \( \equiv_{\text{letcomm}} \mapsto_{\text{B}} \) after \( \mapsto^* \).

**Lemma 16.** If \( e \equiv_{\text{letcomm}} e_1 \mapsto_{\text{B}} e'_1 \mapsto^* e' \) then \( e \mapsto^* e_2 \equiv_{\text{letcomm}} e'_2 \mapsto_{\text{B}} e' \) for some \( e_2 \) and \( e'_2 \).

**Proof.** First, consider the simpler case when \( e_1 \mapsto^* e' \). If \( e_1 \equiv e'_1 \) then \( e \mapsto e_2 \equiv_{\text{letcomm}} e'_2 \mapsto_{\text{B}} e' \) for some \( e_2 \) by Lemma 15. Otherwise \( e_1 \mapsto_{\text{B}} e'_1 \) so \( e \equiv_{\text{letcomm}} e_1 \mapsto^* e_2 \equiv_{\text{letcomm}} e'_2 \mapsto_{\text{B}} e' \) for some \( e_2 \) and \( e'_2 \) by Lemma 14 and \( e \mapsto e_3 \equiv_{\text{letcomm}} e_2 \equiv_{\text{letcomm}} e'_2 \mapsto_{\text{B}} e' \) for some \( e_3 \) and induction on \( e_1 \mapsto^* e_2 \).

Finally, the more general case when \( e'_1 \mapsto^* e' \) by any number of steps follows by induction on the standard reduction sequence.

Generalizing the above postponement procedure to account for any non-standard reduction is difficult because, in general, a non-standard reduction on the right-hand-side of a binding can be duplicated by a standard reduction which copies and inlines that right-hand-side. To counter this complexity we introduce a specific form of grand reduction (also known as parallel reduction) which allows for many single reduction steps to happen simultaneously. We write \( e \mapsto_{\text{i}} e' \) for the non-standard grand reduction internally within the structure of a term, defined inductively in Fig. 8 as \( e \mapsto_{\text{B}} e' \) for postfixing \( e \mapsto_{\text{i}} e' \) with \( \equiv_{\text{letcomm}} \Rightarrow_{\text{B}} \), and \( e \mapsto_{\text{i}} e' \) for prefixing \( e \mapsto_{\text{i}} e' \) with zero or more standard reductions. In addition, we use \( F \mapsto_i F' \) for the grand internal non-standard reduction within frames, also defined inductively in Fig. 8 as well as \( \mathit{vb} \mapsto \mathit{vb}' \), \( \mathit{jb} \mapsto \mathit{jb}' \), and \( \mathit{alt} \mapsto \mathit{alt}' \) which are just defined pointwise by allowing \( e \mapsto e' \) on their immediate subterms.

The grand reduction relation satisfies some important basic properties: \( \mapsto_{\text{i}} \) is reflexive, compatible, and lies between a single step and multiple steps of \( \mapsto \).

**Lemma 17.** 1. \( e \equiv_{\text{letcomm}} e_1 \mapsto e_2 \equiv_{\text{letcomm}} e_3 \).

2. \( e \equiv e \mapsto_{\text{i}} e' \) implies \( C[e] \mapsto_{\text{i}} C[e'] \).
3. \( e \Rightarrow e' \) implies \( e \Rightarrow e' \) implies \( e \Rightarrow^* e' \).

**Proof.** The first fact follows by induction on \( \Rightarrow \). The second fact follows from the first by induction on \( C \). And the third fact follows from the second (to show \( \Rightarrow \) is included in \( \Rightarrow^* \)) as well as by induction on \( \Rightarrow \) (to show \( \Rightarrow \) is included in \( \Rightarrow^* \)).

We are now ready to demonstrate the postponement of \( \Rightarrow \) after \( \rightarrow \), which relies on the following ability to read \( \Rightarrow \) “backwards.”

**Lemma 18.** 1. If \( e \Rightarrow A' \) then \( e \Rightarrow^* B[A] \) for some \( B \) such that \( A \Rightarrow_1 A' \) and \( B[u] \Rightarrow u' \) for all \( u \Rightarrow u \).

2. If \( e \Rightarrow \) \( \text{let } vb \in e' \) then \( e \Rightarrow^* \) \( B[\text{let } vb \in E[e']] \) for some \( B \) and \( E \) such that \( \text{let } vb \in e \Rightarrow^* \), \( \text{let } vb \in e' \), \( B[u] \Rightarrow u \) for all \( u \Rightarrow u' \), and \( E[u] \Rightarrow^* u' \) for all \( u \Rightarrow^* u' \).

3. If \( e \Rightarrow E'[\text{jump } j \not\Rightarrow u' \tau] \) then \( e \Rightarrow^* B[E[j\text{jump } j \not\Rightarrow u' \tau]] \) for some \( B \) and \( E \) such that \( \text{jump } j \not\Rightarrow u' \tau \Rightarrow^* \text{jump } j \not\Rightarrow u' \tau \), \( B[u] \Rightarrow u' \) for all \( u \Rightarrow u' \), and \( j \not\in \text{lv}(E) \).

4. If \( e \Rightarrow S'[x] \) then \( e \Rightarrow^* S[x] \) for some \( S \) such that \( S[u] \Rightarrow S'[u'] \) for all \( u \Rightarrow^* u' \).

**Proof.** By induction on \( e \Rightarrow \) in each case. The general pattern is that each \( \Rightarrow^* \) expansion can create unrenferenceable \( \text{let } \) and \( \text{join } \) bindings via the nonstandard \( \text{drop } \) and \( \text{jdrop } \) reductions. Note that for cases 3 and 4, the contexts \( E \) and \( S \) may additionally differ from \( E' \) and \( S' \) by use of the nonstandard \( \text{casefloat } \) and \( \text{jfloat } \) reductions, but this cannot bind additional labels in \( E \) nor take \( x \) out of the eye of \( S \).

**Lemma 19.** 1. If \( e \Rightarrow e_1 \Rightarrow e' \) then \( e \Rightarrow^* e_2 \Rightarrow^* e' \) for some \( e_2 \).

2. If \( e \Rightarrow e_1 \Rightarrow e' \) then \( e \Rightarrow^* e_2 \Rightarrow^* e_1 \) for some \( e_2 \).

**Proof.** By mutual induction on \( e \Rightarrow e_1 \) and \( e \Rightarrow e_1 \). The postponement of \( \Rightarrow \) after \( \Rightarrow^* \) follows immediately from the inductive hypothesis, Lemma 16 and an induction on the \( \Rightarrow^* \) standard reduction sequence that comes from Lemma 16. Furthermore, none of \( x, l, \lambda x: \sigma.e, \Lambda a.e, K \not\Rightarrow u \), or \( \text{jump } j \not\Rightarrow u \) \( \tau \) have a standard reduction, so the cases for the corresponding rules of \( \Rightarrow^* \) never happen. The remaining cases are for reductions of let bindings, \( \text{let } vb \in e \Rightarrow^* \), \( \text{let } vb \in e_1 \), and reductions within frames, \( F[e] \Rightarrow^* F_1[e_1] \).

In the case of a let binding, we have

\[
\text{let } vb \in e \Rightarrow^* \text{let } vb \in e_1
\]

followed by one of two possible standard reductions:

- \( \text{let } vb \in e \Rightarrow \text{let } vb \in e' \) because \( e_1 \Rightarrow e' \) then by the inductive hypothesis we have an \( e_2 \) such that \( e \Rightarrow^* e_2 \Rightarrow^* e' \) and therefore

\[
\text{let } vb \in e \Rightarrow^* \text{let } vb \in e_2
\]

- \( \text{let } vb \in S_1[x] \Rightarrow \text{let } vb \in S_1[u_1] \) because \( e_1 \equiv S_1[x] \) and \( x: \sigma = u_1 \in vb_1 \); then by Lemma 18 we have \( e \Rightarrow^* S_2[x] \) such that \( S_2[v] \Rightarrow S_1[v'] \) for all \( v \Rightarrow v' \). Furthermore, because there is a \( x: \sigma = u \in vb \) such that \( u \Rightarrow^* u' \Rightarrow u_1 \) we have \( S_2[u] \Rightarrow^* S_2[u_1] \Rightarrow S_1[u_1] \). Therefore

\[
\text{let } vb \in e \Rightarrow^* \text{let } vb \in S_2[x] \Rightarrow \text{let } vb \in S_2[u]
\]

\[
\Rightarrow^* \text{let } vb \in S_2[u_1]
\]

In the case of a frame, we have

\[
F \Rightarrow^* F_1 \Rightarrow F_1[e_1]
\]

followed by one of the following possible standard reductions:
• $F_1[e_1] \rightarrow F_1[e']$ because $e_1 \rightarrow e'$; then by the inductive hypothesis we have an $e_2$ such that $e \rightarrow e_2 \Rightarrow e'$ and therefore

$$F[e] \rightarrow^* F[e_2] \quad \frac{F \Rightarrow_1 F_1 \quad e_2 \Rightarrow e'}{F[e_2] \Rightarrow F[1'e']}
$$

• $(\lambda x: \sigma.v_1) \ u_1 \rightarrow \text{let } x: \sigma = u_1 \text{ in } v_1$ because $e_1 \equiv (\lambda x: \sigma.v_1)$ and $F_1 \equiv \square u_1$; since $F \Rightarrow_1 \square u_1$ we know that $F \equiv \square u$ such that $u \Rightarrow u_1$. Furthermore, since $e \equiv \lambda x: \sigma.v_1$, we know that $e \rightarrow^* B[\lambda x: \sigma.v_1]$ such that $v \rightarrow^* v_1' \Rightarrow v_1$, and $B[u] \Rightarrow u'$ for all $u \Rightarrow u'$ by Lemma 18 Therefore

$$F[e] \rightarrow^* B[\lambda x: \sigma.v_1] \ u \rightarrow^* B[(\lambda x: \sigma.v_1) \ u] \rightarrow^* B[\text{let } x: \sigma = u \text{ in } v_1]
$$

• $(\Lambda a. e') \varphi \rightarrow v_1 \{\varphi / a\}$ because $e_1 \equiv (\Lambda a. v_1)$ and $F_1 \equiv \square \varphi$; similar to the previous case.

• case $K \triangleright v_1$ of $\textit{alt}_1$ $\rightarrow$ \text{let } $x: \sigma \rightarrow v_1' \text{ in } u_1$ because $e_1 \equiv K \triangleright v_1'$, $F_1 \equiv \square \textit{of } \textit{alt}_1$, and $(K \triangleright, x: \sigma \rightarrow u_1) \in \textit{alt}_1$; since $F \Rightarrow_1 \square \textit{of } \textit{alt}_1$ we know that $F \equiv \square \textit{of } \textit{alt}_1$ such that $alt \rightarrow^* \textit{alt}_1 \Rightarrow \textit{alt}_1$. Furthermore, since $e \Rightarrow v \rightarrow^* B[K \triangleright v_1']$ such that $v \Rightarrow v_1'$, and $B[u] \Rightarrow u'$ for all $u \Rightarrow u'$ by Lemma 18 Therefore

$$F[e] \rightarrow^* \text{case } B[K \triangleright v_1'] \text{ of } \textit{alt}_1 \Rightarrow B[\text{let } x: \sigma = v \text{ in } u_1]
$$

• $\text{join } j \text{ in } E_1[\text{jump } j \triangleright \overline{v_1} \tau]$ $\rightarrow$ \text{let } $x: \sigma \rightarrow \overline{v_1} \text{ in } j \text{ in } E_1[\varphi / a]$ because $e_1 \equiv E_1[\text{jump } j \triangleright \overline{v_1} \tau]$, $F_1 \equiv \text{join } j \text{ in } E_1[\square]$, and $(\overline{j a} x: \sigma = u_1) \in j \text{ in } E_1$; since $F \Rightarrow_1 \text{join } j \text{ in } E_1[\square]$ we know that $F \equiv \text{join } j \text{ in } E_1[\square]$ such that $j \rightarrow^* j\# \rightarrow^* j\#$. Furthermore, since $e \Rightarrow E_1[\text{jump } j \triangleright \overline{v_1} \tau]$ we know $e \rightarrow^* B_2[E_2[\text{jump } j \triangleright \overline{v_1} \tau]]$ such that $j \not\in \textup{vs } E_2$, $\overline{v} \not\in \overline{v_1}$, and $B_2[u] \Rightarrow u'$ for all $u \Rightarrow u'$ by Lemma 18 Therefore

$$F[e] \rightarrow^* \text{join } j \text{ in } B_2[E_2[\text{jump } j \triangleright \overline{v_1} \tau]] \rightarrow^* B_2[\text{join } j \text{ in } E_2[\text{jump } j \triangleright \overline{v_1} \tau]] \rightarrow^* B_2[\text{let } x: \sigma = \overline{v} \text{ in } j \text{ in } u_1[\varphi / a]] \rightarrow^* B_2[\text{let } x: \sigma = \overline{v} \text{ in } j \text{ in } u_1[\varphi / a]]
$$

From the postponement of a single $\Rightarrow$ after $\rightarrow$, we can easily derive the postponement of the many-step $\Rightarrow^*$ after $\rightarrow^*$.

Lemma 20. If $e \Rightarrow^* e_1 \rightarrow e'$ then $e \rightarrow^* e_2 \Rightarrow^* e'$ for some $e_2$.

Proof. First, note that Lemma 19 can be generalized to the fact that if $e \Rightarrow e_1 \rightarrow e'$ then $e \rightarrow^* e_2 \Rightarrow e'$ for some $e_2$ by induction on $e_1 \Rightarrow e'$.

Now, since $\Rightarrow$ implies $\Rightarrow$ (Lemma 17), we have $e \Rightarrow^* e_1 \rightarrow e'$. By the above generalization of Lemma 19 and induction on $e_1 \Rightarrow e_1$, we have $e \rightarrow^* e_2 \Rightarrow e'$ for some $e_2$. Finally, since $\Rightarrow$ implies $\Rightarrow^*$, we have $e \rightarrow^* e_2 \Rightarrow^* e'$. □
This final many-step postponement is powerful enough to let us sort between standard and non-standard reductions in a general reduction sequence, putting the standard ones first and thus achieving the standardization procedure.

**Theorem 21 (Standardization).** If \( e \rightarrow^* e' \not\rightarrow \) then \( e \rightarrow^* e'' \rightarrow^* e' \) for some \( e' \not\rightarrow \).

**Proof.** Note that every \( \rightarrow \) is either \( \Rightarrow \) or \( \Rightarrow \) by definition. Therefore, we can proceed by induction on the reduction sequence \( e \rightarrow e' \) to show that \( e \rightarrow^* e'' \rightarrow^* e' \) for some \( e'' \):

- If \( e \equiv e' \) then the result is immediate.
- If \( e \Rightarrow e''' \rightarrow^* e' \) then we have \( e \Rightarrow e''' \rightarrow^* e'' \rightarrow^* e' \) for some \( e'' \) by the inductive hypothesis.
- If \( e \Rightarrow e''' \rightarrow^* e' \) then we have \( e \Rightarrow e''' \rightarrow^* e'' \rightarrow^* e' \) for some \( e'' \) by the inductive hypothesis, and so \( e \Rightarrow e''' \rightarrow^* e'' \rightarrow^* e' \) for some \( e''' \) by Lemma [20].

Finally, because results are preserved by \( \Rightarrow \) expansion (Lemma [18]), we have that \( e' \not\rightarrow^* \) implies that \( e'' \not\rightarrow^* \) by induction on \( e'' \rightarrow^* e' \).

With both confluence and standardization at hand, we can now justify the correctness of the equational axioms.

**Theorem 22 (Correctness).** If \( e = e' \) then \( e \equiv e' \).

**Proof.** Let \( C \) be any context, and (without loss of generality) suppose that \( C[e] \) converges to \( C[e] \rightarrow^* e_1 \not\rightarrow \). Since \( e = e' \), we have \( C[e] = C[e'] \) by the compatibility of \( = \). We have that \( C[e] \rightarrow^* e_2 \leftarrow^* C[e'] \) for some \( e_2 \) by the Church-Rosser property (which is known to follow from confluence (Theorem [12]) by induction on \( = \) as the reflexive-transitive closure of \( \leftarrow~ \)). Since \( e_2 \leftarrow^* C[e] \leftarrow^* e_1 \not\rightarrow \), we have that \( e_2 \leftarrow^* e_3 \leftarrow^* e_1 \) by confluence (since \( \leftarrow \) is included in \( \rightarrow \)) for some \( e_3 \), where \( e_3 \leftarrow^* e_4 \not\rightarrow^* \) for some \( e_4 \). Therefore, by standardization (Theorem [21]) we have an \( e_5 \) such that \( C[e'] \rightarrow^* e_5 \not\rightarrow \). Putting it all together, we get \( C[e'] \rightarrow^* e_4 \rightarrow^* e_1 \not\rightarrow \). Therefore, by standardization (Theorem [21]) we have an \( e_5 \) such that \( C[e'] \rightarrow^* e_5 \not\rightarrow \). Summarizing, Theorem [12], Theorem [21], and Lemma [13] together produce the following diagram:

![Diagram](image)

In conclusion, if \( e = e' \) then for all \( C \), \( C[e] \) converges if and only if \( C[e'] \) does, which means that \( C[e] \) diverges if and only if \( C[e'] \) does. By Lemma [11] this means that \( e \equiv e' \).