Commuting Conversions and Join Points for Call-By-Push-Value

ANONYMOUS AUTHOR(S)

Levy's call-by-push-value (CBPV) is a language that subsumes both call-by-name and call-by-value lambda calculi by syntactically distinguishing values from computations and explicitly specifying execution order. This low-level handling of computation suspension and resumption makes CBPV suitable as a compiler intermediate representation (IR), while its substitution evaluation semantics affords compositional reasoning about programs. In particular, $\beta\eta$-equivalences in CBPV have been used to justify compiler optimizations in low-level IRs. However, these equivalences do not validate *commuting conversions*, which are key transformations in compiler passes such as A-normalization. Such transformations syntactically rearrange computations without affecting evaluation order, and can reveal new opportunities for inlining.

In this work, we identify the commuting conversions of CBPV, define a *commuting conversion normal form* (CCNF) for CBPV, present a single-pass transformation into CCNF based on A-normalization, and prove that well-typed, translated programs evaluate to the same result. To avoid the usual code duplication issues that also arise with ANF, we adapt the explicit join point constructs by Maurer et al. [2017]. Our results are all mechanized in Lean 4.

1 Introduction

Call-by-push-value (CBPV) [Levy 2003] is a programming language paradigm that syntactically distinguishes *values* from *computations*. Explicit constructs between values and computations encode evaluation order by marking where and when computations are suspended and resumed; as a result, CBPV subsumes call-by-name (CBN) and call-by-value (CBV) evaluation strategies. Levy also extends CBPV with various effects, isolating them to the computation fragment.

CBPV's distinction of pure values from effectful computations makes it a suitable compiler intermediate representation (IR) because it makes explicit many low-level considerations of compilers such as closures and control flow. Existing work covers CBPV as an IR to compile to, compile from, and perform compiler optimizations on.

- New [2019] draws parallels between how CBPV only binds values to variables with pushing and popping values on the stack in stack-based typed assembly language (STAL) [Morrisett et al. 2002], showing that they have comparable low-level features. He proposes adding instruction operations to CBPV, bringing it even closer to STAL.
- Explicit thunked computations in CBPV make it clear where closures ought to be created, and Sullivan et al. [2023] add abstract closures and incremental closure conversion to CBPV so that optimizations involving closure conversion can be done gradually.
- By binding all intermediate computations, CBPV makes low-level control flow explicit, but still lends to compositional reasoning with its substitution-based evaluation semantics. Garbuzov et al. [2018] therefore argue for using CBPV as a more compositional alternative to control flow graph (CFG) compiler representations by showing a tight equivalence between the two.
- While CFGs don't have a simple equational theory, Rizkallah et al. [2018] develop an equational theory for CBPV based on $\beta\eta$-reductions. They use it to trivially justify compiler optimizations, such as dead branch elimination and constant folding, which would otherwise be more difficult to justify in CFGs directly.

However, $\beta\eta$-equational theories as developed by Rizkallah et al. [2018] and later by Forster et al. [2019] do not validate *commuting conversion* compiler transformations, which syntactically rearrange computations without affecting their evaluation order. Forster et al. list a few commuting conversions in passing, but they aren't comprehensive and aren't framed in the context of

compilation. Such transformations are key to compilation and form part of well-known passes such as A-normalization [Flanagan et al. 1993], which unnest computations so that control flow is syntactically explicit, making code easier to compile further. Moreover, these transformations can expose new opportunities for further program inlining and simplification. A known issue with naïvely applying commuting conversions, in particular in the presence of branching computations, is that code may be duplicated; applying many commuting conversions can then cause exponential code bloat. Maurer et al. [2017] resolve this with explicit join point constructs.

Therefore, we look at a source-to-source transformation that applies commuting conversions to CBPV, adapting its syntax and type system to accommodate explicit join points. Introducing separate constructs rather than using existing ones allows us to restrict them to only where they are needed, which can lead to more efficient, specialized compilation later on. In particular, because closures are represented by thunks, adding join points to CBPV reinforces the intent that join points represent local code blocks and should *not* be compiled to closures.

This work presents a normal form for CBPV with respect to commuting conversions, and a single-pass transformation into this normal form using join points. We prove that this transformation is type-preserving and evaluation-preserving: it does not change the meaning of CBPV programs. These proofs are mechanized in Lean 4 [de Moura et al. 2015], v4.23.0-rc1. The proof development is provided in the supplementary materials; definitions and theorems in this paper are accompanied by their corresponding file in the `CBPV` directory and name in the file within brackets. In the next section, we give an overview of the language, the transformation, and its motivation, leading to the following contributions.

- We identify a subset of CBPV that is normal with respect to all commuting conversions. To this subset, we add join point and jump constructs, and design a type system enforcing non-escaping join points and tail-only jumps. ↪ Section 3
- We define a single-pass transformation from CBPV into our extended normal form, using joins and jumps to avoid duplicating computations. This transformation preserves well-typedness, proven straightforwardly by induction. ↪ Section 4
- We show that the transformation preserves evaluation behaviour: a closed term returning a value with no thunks and its transformation must return the same terminal value. This is proven via a logical equivalence on terms, and requires showing that commuting conversions are logically equivalent. ↪ Section 5

We discuss some other program equivalences and future work in Section 6, and compare with related work in Section 7.

2 Overview

The core ideas of this paper begin with the thesis that CBPV [Levy 2003] is suitable as a compiler IR because it represents control flow explicitly. In particular, it subsumes both CBN and CBV semantics: compiling the same lambda calculus term with the CBN or CBV compilation strategy yields different CBPV terms whose execution mirrors that of the original evaluation strategy.

$$v, w ::= x \mid () \mid \mathsf{inl}\ v \mid \mathsf{inr}\ v \mid \{m\} \qquad \text{(values)}$$

$$m, n ::= v! \mid \lambda x.\ m \mid n\ v \mid \langle m, m \rangle \mid \mathsf{fst}\ n \mid \mathsf{snd}\ n \mid \mathsf{return}\ v \qquad \text{(computations)}$$
$$\mid \mathsf{let}\ x \leftarrow n\ \mathsf{in}\ m \mid \mathsf{case}\ v\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow m; \mathsf{inr}\ y \Rightarrow m\}$$

Fig. 1. Syntax of call-by-push-value values and computations

CBPV syntactically distinguishes values and computations, listed in Figure 1, using explicit thunks to turn suspended computations into values and explicit returns to embed values into computations,

The CBN and CBV translations use thunks in different ways to enforce when computation occurs. Alongside the thunk and return constructs, we include the unit value, value sums, functions, and computation pairs.[1] In examples to follow, we also use boolean conditionals if … then … else … as syntactic sugar for case expressions on unit sums.

As an example of the way CBN and CBV translations differ, consider the lambda calculus term $(\lambda x.\, x)\ ((\lambda y.\, y)\ z)$, which translates to the following two CBPV terms.

CBN $\quad (\lambda x.\, x!)\ \{(\lambda y.\, y!)\ \{z!\}\}$

CBV $\quad \mathsf{let}\ f \leftarrow \mathsf{return}\ \{\lambda x.\, \mathsf{return}\ x\}\ \mathsf{in}\ \mathsf{let}\ a \leftarrow (\mathsf{let}\ g \leftarrow \mathsf{return}\ \{\lambda y.\, \mathsf{return}\ y\}\ \mathsf{in}\ g!\ z)\ \mathsf{in}\ f!\ a$

While both terms evaluate to the same final value z, their evaluation sequences are different. In the CBN translation, function arguments are thunked ($\{\lambda x.\, \mathsf{return}\ x\}$) and passed wholesale, then forced ($f!$) as they are needed. In the CBV translation, the function and the argument are evaluated in order before carrying out the function application, using let bindings to express the explicit sequencing. First f is evaluated, followed by g, then a, before the final application occurs.

Even though translating to CBPV has made control flow explicit, computations may still be arbitrarily nested, while hides some compiler optimizations. We look at commuting conversions next to handle such computations.

2.1 Commuting conversions

Commuting conversions are syntactic transformations that swap nested eliminators while preserving evaluation order. An important benefit of performing commuting conversions is that it exposes inlining opportunities — that is, subexpressions that can be β-reduced to simplify code. For instance, we can commute a let-bound conditional by pushing the outer let expression into the inner conditional to reveal direct bindings of returned values.

$$\begin{aligned} &\mathsf{let}\ b \leftarrow (\mathsf{if}\ v\ \mathsf{then}\ \mathsf{return}\ \mathsf{false}\ \mathsf{else}\ \mathsf{return}\ w)\ \mathsf{in}\ m \\ \Longrightarrow\ &\mathsf{if}\ v\ \mathsf{then}\ (\mathsf{let}\ b \leftarrow \mathsf{return}\ \mathsf{false}\ \mathsf{in}\ m)\ \mathsf{else}\ (\mathsf{let}\ b \leftarrow \mathsf{return}\ w\ \mathsf{in}\ m) \end{aligned} \quad [1]$$

Now we may choose to inline false inside of m in place of b and simplify the then branch. We may also choose *not* to inline w in the else branch if it happens that w is a particularly large value we don't want to duplicate. The inlining optimization wouldn't have been possible without the commutation, and is a well-known technique [Maurer et al. 2017].

Following Maurer et al. [2017], we define commuting conversions as all transformations that push elimination forms inside of tail positions, *i.e.* the bodies of let expressions and the branches of case expressions. Equation 1 pushes let bindings into conditionals; as shown below, we can also push let bindings into other let bindings, as well as function applications into let bindings and conditionals, to expose those inlining opportunities.

$$\mathsf{let}\ x \leftarrow (\mathsf{let}\ y \leftarrow n\ \mathsf{in}\ \mathsf{return}\ v)\ \mathsf{in}\ m \Longrightarrow \mathsf{let}\ y \leftarrow n\ \mathsf{in}\ (\mathsf{let}\ x \leftarrow \mathsf{return}\ v\ \mathsf{in}\ m) \quad [2]$$

$$(\mathsf{let}\ x \leftarrow n\ \mathsf{in}\ \lambda y.\, m)\ v \Longrightarrow \mathsf{let}\ x \leftarrow n\ \mathsf{in}\ (\lambda y.\, m)\ v \quad [3]$$

$$(\mathsf{if}\ w\ \mathsf{then}\ (\lambda x.\, m_1)\ \mathsf{else}\ (\lambda x.\, m_2))\ v \Longrightarrow \mathsf{if}\ w\ \mathsf{then}\ (\lambda x.\, m_1)\ v\ \mathsf{else}\ (\lambda x.\, m_2)\ v \quad [4]$$

Another benefit of commuting conversions is that they unnest expressions, moving multiple steps of computation out of evaluation contexts, which brings them closer to lower-level code. Commuted code is easier to compile because control flow follows the shape of the syntax. In Equation 2, finding the next computation to execute on the left-hand side requires traversing into the x binding, then evaluating the y binding, and finally popping back out to evaluate the body m.

[1] For simplicity, we omit value pairs, which are present in Levy's original CBPV; they are interesting in our setting because the eliminator is pattern matching, but we already have pattern matching on value sums to consider.

In contrast, the right-hand side makes it explicit that we first evaluate the y binding, then the x binding, then the body.

This sequential nature of commuting conversions resembles the A-normalization compiler pass for the lambda calculus into A-normal form (ANF) because commuting conversions are part of the A-reductions that characterize ANF [Flanagan et al. 1993]. A-normalization makes control flow syntactically explicit by binding intermediate computations and by sequentializing computations. Translations from the lambda calculus to CBPV already bind intermediate computations, so what remains is to sequentialize them via commuting conversions. Rather than performing commuting conversions one by one, which examines the whole program each time, we present a single-pass transformation into commuting conversion normal form (CCNF) in Section 4 that resembles the usual single-pass transformation into ANF.

$$n ::= v! \mid \lambda x.\, m \mid n\ v \mid \mathsf{return}\ v \mid \langle m, m \rangle \mid \mathsf{fst}\ n \mid \mathsf{snd}\ n \qquad \text{(tail-free computations)}$$
$$m ::= n \mid \mathsf{let}\ x \leftarrow n\ \mathsf{in}\ m \mid \mathsf{case}\ v\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow m_1; \mathsf{inr}\ y \Rightarrow m_2\} \qquad \text{(configurations)}$$

Fig. 2. Commuting conversion normal form of computations

In ANF, terms are divided into values, computations, and configurations, where computations are a subset of configurations that don't include let expressions and conditionals (or case expressions generally). Figure 2 makes the same distinction for CBPV, where values remain unchanged; to avoid confusion, we also call the latter *configurations*, while we call the former *tail-free computations*, which are both subsets of CBPV *computations*. Because commuting conversions push computations only into tail positions, *new* opportunities for inlining only occur in m positions, so performing those new inlinings won't violate CCNF.

2.2 Join points

Generalizing Equation 1 to case expressions and arbitrary computations, the corresponding commuting conversion pushes let bindings into case branches.

$$\begin{aligned} &\mathsf{let}\ x \leftarrow (\mathsf{case}\ v\ \mathsf{of}\ \{\mathsf{inl}\ y_1 \Rightarrow n_1; \mathsf{inr}\ y_2 \Rightarrow n_2\})\ \mathsf{in}\ m \\ \Longrightarrow\ &\mathsf{case}\ v\ \mathsf{of}\ \{\mathsf{inl}\ y_1 \Rightarrow (\mathsf{let}\ x \leftarrow n_1\ \mathsf{in}\ m); \mathsf{inr}\ y_2 \Rightarrow (\mathsf{let}\ x \leftarrow n_1\ \mathsf{in}\ m)\} \end{aligned} \qquad [5]$$

There is a code optimization issue with Equations 1 and 5: the let body m gets duplicated across the branches. If the size of m is very large, this can cause code bloat, especially if the branches contain further case expressions. The usual solution for the lambda calculus with let expressions is to let-bind a closure containing m to be called at the end of the branch, also known as a *join point*. Similarly, in CBPV, we can bind a thunked function to be forced and applied.

$$\begin{aligned} &\mathsf{let}\ x \leftarrow (\mathsf{case}\ v\ \mathsf{of}\ \{\mathsf{inl}\ y_1 \Rightarrow m_1; \mathsf{inr}\ y_2 \Rightarrow m_2\})\ \mathsf{in}\ m \\ \Longrightarrow\ &\mathsf{let}\ z \leftarrow \mathsf{return}\ \{\lambda x.\, m\}\ \mathsf{in} \\ &\quad \mathsf{case}\ v\ \mathsf{of}\ \{\mathsf{inl}\ y_1 \Rightarrow (\mathsf{let}\ x \leftarrow m_1\ \mathsf{in}\ z!\ x); \mathsf{inr}\ y_2 \Rightarrow (\mathsf{let}\ x \leftarrow m_2\ \mathsf{in}\ z!\ x)\} \end{aligned} \qquad [6]$$

To the next compiler passes that see this code, the thunk is a value that may capture variables and escape its scope, so somewhere along the pipeline the thunk will be converted into a closure and lifted out, and $z!\ x$ will correspond to a function call. However, we know from the commuting conversion that the thunk will never escape its scope, since it's never passed to a function or stored in another thunk; all that we want to do is jump to the m within and pop an argument x. Its purpose is only to join up branches of a computation, and should be compiled to a local code block.

Inspired by Maurer et al. [2017], who tackle the same issue with commuting conversions in System F with case expressions, we add explicit join point and jump constructs to CBPV in Figure 3. They are accompanied by typing rules that restrict where join points may be used, which we cover

$$m ::= \cdots \mid \mathsf{join}\ j\ x = m\ \mathsf{in}\ m \mid \mathsf{jump}\ j\ v \qquad \text{(join points, jumps)}$$

Fig. 3. Extended configurations

in Section 3. In contrast to *op. cit.*, our jumps aren't tail-free computations and may only appear in tail position, which simplifies both the evaluation semantics and our proofs. Section 7 discusses why we can make this restriction.

Coming back to the commuting conversion of Equation 6, we use join points in place of the bound thunk, jumping to them after binding the branches. Let expressions only bind a single variable, so we only need join points and jumps that take a single argument.

$$\begin{aligned} &\mathsf{let}\ x \leftarrow (\mathsf{case}\ v\ \mathsf{of}\ \{\mathsf{inl}\ y_1 \Rightarrow m_1; \mathsf{inr}\ y_2 \Rightarrow m_2\})\ \mathsf{in}\ m \\ \Longrightarrow\ &\mathsf{join}\ j\ x = m\ \mathsf{in} \\ &\quad \mathsf{case}\ v\ \mathsf{of}\ \{\mathsf{inl}\ y_1 \Rightarrow (\mathsf{let}\ x \leftarrow m_1\ \mathsf{in}\ \mathsf{jump}\ j\ x); \\ &\qquad \mathsf{inr}\ y_2 \Rightarrow (\mathsf{let}\ x \leftarrow m_2\ \mathsf{in}\ \mathsf{jump}\ j\ x)\} \end{aligned} \qquad [7]$$

3 CBPV with Join Points

While Section 2 presents the source (plain CBPV) and target (CCNF CBPV with join points) languages with distinct syntactic forms, in our mechanization (and thus our technical presentation here), we use a single unified syntax and treat CC-normalization as a source-to-source translation, showing *a posteriori* in Section 4 that the output of the translation satisfies Figures 2 and 3. In Section 5, we reason about equivalence between a CBPV term and its translation, which requires both sides of the equivalence to belong to the same syntactic category.

$$\begin{aligned} A &::= \top \mid A + A \mid \mathsf{U}\ B && \text{(value types)} \\ B &::= A \to B \mid B\ \&\ B \mid \mathsf{F}\ A && \text{(computation types)} \\ v, w &::= x \mid () \mid \mathsf{inl}\ v \mid \mathsf{inr}\ v \mid \{m\} && \text{(values)} \\ m, n &::= v! \mid \lambda x.\ m \mid n\ v \mid \langle m, m\rangle \mid \mathsf{fst}\ n \mid \mathsf{snd}\ n \mid \mathsf{return}\ v && \text{(computations)} \\ &\quad \mid \mathsf{let}\ x \leftarrow n\ \mathsf{in}\ m \mid \mathsf{case}\ v\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow m; \mathsf{inr}\ y \Rightarrow m\} \\ &\quad \mid \mathsf{join}\ j\ x = m\ \mathsf{in}\ m \mid \mathsf{jump}\ j\ v \end{aligned}$$

Fig. 4. Syntax of value, computations, and their types [`Syntax:ValType,ComType,Val,Com`]

Figure 4 lists the full syntax of values and computations, along with value types and computation types. The new syntactic forms not found in plain CBPV are highlighted in salmon. For clarity, although the mechanization uses de Bruijn indexing and simultaneous substitutions, we present the syntax here in nominal form, with $\boxed{v\{x \mapsto w\}}$, $\boxed{m\{x \mapsto w\}}$ denoting single (capture-avoiding) substitution of x for w in v and m, respectively. The syntax and related definitions in the mechanization are also intrinsically well scoped with respect to jump variables, which we omit here, and freely use jump well-scopedness throughout the proofs.

3.1 Evaluation semantics

The purpose of join points is best explained by what they do; their single-step evaluation rules are listed in Figure 5, alongside rules for the usual CBPV constructs. We write $\boxed{m \rightsquigarrow^* m'}$ for the reflexive, transitive closure of evaluation. The names of the rules involving join points and jumps not found in plain CBPV are highlighted in salmon.

$\boxed{m \rightsquigarrow m}$

$$\{m\}! \rightsquigarrow m \qquad \text{(E-FORCE)}$$
$$(\lambda x.\, m)\ v \rightsquigarrow m\{x \mapsto v\} \qquad \text{(E-APP)}$$
$$\mathsf{fst}\ \langle m_1, m_2\rangle \rightsquigarrow m_1 \qquad \text{(E-FST)}$$
$$\mathsf{snd}\ \langle m_1, m_2\rangle \rightsquigarrow m_2 \qquad \text{(E-SND)}$$
$$\mathsf{let}\ x \leftarrow \mathsf{return}\ v\ \mathsf{in}\ m \rightsquigarrow m\{x \mapsto v\} \qquad \text{(E-RET)}$$
$$\mathsf{case}\ (\mathsf{inl}\ v)\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow m_1; \mathsf{inr}\ y \Rightarrow m_2\} \rightsquigarrow m_1\{x \mapsto v\} \qquad \text{(E-LEFT)}$$
$$\mathsf{case}\ (\mathsf{inr}\ v)\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow m_1; \mathsf{inr}\ y \Rightarrow m_2\} \rightsquigarrow m_2\{y \mapsto v\} \qquad \text{(E-RIGHT)}$$
$$\mathsf{join}\ j\ x = m\ \mathsf{in}\ \mathsf{jump}\ j\ v \rightsquigarrow m\{x \mapsto v\} \qquad \text{(E-JUMP)}$$
$$\mathsf{join}\ j'\ x = m'\ \mathsf{in}\ \mathsf{jump}\ j\ v \rightsquigarrow \mathsf{jump}\ j\ v \qquad \text{(E-SKIP)}$$
$$\mathsf{join}\ j'\ x = m'\ \mathsf{in}\ tm \rightsquigarrow tm \qquad \text{(E-DROP)}$$

$$\frac{m_1 \rightsquigarrow m_2}{E[m_1] \rightsquigarrow E[m_2]}\ \text{E-CONG} \qquad \frac{m_1 \rightsquigarrow m_2}{\mathsf{join}\ j\ x = m\ \mathsf{in}\ m_1 \rightsquigarrow \mathsf{join}\ j\ x = m\ \mathsf{in}\ m_2}\ \text{E-JOIN}$$

$$\textit{where} \quad E ::= \mathsf{let}\ x \leftarrow \square\ \mathsf{in}\ m \mid \square\ v \mid \mathsf{fst}\ \square \mid \mathsf{snd}\ \square \qquad \text{(evaluation contexts)}$$
$$tm ::= \lambda x.\, m \mid \mathsf{return}\ v \mid \langle m, m\rangle \qquad \text{(terminal computations)}$$

Fig. 5. Evaluation rules for computations [`Evaluation:Eval,nf`]

The usual rules E-FORCE through E-RIGHT say that thunks get forced to their inner computations, applied functions substitute in their arguments, the first and second components of a computational pair can be projected out, and case analysis on the left or right injections reduce to the left or right branches, respectively. Rule E-CONG states that evaluation may occur under evaluation contexts E, which are elimination forms with holes in scrutinee position. We exclude join expressions from evaluation contexts because we later use E to define our set of commuting conversions, which don't commute into join expressions.

Join expressions evaluate under a context $\mathsf{join}\ j\ x = m\ \mathsf{in}\ \square$ in rule E-JOIN. The inner evaluation is done when it reaches a jump or a terminal computation tm, which are computation introduction forms. If the body of the join expression is a jump to the bound join point, then the value jumped with is substituted into the join point in rule E-JUMP. If it jumps to a join point further outward, then this inner join point binding is discarded in rule E-SKIP. The binding is also discarded when a terminal is reached in rule E-DROP.

To illustrate, consider the following reduction sequence for three join points. Intuitively, we jump to j_3 which jumps to j_1 which is m_1, so we end up at $m_1\{x \mapsto v\}$; the reductions that apply to get there are first to jump, then to skip, then to jump.

$$\begin{aligned}
&\mathsf{join}\ j_1\ x = m_1\ \mathsf{in}\ \mathsf{join}\ j_2\ x = m_2\ \mathsf{in}\ \mathsf{join}\ j_3\ x = \mathsf{jump}\ j_1\ x\ \mathsf{in}\ \mathsf{jump}\ j_3\ v \\
&\rightsquigarrow \mathsf{join}\ j_1\ x = m_1\ \mathsf{in}\ \mathsf{join}\ j_2\ x = m_2\ \mathsf{in}\ \mathsf{jump}\ j_1\ v && \text{(E-JUMP)} \\
&\rightsquigarrow \mathsf{join}\ j_1\ x = m_1\ \mathsf{in}\ \mathsf{jump}\ j_1\ v && \text{(E-SKIP)} \\
&\rightsquigarrow m_1\{x \mapsto v\} && \text{(E-JUMP)}
\end{aligned}$$

This is in fact the *only* sequence of reductions, because evaluation is deterministic. If we define normalization as evaluation to a terminal,[2] then normalization is deterministic as well, and multi-step evaluations always evaluate to the same terminal.

Lemma 3.1 (Determinism of evaluation). [Evaluation:Eval.det]
If $m \rightsquigarrow m_1$ *and* $m \rightsquigarrow m_2$, *then* $m_1 = m_2$.

Definition 3.2 (Normalization). [Evaluation:Norm]
m normalizes to tm, written as $\boxed{m \Downarrow tm}$, if $m \rightsquigarrow^* tm$.

Corollary 3.3 (Determinism of normalization). [Evaluation:Norm.join]
If $m \Downarrow tm_1$ *and* $m \Downarrow tm_2$, *then* $tm_1 = tm_2$.

Corollary 3.4 (Merging). [Evaluation:Evals.merge] *If* $m \Downarrow tm$ *and* $m \rightsquigarrow^* m'$, *then* $m' \Downarrow tm$.

3.2 Typing rules

By adding join points, we now have two different forms of bindings: value variables x bind values and have value types, while jump variables j bind join points, which are computations of type B that take some value argument of type A. As the bindings have different types, we use two different typing contexts: $\Gamma ::= \cdot \mid \Gamma, x : A$ for value contexts, and $\Delta ::= \cdot \mid \Delta, j : A \nearrow B$ for jump contexts.

Without the jump contexts, rules T-var through T-snd are the usual typing rules for the values and computations of plain CBPV. When we include jump contexts, they are either threaded through rules unchanged, or they are explicitly empty in some premises. In particular, joins represent local blocks of code that are jumped to with the current environment, and aren't closures. The computation inside of a thunk in rule T-thunk needs to be closed with respect to jump variables to prevent jumps from escaping their local scopes.

Furthermore, we restrict jumps to tail positions in contrast to Maurer et al. [2017], who allow jumps in scrutinee positions. They require type polymorphism to assign arbitrary types to jumps so that type safety isn't violated. We propagate the jump context only in premises with the same type so that jumping never changes the type. Although this restricts jumps to only tail positions, it exactly captures what CC-normalization needs: the ability to jump to another computation at the very end of the previous one. As a result, the evaluation semantics are simpler and the metatheory is cleaner. To enforce this restriction, the scrutinee premises of rules T-app, T-let, T-fst, and T-snd require empty jump contexts, while the tail premises of rules T-let and T-case may contain jumps.

Additionally, we don't allow computation constructors rules T-fun and T-pair to contain jumps, which prevents stuck terms such as $\mathsf{fst}\,(\mathsf{join}\, j\, x = m\, \mathsf{in}\, \langle \mathsf{jump}\, j\, v, \mathsf{jump}\, j\, w \rangle)$. While the evaluation rules can be extended so that this computation reduces to $m\{x \mapsto v\}$, doing so again complicates both the evaluation semantics and the metatheory, and join points for commuting conversions don't require such flexibility.

In rule T-join, we extend Δ with a join declaration when checking the body m_2, which may jump to the join point m_1. Both have the same type B because jumping in tail position doesn't change the type, and rule T-jump indicates that jumping to the join point at j with a value of type A indeed has the same B. We can then, for example, jump to a join point in only one branch of a case expression so long as the join point has the same type as the other branch, as in the following derivable judgement, which we typeset with a dashed bar.

$$\frac{\Gamma \vdash v : A_1 + A_2 \qquad \Gamma, x : A_1 \mid \cdot \vdash m_1 : B \qquad \Gamma, y : A_2 \mid \cdot \vdash m_2 : B}{\Gamma \mid \cdot \vdash \mathsf{join}\, j\, x = m_1\, \mathsf{in}\, \mathsf{case}\, v\, \mathsf{of}\, \{\mathsf{inl}\, x \Rightarrow \mathsf{jump}\, j\, x; \mathsf{inr}\, y \Rightarrow m_2\} : B}$$

[2] This is weak normalization, not strong normalization, since subterms are not normalized. We won't consider strong normalization, so we simply call it normalization.

$\boxed{\Gamma \vdash v : A}$ $\boxed{\Gamma \mid \Delta \vdash m : B}$

$$\textsc{T-var}\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{T-unit}\ \frac{}{\Gamma \vdash () : \top} \qquad \textsc{T-left}\ \frac{\Gamma \vdash v : A_1}{\Gamma \vdash \mathsf{inl}\ v : A_1 + A_2} \qquad \textsc{T-right}\ \frac{\Gamma \vdash v : A_2}{\Gamma \vdash \mathsf{inr}\ v : A_1 + A_2} \qquad \textsc{T-thunk}\ \frac{\Gamma \mid \cdot \vdash m : B}{\Gamma \vdash \{m\} : \mathsf{U}\, B}$$

$$\textsc{T-force}\ \frac{\Gamma \vdash v : \mathsf{U}\, B}{\Gamma \mid \Delta \vdash v! : B} \qquad \textsc{T-fun}\ \frac{\Gamma, x : A \mid \cdot \vdash m : B}{\Gamma \mid \Delta \vdash \lambda x.\, m : A \to B} \qquad \textsc{T-app}\ \frac{\Gamma \mid \cdot \vdash n : A \to B \qquad \Gamma \vdash v : A}{\Gamma \mid \Delta \vdash n\ v : B}$$

$$\textsc{T-ret}\ \frac{\Gamma \vdash v : A}{\Gamma \mid \Delta \vdash \mathsf{return}\ v : \mathsf{F}\, A} \qquad \textsc{T-let}\ \frac{\Gamma \mid \cdot \vdash n : \mathsf{F}\, A \qquad \Gamma, x : A \mid \Delta \vdash m : B}{\Gamma \mid \Delta \vdash \mathsf{let}\ x \leftarrow n\ \mathsf{in}\ m : B}$$

$$\textsc{T-case}\ \frac{\Gamma \vdash v : A_1 + A_2 \qquad \Gamma, x : A_1 \mid \Delta \vdash m_1 : B \qquad \Gamma, y : A_2 \mid \Delta \vdash m_2 : B}{\Gamma \mid \Delta \vdash \mathsf{case}\ v\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow m_1; \mathsf{inr}\ y \Rightarrow m_2\} : B}$$

$$\textsc{T-pair}\ \frac{\Gamma \mid \cdot \vdash m_1 : B_1 \qquad \Gamma \mid \cdot \vdash m_2 : B_2}{\Gamma \mid \Delta \vdash \langle m_1, m_2 \rangle : B_1 \,\&\, B_2} \qquad \textsc{T-fst}\ \frac{\Gamma \mid \cdot \vdash n : B_1 \,\&\, B_2}{\Gamma \mid \Delta \vdash \mathsf{fst}\ n : B_1} \qquad \textsc{T-snd}\ \frac{\Gamma \mid \cdot \vdash n : B_1 \,\&\, B_2}{\Gamma \mid \Delta \vdash \mathsf{snd}\ n : B_2}$$

$$\textsc{T-join}\ \frac{\Gamma, x : A \mid \Delta \vdash m_1 : B \qquad \Gamma \mid \Delta, j : A \nearrow B \vdash m_2 : B}{\Gamma \mid \Delta \vdash \mathsf{join}\ j\ x = m_1\ \mathsf{in}\ m_2 : B} \qquad \textsc{T-jump}\ \frac{j : A \nearrow B \in \Delta \qquad \Gamma \vdash v : A}{\Gamma \mid \Delta \vdash \mathsf{jump}\ j\ v : B}$$

Fig. 6. Typing rules for values and computations [Typing:ValWt,ComWt]

The important typing lemmas that we need are weakening lemmas for both value and jump contexts. In the mechanization, they are proven by induction on the typing derivation via renaming lemmas; for now, we ignore issues of renaming de Bruijn indices, as they are standard.

Lemma 3.5 (Weakening (value contexts)). [Typing:wtWeakenVal,wtWeakenCom]
Suppose x is not free in v, m. If $\Gamma_1, \Gamma_2 \vdash v : A$, then $\Gamma_1, x : A', \Gamma_2 \vdash v : A$, and if $\Gamma_1, \Gamma_2 \mid \Delta \vdash m : B$, then $\Gamma_1, x : A, \Gamma_2 \mid \Delta \vdash m : B$.

Lemma 3.6 (Weakening (jump contexts)). [Typing:wtWeakenJ]
Suppose j is not free in m. If $\Gamma \mid \Delta_1, \Delta_2 \vdash m : B$, then $\Gamma \mid \Delta_1, j : A \nearrow B', \Delta_2 \vdash m : B$.

4 Commuting Conversion Normalization

In Section 2, we listed four examples of commuting conversions as Equations (1) to (4). The general formulation commutes evaluation contexts with *tail contexts*, given below, which are elimination forms with holes where computations continue, which in our case are the bodies of let expressions and the branches of case expressions. We can think of computations with a tail as those that have a "next step". As with evaluation contexts, we exclude join point expressions from tail contexts because our source language is plain CBPV with no join points or jumps.

$$L ::= \mathsf{let}\ x \leftarrow n\ \mathsf{in}\ \square \mid \mathsf{case}\ v\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow \square; \mathsf{inr}\ y \Rightarrow \square\} \qquad \text{(tail contexts)}$$

$$
\begin{array}{rll}
v & ::= x \mid () \mid \mathsf{inl}\ v \mid \mathsf{inr}\ v \mid \{m\} & \text{(values)} \\
n & ::= v! \mid \lambda x.\ m \mid n\ v \mid \mathsf{return}\ v \mid \langle m, m \rangle \mid \mathsf{fst}\ n \mid \mathsf{snd}\ n & \text{(tail-free computations)} \\
m & ::= n \mid \mathsf{let}\ x \leftarrow n\ \mathsf{in}\ m \mid \mathsf{case}\ v\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow m_1; \mathsf{inr}\ y \Rightarrow m_2\} & \text{(configurations)} \\
& \quad \mid \mathsf{join}\ j\ x = m_1\ \mathsf{in}\ m_2 \mid \mathsf{jump}\ j\ v &
\end{array}
$$

Fig. 7. Commuting conversion normal form with join points [CCNF:isVal,isCom,isCfg]

The commuting conversions we consider can then be stated as $E[L[m]] \Rightarrow L[E[m]]$. Informally, they unnest code because they move evaluation contexts into the "next step", and they expose inlining opportunities because these evaluation contexts are no longer blocked by tail contexts that still have a "first step" to compute. The shape of commuting conversions inform the shape of the normal form of computations, reproduced in Figure 7, where tail-free computations n are computations that don't contain tail contexts, and configurations m are all computations that don't contain tail contexts in scrutinee positions, *i.e.* where holes appear in evaluation contexts. By inspection, CCNF is indeed normal with respect to commuting conversions because no ms appear in ns, and therefore there must be no more commuting conversions to do.

To transform a plain CBPV program to one in CCNF, we follow Flanagan et al. [1993] and define a compiler using a continuation K, whose forms are given below. As usual, it can be the empty continuation $\Box$, or it can be the let continuation $\mathsf{let}\ x \leftarrow \Box\ \mathsf{in}\ m$: a let expression is compiled by first translating the let-bound expression, then binding its result to x, and finally continuing on with a configuration m.

$$K ::= \Box \mid \mathsf{let}\ x \leftarrow \Box\ \mathsf{in}\ m \mid K :: k \qquad k ::= \Box\ v \mid \mathsf{fst}\ \Box \mid \mathsf{snd}\ \Box \qquad [\mathsf{CCNF:K}]$$

However, we have three more continuation forms corresponding to each of the remaining three evaluation contexts: application, first projection, and second projection. They are needed because in contrast to ANF for the lambda calculus, functions and computation pairs are *computations* and not *values*, so they can't be let-bound and their elimination forms can take arbitrary tail-free computations. The mutual definitions of the translation of values $\boxed{[\![v]\!]}$ and computations $\boxed{[\![m]\!]K}$ in Figure 8 show how they are used; we delay the translation of case expressions for the moment.

The translation of computations takes a continuation as a second argument, representing the rest of the computation that expects the result of the translated computation. The translation of values is directly recursive on the term, with the translation of thunks being the thunk of the translated computation using the empty continuation, since there's no computation left to do inside the thunk.

$$
\begin{array}{rl}
[\![x]\!] & := x \\
[\![()]\!] & := () \\
[\![\mathsf{inl}\ v]\!] & := \mathsf{inl}\ [\![v]\!] \\
[\![\mathsf{inr}\ v]\!] & := \mathsf{inr}\ [\![v]\!] \\
[\![\{m\}]\!] & := \{[\![m]\!]\Box\}
\end{array}
$$

(a) Translation of values [CCNF:CCval]

$$
\begin{array}{rl}
[\![v!]\!]K & := K[[\![v]\!]!] \\
[\![\mathsf{return}\ v]\!]K & := K[\mathsf{return}\ [\![v]\!]] \\
[\![\lambda x.\ m]\!]K & := K[\lambda x.\ [\![m]\!]\Box] \\
[\![\langle m_1, m_2 \rangle]\!]K & := K[\langle [\![m_1]\!]\Box, [\![m_2]\!]\Box \rangle] \\
[\![m\ v]\!]K & := [\![m]\!](K :: \Box\ v) \\
[\![\mathsf{fst}\ m]\!]K & := [\![m]\!](K :: \mathsf{fst}\ \Box) \\
[\![\mathsf{snd}\ m]\!]K & := [\![m]\!](K :: \mathsf{snd}\ \Box) \\
[\![\mathsf{let}\ x \leftarrow m_1\ \mathsf{in}\ m_2]\!]K & := [\![m_1]\!](\mathsf{let}\ x \leftarrow \Box\ \mathsf{in}\ [\![m_2]\!]K)
\end{array}
$$

(b) Translation of computations [CCNF:CCval]

Fig. 8. Commuting conversion normalization translation (excluding case)

$$(\text{let } x \leftarrow \square \text{ in } m)[n] \coloneqq \text{let } x \leftarrow n \text{ in } m \qquad (K :: \text{fst } \square)[n] \coloneqq K[\text{fst } n] \qquad \square[n] \coloneqq n$$
$$(K :: \square\ v)[n] \coloneqq K[n\ v] \qquad (K :: \text{snd } \square)[n] \coloneqq K[\text{snd } n]$$

Fig. 9. Plugging a computation in a continuation [CCNF:plug]

To define the translation of computations, we need a plugging operation $\boxed{K[n]}$ that collapses the stack of ks in the continuation and replaces the final hole $\square$ by the given computation n, defined in Figure 9. Although the operation is defined over all computations, as the holes only appear in n positions, we can only plug tail-free computations into continuations if we want to produce a configuration. This is the case for the translations of forcing thunks, returning values, functions, and computation pairs: we translate their subterms—using empty continuations if needed, since we don't commute into introduction forms—and plug them directly into the continuation. (Section 6.3 discusses why we don't commute.)

In the translation of let expressions, the evaluation order would first compute m_1, then m_2, followed by whatever computation remains in K. Therefore, we translate m_1 using a let continuation that represents binding the result of m_1 to x then running the translation of m_2. For function applications and pair projections, we translate the subterm, which we expect to yield a function or a pair, under the continuation extended with application or projection, respectively.

To see how the translation yields CCNFs, we can look at how they act on the left-hand side of the commuting conversions in Equations 2 and 3, assuming that n is already in CCNF, and using the fact that $[\![n]\!]K = K[n]$.

$$\begin{aligned}
&[\![\text{let } x \leftarrow (\text{let } y \leftarrow n \text{ in return } v) \text{ in } m]\!]\square && \text{(Equation 2)}\\
&= [\![\text{let } x \leftarrow n \text{ in return } v]\!](\text{let } y \leftarrow \square \text{ in } [\![m]\!]\square) && (\textit{let})\\
&= [\![n]\!](\text{let } x \leftarrow \square \text{ in } [\![\text{return } v]\!](\text{let } y \leftarrow \square \text{ in } [\![m]\!]\square)) && (\textit{let})\\
&= (\text{let } x \leftarrow \square \text{ in } (\text{let } y \leftarrow \square \text{ in } [\![m]\!]\square)[\text{return } [\![v]\!]])[n] && (n, \textit{ret})\\
&= \text{let } x \leftarrow n \text{ in } (\text{let } y \leftarrow \text{return } [\![v]\!] \text{ in } [\![m]\!]\square) && (\textit{plug})\\
&[\![(\text{let } x \leftarrow n \text{ in } \lambda y.\ m)\ v]\!]\square && \text{(Equation 3)}\\
&= [\![\text{let } x \leftarrow n \text{ in } \lambda y.\ m]\!](\square\ [\![v]\!]) && (\textit{app})\\
&= [\![n]\!](\text{let } x \leftarrow \square \text{ in } [\![\lambda y.\ m]\!](\square\ [\![v]\!])) && (\textit{let})\\
&= (\text{let } x \leftarrow \square \text{ in } (\square\ [\![v]\!])[\lambda y.\ [\![m]\!]\square])[n] && (n, \textit{fun})\\
&= \text{let } x \leftarrow n \text{ in } (\lambda y.\ [\![m]\!]\square)\ [\![v]\!] && (\textit{plug})
\end{aligned}$$

Branching and join points. The naïve translation of conditionals duplicates the continuation:

$$[\![\text{if } v \text{ then } m_1 \text{ else } m_2]\!]K = \text{if } [\![v]\!] \text{ then } [\![m_1]\!]K \text{ else } [\![m_2]\!]K.$$

If K only contains application and projection continuations, this is the desired translation. From fst (if v then $\langle m_1, m_2\rangle$ else n) we get if v then fst $\langle m_1, m_2\rangle$ else fst n, whose true branch can then be inlined. It doesn't make sense to jump to a join point that performs the projection, since it would make the inlining opportunity harder to find, and require unnecessarily thunking the computation pair to pass it to the join point.

If the continuation is let $x \leftarrow \square$ in m, duplicating it would duplicate the arbitrary configuration m. In this case, we move m to a join point and replace it in the continuation by a jump. For example, from let $x \leftarrow$ (if v then return w else n) in m we get

$$\text{join } j\ x = m \text{ in } (\text{if } v \text{ then } (\text{let } x \leftarrow \text{return } w \text{ in jump } j\ x) \text{ else } (\text{let } x \leftarrow n \text{ in jump } j\ x)),$$

$$[\![\text{case } v \text{ of } \{\text{inl } x \Rightarrow m_1; \text{inr } y \Rightarrow m_2\}]\!]K :=$$

$$\begin{cases} \text{case } [\![v]\!] \text{ of } \{\text{inl } y \Rightarrow [\![m_1]\!]K; \text{inr } z \Rightarrow [\![m_2]\!]K\} & \text{if } K \equiv \Box :: k_1 :: \ldots :: k_i \\ & \text{or } K \equiv (\text{let } x \leftarrow \Box \text{ in jump } j\ x) :: k_1 :: \ldots :: k_i \\ \text{join } j\ x = m \text{ in} & \text{if } K \equiv (\text{let } x \leftarrow \Box \text{ in } m) :: k_1 :: \ldots :: k_i \\ \quad \text{case } [\![v]\!] \text{ of } \{\text{inl } y \Rightarrow [\![m_1]\!]K'; \text{inr } z \Rightarrow [\![m_2]\!]K'\} & \text{and } m \not\equiv \text{jump } j\ x \\ & \text{where } K' := (\text{let } x \leftarrow \Box \text{ in jump } j\ x) :: k_1 :: \ldots :: k_i \end{cases}$$

Fig. 10. Translation of case expressions to CCNF with join points [CCNF:CCcom,K.jumpify]

and the true branch can still inline the let binding. However, we avoid creating extra join points in nested branching. For example, from let $x \leftarrow (\text{if } v \text{ then } n_1 \text{ else } (\text{if } w \text{ then } n_2 \text{ else } n_3))$ in m we get

$$\begin{array}{l} \text{join } j\ x = m \text{ in} \\ \quad \text{if } v \text{ then } (\text{let } x \leftarrow n_1 \text{ in jump } j\ x) \text{ else} \\ \quad\quad (\text{if } w \text{ then } (\text{let } x \leftarrow n_2 \text{ in jump } j\ x) \text{ else } (\text{let } x \leftarrow n_2 \text{ in jump } j\ x)), \end{array}$$

rather than another extraneous join point $j'\ x = \text{jump } j\ x$ in the outer false branch.

Figure 10 generalizes this strategy to case expressions and nested continuations; whether we construct a join point depends on whether the continuation ends in a let continuation whose body isn't already a jump to a join point. In the mechanization, this is a source-to-source translation over CBPV terms (without join points) to CBPV terms (with join points). We need to explicitly prove that the translation indeed produces terms in CCNF, which holds by inspection, *i.e.* mutual induction over the syntax.

Lemma 4.1 (Plugging preserves CCNF). [CCNF:isK.plug]
If the subterms of K are in CCNF and n is in CCNF, then $K[n]$ is in CCNF.

Lemma 4.2 (CCNF preservation). [CCNF:isCCNF,isK.jumpify]
If the subterms of K are in CCNF, then $[\![v]\!]$ and $[\![m]\!]K$ are in CCNF.

4.1 Commuting conversion preserves typing

With Lemma 4.2, we know that the translation performs all the commuting conversions in the directions we desire. However, this alone doesn't guarantee that the translation preserves the *meaning* of the values and computations. We say what we mean by meaning preservation in the next section, whose proof uses typing preservation: a translated term must behave like the same kind of term as the original term.

$$\boxed{\Gamma \mid \Delta \vdash K : B_1 \Rightarrow B_2}$$

$$\text{K-let}\ \frac{\Gamma, x : A \mid \Delta \vdash m : B}{\Gamma \mid \Delta \vdash \text{let } x \leftarrow \Box \text{ in } m : \mathsf{F}\,A \Rightarrow B} \qquad \text{K-app}\ \frac{\Gamma \vdash v : A \qquad \Gamma \mid \Delta \vdash K : B_1 \Rightarrow B_2}{\Gamma \mid \Delta \vdash K :: \Box\ v : (A \to B_1) \Rightarrow B_2}$$

$$\text{K-fst}\ \frac{\Gamma \mid \Delta \vdash K : B_1 \Rightarrow B}{\Gamma \mid \Delta \vdash K :: \text{fst } \Box : (B_1\ \&\ B_2) \Rightarrow B} \qquad \text{K-snd}\ \frac{\Gamma \mid \Delta \vdash K : B_1 \Rightarrow B}{\Gamma \mid \Delta \vdash K :: \text{snd } \Box : (B_1\ \&\ B_2) \Rightarrow B} \qquad \text{K-hole}\ \frac{}{\Gamma \mid \Delta \vdash \Box : B \Rightarrow B}$$

Fig. 11. Typing rules for continuations [CCNF:wtK]

To show type preservation, we first need a typing judgement for the continuations, given in Figure 11. A continuation K having type $B_1 \Rightarrow B_2$ means that if the hole in K represents a missing computation of type B_1, then K with its hole plugged in would be a computation of type B_2. Consequently, we can show that plugging preserves typing, which is used in the proof of type preservation. We also need an inversion lemma for continuations that end in let continuations.

Lemma 4.3 (Plugging preserves typing). [CCNF:wtK.plug]
If $\Gamma \mid \Delta \vdash K : B_1 \Rightarrow B_2$ *and* $\Gamma \mid \cdot \vdash n : B_1$, *then* $\Gamma \mid \Delta \vdash K[n] : B_2$.

Proof. By induction on the typing derivation of K, using Lemma 3.6 in the K-hole case. □

Lemma 4.4 (Let continuation inversion). [CCNF:wtK.jumpify] *If* $K = (\mathsf{let} x \leftarrow \square \mathsf{\ in\ } m)[k_1] \ldots [k_i]$ *and* $\Gamma \mid \Delta \vdash K : B_1 \Rightarrow B_2$, *then there exists some* A *such that* $\Gamma \mid \Delta, j : A \nearrow B_2 \vdash K' : B_1 \Rightarrow B_2$ *and* $\Gamma, x : A \mid \Delta \vdash m : B_2$, *where* $K' = (\mathsf{let}\ x \leftarrow \square\ \mathsf{in}\ \mathsf{jump}\ j\ x)[k_1] \ldots [k_i]$.

Proof. By induction on the typing derivation of K. □

Lemma 4.5 (Type preservation). [CCNF:preservation] *Suppose* v *and* m *are plain CBPV terms (and thus have no join points or jumps). If* $\Gamma \vdash v : A$, *then* $\Gamma \vdash [\![v]\!] : A$, *and if* $\Gamma \mid \cdot \vdash m : B_1$ *and* $\Gamma \mid \Delta \vdash K : B_1 \Rightarrow B_2$, *then* $\Gamma \mid \Delta \vdash [\![m]\!]K : B_2$.

Proof. By mutual induction on the typing derivations of v and m, using Lemma 4.3 in the T-force, T-fun, and T-pair cases. In the T-case case, if K is a let continuation, use Lemma 4.4 and rule T-join to construct the derivation. □

5 Commuting Conversion Normalization Preserves Evaluation

Normalizing by all commuting conversions shouldn't affect the meaning of a program. Formally, if a closed computation evaluates to a returned value, then its translation runs to the same value. Because we don't evaluate inside of thunks, we consider only computations that return *ground* values, which are those of type $T ::= \top \mid T + T$. Otherwise, a thunk and its CCNF are values that don't evaluate any further, but aren't necessarily syntactically equal.

Theorem 5.1. *Given* m *such that* $\cdot \mid \cdot \vdash m : \mathsf{F}\ T$, *if* $m \Downarrow \mathsf{return}\ v$, *then* $[\![m]\!]\square \Downarrow \mathsf{return}\ v$.

We prove this property as a corollary of a logical equivalence between a computation and its translation. This machinery is required because the simpler of method using a simulation argument, such as the following statement, unfortunately doesn't work.

Falsehood 5.1 (Simulation). *If* $m \rightsquigarrow n$ *then* $[\![m]\!]\square \rightsquigarrow^* [\![n]\!]\square$.

Counterexample. Suppose we have tail-free computations n_1, n_2 and configuration m. Consider the term $\mathsf{let}\ x \leftarrow \{\mathsf{let}\ y \leftarrow n_1\ \mathsf{in}\ n_2\}!\ \mathsf{in}\ m$, which reduces to $\mathsf{let}\ x \leftarrow (\mathsf{let}\ y \leftarrow n_1\ \mathsf{in}\ n_2)\ \mathsf{in}\ m$. The left-hand side translates to itself, since it's already in CCNF, while the right-hand side translates to $\mathsf{let}\ y \leftarrow n_1\ \mathsf{in}\ \mathsf{let}\ x \leftarrow n_2\ \mathsf{in}\ m$. By transitivity, it remains to show that $\mathsf{let}\ x \leftarrow (\mathsf{let}\ y \leftarrow n_1\ \mathsf{in}\ n_2)\ \mathsf{in}\ m \rightsquigarrow^* \mathsf{let}\ y \leftarrow n_1\ \mathsf{in}\ \mathsf{let}\ x \leftarrow n_2\ \mathsf{in}\ m$, but there is no such reduction sequence since there is no reduction step that commutes let bindings. □

However, if we know that n_1 in the counterexample reduces to some return v, then we can deduce that the right side and its translation must reduce to $\mathsf{let}\ x \leftarrow n_2\{y \mapsto v\}\ \mathsf{in}\ m$, which gives us an equivalence between the original term and its translation. If our counterexample is well typed, then a logical equivalence gives us precisely the required information that subterms must reduce to canonical terms such as returned values.

To handle translations with arbitrary translation continuation K, we generalize to proving that a translation $[\![m]\!]K$ must be equivalent to plugging the computation back in as $K[m]$. The proof of Theorem 5.1 then proceeds by:

1. Defining a standard logical equivalence over CBPV types;
2. Closing over term and join point contexts with a semantic equivalence;
3. Proving the fundamental theorem of this equivalence, namely that well-typed terms are semantically equivalent to themselves;
4. Showing that well-typed commuting conversions are in the semantic equivalence;
5. Defining semantic equivalence of continuations and proving its fundamental theorem;
6. Showing that plugging continuations respects semantic equivalence; and finally,
7. Proving that given a well-typed computation and a well-typed continuation, plugging the continuation with the computation is equivalent to translating the computation using the continuation.

The theorem then holds by instantiating with the empty continuation. Section 5.1 covers step 1 to 3, Section 5.2 covers step 4, Section 5.3 covers step 5 to 6, and Section 5.4 covers step 7.

5.1 Logical equivalence and the fundamental theorem

$$\boxed{(v, w) \in [\![A]\!]} \quad \boxed{(m, n) \in [\![B]\!]} \quad \boxed{(m, n) \in [\![B]\!]^*}$$

$$(( ), ( )) \in [\![\top]\!]$$

$$(\mathsf{inl}\ v, \mathsf{inl}\ w) \in [\![A_1 + A_2]\!] \ \mathit{iff}\ (v, w) \in [\![A_1]\!]$$

$$(\mathsf{inr}\ v, \mathsf{inr}\ w) \in [\![A_1 + A_2]\!] \ \mathit{iff}\ (v, w) \in [\![A_2]\!]$$

$$(\{m\}, \{n\}) \in [\![\mathsf{U}\ B]\!] \ \mathit{iff}\ (m, n) \in [\![B]\!]^*$$

$$(\mathsf{return}\ v, \mathsf{return}\ w) \in [\![\mathsf{F}\ A]\!] \ \mathit{iff}\ (v, w) \in [\![A]\!]$$

$$(\lambda x.\ m, \lambda x.\ n) \in [\![A \to B]\!] \ \mathit{iff}\ \forall v, w.\ (v, w) \in [\![A]\!] \Rightarrow (m\{x \mapsto v\}, n\{x \mapsto w\}) \in [\![B]\!]^*$$

$$(\langle m_1, m_2 \rangle, \langle n_1, n_2 \rangle) \in [\![B_1\ \&\ B_2]\!] \ \mathit{iff}\ (m_1, n_1) \in [\![B_1]\!]^* \wedge (m_2, n_2) \in [\![B_2]\!]^*$$

$$(m, n) \in [\![B]\!]^* \ \mathit{iff}\ \exists tm_1, tm_2.\ (m \Downarrow tm_1) \wedge (n \Downarrow tm_2) \wedge (tm_1, tm_2) \in [\![B]\!]$$

Fig. 12. Logical equivalence of terms over CBPV types [Equivalence:$\mathscr{V}, \mathscr{C}, \mathscr{E}$]

Our logical relation relates two closed values or computations at a value or computation type, respectively. We equivalently say that a pair of terms are in the interpretation of some type. Following the presentation by Forster et al. [2019], we use an auxiliary interpretation $[\![B]\!]^*$ which relates computations that normalize to terminals related at B. It follows that if $(m, n) \in [\![B]\!]$, then $(m, n) \in [\![B]\!]^*$.

The interpretations, defined by mutual recursion over types, are otherwise standard: unit values are related, left and right injections are related when their values are related at the respective left or right type, thunks are related when their computations are related, functions are related when their bodies are related for all related arguments, and pairs are related when their first and second components are respectively related.

The first property we need of logical equivalence is backward closure under evaluation of $[\![B]\!]^*$, which holds by unfolding its definition and transitivity of evaluation. A helpful corollary adds congruence under join points.

Lemma 5.2 (Backward closure). [Equivalence:bwds]
If $m_1 \rightsquigarrow^* m_2$ *and* $n_1 \rightsquigarrow^* n_2$ *and* $(m_2, n_2) \in [\![B]\!]^*$, *then* $(m_1, n_1) \in [\![B]\!]^*$.

$\boxed{(\sigma, \tau) \in [\![\Gamma]\!]}$ $\boxed{(\varphi, \psi) \in [\![\Delta]\!]}$

$$\textsc{S-nil}\ \frac{}{(\bullet, \bullet) \in \bullet} \qquad \textsc{S-cons}\ \frac{(\sigma, \tau) \in [\![\Gamma]\!] \qquad (v, w) \in [\![A]\!]}{((\sigma, x \mapsto v), (\tau, x \mapsto w)) \in [\![\Gamma, x : A]\!]} \qquad \textsc{J-nil}\ \frac{}{(\bullet, \bullet) \in \bullet}$$

$$\textsc{J-cons}\ \frac{(\varphi, \psi) \in [\![\Delta]\!] \qquad \forall v, w. (v, w) \in [\![A]\!] \Rightarrow (\text{joins } \varphi \text{ in } m\{x \mapsto v\}, \text{joins } \psi \text{ in } n\{x \mapsto w\}) \in [\![B]\!]}{((\varphi,\ j\ x = m), (\psi,\ j\ x = n)) \in [\![\Delta, j : A \nearrow B]\!]}$$

Fig. 13. Logical equivalence of substitution maps and join stacks [Equivalence:semCtxt,semDtxt]

Lemma 5.3 (Backward closure under join points). [Equivalence:bwdsRejoin]
If $m_1 \rightsquigarrow^* m_2$ *and* $n_1 \rightsquigarrow^* n_2$ *and* $(m_2, n_2) \in [\![B]\!]^*$, *then* $(\text{join} j x = m' \text{ in } m_1, \text{join} j x = n' \text{ in } m_2) \in [\![B]\!]^*$, *using rules* E-join *and* E-drop.

The second property is symmetry and transitivity, making it a partial equivalence relation over all terms. These lemmas are proven by induction over the type and unfolding definitions.

Lemma 5.4 (Symmetry and transitivity (logical equivalence)).

- *If* $(v, w) \in [\![A]\!]$ *then* $(w, v) \in [\![A]\!]$. [Equivalence:$\mathscr{V}$.sym]
- *If* $(m, n) \in [\![B]\!]$ *then* $(n, m) \in [\![B]\!]$. [Equivalence:$\mathscr{C}$.sym]
- *If* $(m, n) \in [\![B]\!]^*$ *then* $(n, m) \in [\![B]\!]^*$. [Equivalence:$\mathscr{E}$.sym]
- *If* $(v_1, v_2) \in [\![A]\!]$ *and* $(v_2, v_3) \in [\![A]\!]$ *then* $(v_1, v_3) \in [\![A]\!]$. [Equivalence:$\mathscr{V}$.trans]
- *If* $(m_1, m_2) \in [\![B]\!]$ *and* $(m_2, m_3) \in [\![B]\!]$ *then* $(m_1, m_3) \in [\![B]\!]$. [Equivalence:$\mathscr{C}$.trans]
- *If* $(m_1, m_2) \in [\![B]\!]^*$ *and* $(m_2, m_3) \in [\![B]\!]^*$ *then* $(m_1, m_3) \in [\![B]\!]^*$. [Equivalence:$\mathscr{E}$.trans]

To work with equivalence of well-typed terms, which may be open with respect to value and jump contexts, we need close over value and jump variables using substitution maps σ and join stacks φ, defined below. Applying a substitution map $v\{\sigma\}$, $m\{\sigma\}$ corresponds to simultaneous substitution (so order doesn't matter in σ). Given a join stack $\varphi \equiv j_1\ x_1 = m_1, \ldots, j_i\ x_i = m_i$, the computation joins φ in m represents wrapping m in join points outward in, corresponding to the term join $j_1\ x_1 = m_1$ in ... join $j_i\ x_i = m_i$ in m (so order *does* matter in φ).

$$\sigma, \tau ::= \cdot \mid \sigma, x \mapsto v \qquad \varphi, \psi ::= \cdot \mid \varphi,\ j\ x = m \qquad \text{[Rejoin:J]}$$

We extend logical equivalence to substitution maps and join stacks, relating two of them at a particular context. Figure 13 gives inductive rules for these logical equivalences. Rules S-nil and S-cons are equivalent to stating that $(\sigma, \tau) \in [\![\Gamma]\!]$ holds when for all $x : A \in \Gamma$, $(\sigma(x), \tau(x)) \in [\![A]\!]$ holds. Rule J-cons states that we can extend a pair of related join stacks φ and ψ by join points m and n of type $A \nearrow B$ when the join points themselves are related at B given arguments related at A *and* closed over by φ and ψ. They need to be closed over since m and n themselves may jump to earlier join points. We can show that substitution maps and join stacks are partial equivalence relations by induction on their derivations.

Lemma 5.5 (Symmetry and transitivity (logical equivalence at contexts)).

- *If* $(\sigma, \tau) \in [\![\Gamma]\!]$ *then* $(\tau, \sigma) \in [\![\Gamma]\!]$. [Equivalence:semCtxt.sym]
- *If* $(\varphi, \psi) \in [\![\Delta]\!]$ *then* $(\varphi, \psi) \in [\![\Delta]\!]$. [Equivalence:semDtxt.sym]
- *If* $(\sigma_1, \sigma_2) \in [\![\Gamma]\!]$ *and* $(\sigma_2, \sigma_3) \in [\![\Gamma]\!]$ *then* $(\sigma_1, \sigma_3) \in [\![\Gamma]\!]$. [Equivalence:semCtxt.trans]
- *If* $(\varphi_1, \varphi_2) \in [\![\Delta]\!]$ *and* $(\varphi_2, \varphi_3) \in [\![\Delta]\!]$ *then* $(\varphi_1, \varphi_3) \in [\![\Delta]\!]$. [Equivalence:semDtxt.trans]

Using substitution maps and join stacks, we can define semantic equivalence of open terms. They are also partial equivalence relations, which follows from Lemmas 5.4 and 5.5. In all proofs that follow, we freely use symmetry and transitivity without explicit reference.

Definition 5.6 (Semantic equivalence of values). [Equivalence:semVal]
Values v and w are semantically equivalent under context Γ at type A, written $\boxed{\Gamma \vDash v \sim w : A}$, when for all substitution maps $(\sigma, \tau) \in [\![\Gamma]\!]$, $(v\{\sigma\}, w\{\tau\}) \in [\![A]\!]$ holds.

Definition 5.7 (Semantic equivalence of computations). [Equivalence:semCom]
Computations m and n are semantically equivalent under contexts Γ and Δ at type B, written $\boxed{\Gamma \mid \Delta \vDash m \sim n : B}$, when for all substitution maps $(\sigma, \tau) \in [\![\Gamma]\!]$ and join stacks $(\varphi, \psi) \in [\![\Delta]\!]$, $(\text{joins } \varphi \text{ in } m\{\sigma\}, \text{joins } \psi \text{ in } n\{\tau\}) \in [\![B]\!]$ holds.

Lemma 5.8 (Symmetry and transitivity (semantic equivalence)).

- *If* $\Gamma \vDash v \sim w : A$ *then* $\Gamma \vDash w \sim v : A$. [Equivalence:semVal.sym]
- *If* $\Gamma \mid \Delta \vDash m \sim n : B$ *then* $\Gamma \mid \Delta \vDash n \sim m : B$. [Equivalence:semCom.sym]
- *If* $\Gamma \vDash v_1 \sim v_2 : A$ *and* $\Gamma \vDash v_2 \sim v_3 : A$ *then* $\Gamma \vDash v_1 \sim v_3 : A$. [Equivalence:semVal.trans]
- *If* $\Gamma \mid \Delta \vDash m_1 \sim m_2 : B$ *and* $\Gamma \mid \Delta \vDash m_2 \sim m_3 : B$ *then* $\Gamma \mid \Delta \vDash m_1 \sim m_3 : B$. [Equivalence:semCom.trans]

Finally, we show the fundamental theorem of semantic equivalence: well-typed terms are semantically equivalent to themselves.

Theorem 5.9 (Fundamental theorem of semantic equivalence). [Equivalence:soundness]
If $\Gamma \vdash v : A$ *then* $\Gamma \vDash v \sim v : A$, *and if* $\Gamma \mid \Delta \vdash m : B$ *then* $\Gamma \mid \Delta \vDash m \sim m : B$.

Proof. By induction on the typing derivations of v and m. Letting $(\sigma, \tau) \in [\![\Gamma]\!]$ and $(\varphi, \psi) \in [\![\Delta]\!]$, the goal is to show that $(v\{\sigma\}, v\{\tau\}) \in [\![A]\!]$ and $(\text{joins } \varphi \text{ in } m\{\sigma\}, \text{joins } \psi \text{ in } m\{\tau\}) \in [\![B]\!]^*$ hold.

- **Case** T-var: Holds by $(\sigma, \tau) \in [\![\Gamma]\!]$ on the variable.
- **Cases** T-unit, T-left, T-right, and T-thunk: Hold by definition of the logical relation, using the induction hypotheses on any subterms.
- **Cases** T-fun, T-ret, and T-prod: Hold by Lemma 5.3 with no reduction, using the induction hypotheses on subterms.
- **Cases** T-app, T-fst, and T-snd: By the induction hypotheses on the scrutinee premises, they must reduce to related functions, related first pair projections, or related second pair projections, respectively. Then the goal holds by Lemma 5.3 on these relations, reducing by E-app, E-fst, or E-snd, respectively.
- **Cases** T-let and T-case: By the induction hypotheses on the scrutinee premises, they must reduce to related returns, related left injections, or related right injections, respectively. Then the goal holds by Lemma 5.2 on these relations, reducing by E-ret, E-left, or E-right, respectively.
- **Case** T-join: Holds by the induction hypothesis on the join expression body, extending semantic equivalence of join stacks using the induction hypothesis on the join points.
- **Case** T-jump: Holds by $(\varphi, \psi) \in [\![\Delta]\!]$ on the jump variable, instantiating the join point by the induction hypothesis on related values. □

As a corollary, we have type safety and normalization: a closed, well-typed computation never evaluates to a stuck term, and moreover evaluates to a terminal.

Corollary 5.10 (Normalization). [Equivalence:safety] *If* $\cdot \mid \cdot \vdash m : B$ *then* $\exists tm, m \rightsquigarrow^* tm$.

$$\frac{\Gamma \mid \cdot \vdash \mathsf{let}\ x \leftarrow n\ \mathsf{in}\ m : \mathsf{F}\ A \qquad \Gamma, y : A \mid \Delta \vdash m' : B}{\Gamma \mid \Delta \vDash \mathsf{let}\ y \leftarrow (\mathsf{let}\ x \leftarrow n\ \mathsf{in}\ m)\ \mathsf{in}\ m' \sim \mathsf{let}\ x \leftarrow n\ \mathsf{in}\ \mathsf{let}\ y \leftarrow m\ \mathsf{in}\ m' : B}\ \textsc{let-let}$$

$$\frac{\Gamma \mid \cdot \vdash \mathsf{let}\ x \leftarrow n\ \mathsf{in}\ m : A \to B \qquad \Gamma \vdash v : A}{\Gamma \mid \Delta \vDash (\mathsf{let}\ x \leftarrow n\ \mathsf{in}\ m)\ v \sim \mathsf{let}\ x \leftarrow n\ \mathsf{in}\ (m\ v) : B}\ \textsc{let-app}$$

$$\frac{\Gamma \mid \cdot \vdash \mathsf{let}\ x \leftarrow n\ \mathsf{in}\ m : B_1 \,\&\, B_2}{\Gamma \mid \Delta \vDash \mathsf{fst}\ (\mathsf{let}\ x \leftarrow n\ \mathsf{in}\ m) \sim \mathsf{let}\ x \leftarrow n\ \mathsf{in}\ (\mathsf{fst}\ m) : B_1}\ \textsc{let-fst}$$

$$\frac{\Gamma \mid \cdot \vdash \mathsf{let}\ x \leftarrow n\ \mathsf{in}\ m : B_1 \,\&\, B_2}{\Gamma \mid \Delta \vDash \mathsf{snd}\ (\mathsf{let}\ x \leftarrow n\ \mathsf{in}\ m) \sim \mathsf{let}\ x \leftarrow n\ \mathsf{in}\ (\mathsf{snd}\ m) : B_2}\ \textsc{let-snd}$$

$$\frac{\Gamma \mid \cdot \vdash \mathsf{case}\ v\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow m_1; \mathsf{inr}\ y \Rightarrow m_2\} : \mathsf{F}\ A \qquad \Gamma, z : A \mid \Delta \vdash m : B}{\begin{array}{l}\Gamma \mid \Delta \vDash \mathsf{let}\ z \leftarrow (\mathsf{case}\ v\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow m_1; \mathsf{inr}\ y \Rightarrow m_2\})\ \mathsf{in}\ m \\ \quad \sim \mathsf{case}\ v\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow (\mathsf{let}\ z \leftarrow m_1\ \mathsf{in}\ m); \mathsf{inr}\ y \Rightarrow (\mathsf{let}\ z \leftarrow m_2\ \mathsf{in}\ m)\} : B\end{array}}\ \textsc{case-let}$$

$$\frac{\Gamma \mid \cdot \vdash \mathsf{case}\ v\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow m_1; \mathsf{inr}\ y \Rightarrow m_2\} : A \to B \qquad \Gamma \vdash w : A}{\begin{array}{l}\Gamma \mid \Delta \vDash (\mathsf{case}\ v\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow m_1; \mathsf{inr}\ y \Rightarrow m_2\})\ w \\ \quad \sim \mathsf{case}\ v\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow (m_1\ w); \mathsf{inr}\ y \Rightarrow (m_2\ w)\} : B\end{array}}\ \textsc{case-app}$$

$$\frac{\Gamma \mid \cdot \vdash \mathsf{case}\ v\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow m_1; \mathsf{inr}\ y \Rightarrow m_2\} : B_1 \,\&\, B_2}{\begin{array}{l}\Gamma \mid \Delta \vDash \mathsf{fst}\ (\mathsf{case}\ v\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow m_1; \mathsf{inr}\ y \Rightarrow m_2\}) \\ \quad \sim \mathsf{case}\ v\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow (\mathsf{fst}\ m_1); \mathsf{inr}\ y \Rightarrow (\mathsf{fst}\ m_2)\} : B_1\end{array}}\ \textsc{case-fst}$$

$$\frac{\Gamma \mid \cdot \vdash \mathsf{case}\ v\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow m_1; \mathsf{inr}\ y \Rightarrow m_2\} : B_1 \,\&\, B_2}{\begin{array}{l}\Gamma \mid \Delta \vDash \mathsf{snd}\ (\mathsf{case}\ v\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow m_1; \mathsf{inr}\ y \Rightarrow m_2\}) \\ \quad \sim \mathsf{case}\ v\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow (\mathsf{snd}\ m_1); \mathsf{inr}\ y \Rightarrow (\mathsf{snd}\ m_2)\} : B_2\end{array}}\ \textsc{case-snd}$$

Fig. 14. Semantic equivalence of commuting conversions
[Commutation:letLet,appLet,fstLet,sndLet,letCase,appCase,fstCase,sndCase]

5.2 Semantic equivalence of commuting conversions

As the translation normalizes with respect to all commuting conversions, to show that semantic equivalence of the translation, naturally we need to show semantic equivalence of the commuting conversions. Recall that with four evaluation contexts and two tail contexts, there are eight total commuting conversions to handle, each assuming well-typedness of various subterms. We present them as derivable rules in Figure 14 for legibility.

The general strategy to proving these is by using the Theorem 5.9 on the typing derivations, extracting a logical equivalence for the desired type, applying Lemma 5.2 or Lemma 5.3, then using the evaluation rules and Corollary 3.4 to find the correct evaluation sequence from the left and right sides of the lemma statement to the respective sides of the extracted logical equivalence. We step through only the proof of LET-APP as a representative case.

Proof. Because substitution maps and join stacks don't play much of a role, we mostly ignore them here. The goal is then to show that (let $x \leftarrow n$ in m) v is equivalent to let $x \leftarrow n$ in $(m\ v)$, knowing that all subterms are well typed. By Theorem 5.9, we have that n reduces to some return w, and that $m\{x \mapsto w\}$ reduces to some $\lambda y.\ m'$. Then we know that the right-hand side reduces by E-ret and E-app to $m'\{y \mapsto v\}$. We also have that let $x \leftarrow n$ in m reduces to $m\{x \mapsto w\}$ by E-ret. Then we know that the left-hand side reduces by E-app also to $m'\{y \mapsto v\}$. Therefore, by Lemma 5.3, the left- and right-hand sides are equivalent. □

Although our translation doesn't involve commuting join points, we do require a corresponding semantic equivalence to unnest join points, stated in Figure 15.

$$\textsc{join-join}\quad \frac{\Gamma, x_1 : A_1 \mid \Delta \vdash m_1 : B \qquad \Gamma, x_2 : A_2 \mid \Delta, j_1 : A_1 \nearrow B \vdash m_2 : B \qquad \Gamma \mid \Delta, j_2 : A_2 \nearrow B \vdash m : B}{\Gamma \mid \Delta \vDash \mathsf{join}\ j_2\ x_2 = (\mathsf{join}\ j_1\ x_1 = m_1\ \mathsf{in}\ m_2)\ \mathsf{in}\ m \sim \mathsf{join}\ j_1\ x_1 = m_1\ \mathsf{in}\ \mathsf{join}\ j_2\ x_2 = m_2\ \mathsf{in}\ m : B}$$

Fig. 15. Semantic equivalence of commuting join points [Commutation:joinJoin]

On the right-hand side, m can't directly jump to j_1, since it isn't in scope in its typing, so if m doesn't jump to j_2, then both join points can be dropped. In the mechanization, this isn't so easy to prove, since we're using de Bruijn indexed jump variables. Consequently, m must be explicitly weakened on the right-hand side, and to show that join points can be dropped, we first need to show that evaluation preserves weakening. These technical details aside, the proof proceeds similarly to those for the commuting conversions.

5.3 Semantic equivalence of plugging continuations

Just as the Fundamental theorem of semantic equivalence requires related substitution maps and related join stacks, for the final proof in the next section, we need a notion of related continuations. Rather than first defining a logical equivalence over continuations, we directly define semantic equivalence in a way that incorporates plugging. Here, $K\{\sigma\}$ denotes applying the substitution map σ to all value and computation subterms of K.

Definition 5.11 (Semantic equivalence of continuations). [Soundness:semK]
Continuations K_1 and K_2 are semantically equivalent under Γ and Δ at type $B_1 \Rightarrow B_2$, written $\boxed{\Gamma \mid \Delta \vDash K_1 \sim K_2 : B_1 \Rightarrow B_2}$, when for all substitution maps $(\sigma, \tau) \in [\![\Gamma]\!]$, join stacks $(\varphi, \psi) \in [\![\Delta]\!]$, and computations $(n_1, n_2) \in [\![B_1]\!]^*$, $(\mathsf{joins}\ \varphi\ \mathsf{in}\ K\{\sigma\}[n_1], \mathsf{joins}\ \psi\ \mathsf{in}\ K\{\tau\}[n_2]) \in [\![B_2]\!]$ holds.

A fundamental theorem for semantic equivalence of continuations holds as well, which relies on the corresponding theorem for terms, as well as congruence of evaluating under plugging.

Lemma 5.12 (E-plug). [CCNF:Evals.plug]
If $n_1 \rightsquigarrow^* n_2$, then $K[n_1] \rightsquigarrow^* K[n_2]$, by induction on the structure of K.

Theorem 5.13 (Fundamental theorem of semantic equivalence of continuations).
If $\Gamma \mid \Delta \vdash K : B_1 \Rightarrow B_2$ then $\Gamma \mid \Delta \vDash K \sim K : B_1 \Rightarrow B_2$. [Soundness:soundK]

Proof. By induction on the typing derivation of K. In the K-app case, use Theorem 5.9 on the value argument, and in the K-let case, use Theorem 5.9 on the computation body. In the K-app, K-fst, and K-snd cases, use E-plug to reduce under the rest of the continuation. □

Plugging semantically equivalent computations into both the same continuation and equivalent continuations then yields equivalent computations as long as substitution commutes with plugging.

Lemma 5.14 (Substitution commutes with plugging). [CCNF:substPlug]
$(K[n])\{\sigma\} = (K\{\sigma\})[n\{\sigma\}]$, by induction on the structure of K.

Lemma 5.15 (Semantic equivalence of plugging). [Soundness:semK.plug]
If $\Gamma \mid \Delta \vDash K_1 \sim K_2 : B_1 \Rightarrow B_2$ *and* $\Gamma \mid \cdot \vDash n_1 \sim n_2 : B_1$, *then* $\Gamma \mid \Delta \vDash K_1[n_1] \sim K_2[n_2] : B_2$, *trivially after rewriting by Lemma 5.14.*

Corollary 5.16. [Soundness:semPlug] *If* $\Gamma \mid \Delta \vdash K : B_1 \Rightarrow B_2$ *and* $\Gamma \mid \cdot \vDash n_1 \sim n_2 : B_1$, *then* $\Gamma \mid \Delta \vDash K[n_1] \sim K[n_2] : B_2$, *using Lemma 5.15 and Theorem 5.13.*

5.4 Commuting conversion normalization is semantically equivalent to plugging

Now that we have both semantic equivalence of commuting conversions and of plugging, we use them together to show that plugging commutes with tail contexts.

Lemma 5.17 (Semantic equivalence of plugging let expressions). [Soundness:semKletin]
If $\Gamma \mid \Delta \vdash K : B_1 \Rightarrow B_2$ *and* $\Gamma \mid \cdot \vdash \mathsf{let}\ x \leftarrow n\ \mathsf{in}\ m : B_1$,
then $\Gamma \mid \Delta \vDash K[\mathsf{let}\ x \leftarrow n\ \mathsf{in}\ m] \sim \mathsf{let}\ x \leftarrow n\ \mathsf{in}\ K[m] : B_2$.

Proof. By induction on the typing derivation of K. In the K-nil case, this holds by Theorem 5.9. In all other cases, the left-hand side involves a subterm of the form $\mathsf{let}\ y \leftarrow (\mathsf{let}\ x \leftarrow n\ \mathsf{in}\ m)\ \mathsf{in}\ m'$, or $(\mathsf{let}\ x \leftarrow n\ \mathsf{in}\ m)\ v$, or $\mathsf{fst}\ (\mathsf{let}\ x \leftarrow n\ \mathsf{in}\ m)$, or $\mathsf{snd}\ (\mathsf{let}\ x \leftarrow n\ \mathsf{in}\ m)$, so we use derived rules let-let, let-app, let-fst, and let-snd respectively to commute them. For case K-let, we are done; for the remaining cases, we require Corollary 5.16 to commute under a plugged K, then use the induction hypothesis to transitively connect to the right-hand side. □

Lemma 5.18 (Semantic equivalence of plugging case expressions). [Soundness:semKcase]
If $\Gamma \mid \Delta \vdash K : B_1 \Rightarrow B_2$ *and* $\Gamma \mid \cdot \vdash \mathsf{case}\ v\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow m_1; \mathsf{inr}\ y \Rightarrow m_2\} : B_1$,
then $\Gamma \mid \Delta \vDash K[\mathsf{case}\ v\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow m_1; \mathsf{inr}\ y \Rightarrow m_2\}] \sim \mathsf{case}\ v\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow K[m_1]; \mathsf{inr}\ y \Rightarrow K[m_2]\} : B_2$.

Proof. By induction on the typing derivation of K, using Theorem 5.9 and Corollary 5.16 along with derived rules case-let, case-app, case-fst, and case-snd similarly to Lemma 5.17. □

These two lemmas suffice to prove the desired equivalence for a naïve translation that duplicates continuations instead of using join points. To handle join points, we need one more lemma that gives an equivalence between a translation that doesn't use a join point and one that does.

Lemma 5.19 (Jump equivalence of plugging). [Soundness:semKjoin]
If $\Gamma \mid \Delta \vdash K : B_1 \Rightarrow B_2$, $\Gamma \mid \cdot \vdash n : B_1$, *and* $K \equiv (\mathsf{let}\ x \leftarrow \square\ \mathsf{in}\ m) :: k_1 :: \ldots :: k_i$, *then* $\Gamma \mid \Delta \vDash K[n] \sim \mathsf{join}\ j\ x = m\ \mathsf{in}\ K'[n] : B_2$, *where* $K' \equiv (\mathsf{let}\ x \leftarrow \square\ \mathsf{in}\ \mathsf{jump}\ j\ x) :: k_1 :: \ldots :: k_i$.

Proof. By induction on the typing derivation of K. The rule K-let case uses Theorem 5.9 on the typing derivation of n to show it must evaluate to some return v; then both sides evaluate to $m\{x \mapsto v\}$, and we get the equivalence from Theorem 5.9 on the typing derivation of m. The remaining cases follow from the induction hypotheses. □

Lemma 5.20 (Jump equivalence of the translation). [Soundness:soundCCjoin]
Suppose that m *contains no join points or jumps. If* $\Gamma \mid \Delta \vdash K : B_1 \Rightarrow B_2$, $\Gamma \mid \Delta \vdash m : B_1$, *and* $K \equiv (\mathsf{let}\ x \leftarrow \square\ \mathsf{in}\ m') :: k_1 :: \ldots :: k_i$, *then* $\Gamma \mid \Delta \vDash [\![m]\!]K \sim \mathsf{join}\ j\ x = m'\ \mathsf{in}\ [\![m]\!]K' : B_2$, *where* $K' \equiv (\mathsf{let}\ x \leftarrow \square\ \mathsf{in}\ \mathsf{jump}\ j\ x) :: k_1 :: \ldots :: k_i$.

Proof. By induction on the typing derivation of m.

- **Cases** T-force, T-fun, T-ret, and T-prod: Hold directly from Lemma 5.19, using Lemma 4.5 to type translated subterms.
- **Cases** T-app, T-fst, and T-snd: Hold directly from the induction hypotheses.

- **Case** T-LET: The goal is to show that $[\![\text{let } y \leftarrow n \text{ in } m]\!]K$ is semantically equivalent to $\text{join } j\ x = m' \text{ in } [\![\text{let } y \leftarrow n \text{ in } m]\!]K'$, which holds by the following chain of equivalences.

$$
\begin{aligned}
&[\![\text{let } y \leftarrow n \text{ in } m]\!]K \\
&= [\![n]\!](\text{let } y \leftarrow \Box \text{ in } [\![m]\!]K) && (\textit{by definition}) \\
&\sim \text{join } j'\ y = [\![m]\!]K \text{ in } [\![n]\!](\text{let } y \leftarrow \Box \text{ in jump } j'\ y) && (\textit{by IH on } n) \\
&\sim \text{join } j'\ y = (\text{join } j\ x = m' \text{ in } [\![m]\!]K') \text{ in } [\![n]\!](\text{let } y \leftarrow \Box \text{ in jump } j'\ y) && (\textit{by IH on } m) \\
&\sim \text{join } j\ x = m' \text{ in join } j'\ y = [\![m]\!]K' \text{ in } [\![n]\!](\text{let } y \leftarrow \Box \text{ in jump } j'\ y) && (\textit{by JOIN-JOIN}) \\
&\sim \text{join } j\ x = m' \text{ in } [\![n]\!](\text{let } y \leftarrow \Box \text{ in } [\![m]\!]K') && (\textit{by IH on } n) \\
&= \text{join } j\ x = m' \text{ in } [\![\text{let } y \leftarrow n \text{ in } m]\!]K' && (\textit{by definition})
\end{aligned}
$$

- **Case** T-CASE: Let the translated computation be $\text{case } v \text{ of } \{\text{inl } y \Rightarrow m_1; \text{inr } y \Rightarrow m_2\}$. There are two subcases depending on what m' is.
  - If $m' \equiv \text{jump } j'\ w$, then by the translation, the left-hand side is
    $$\text{case } [\![v]\!] \text{ of } \{\text{inl } y \Rightarrow [\![m_1]\!]K; \text{inr } y \Rightarrow [\![m_2]\!]K\},$$
    and the right-hand side is
    $$\text{join } j\ x = m' \text{ in case } [\![v]\!] \text{ of } \{\text{inl } y \Rightarrow [\![m_1]\!]K'; \text{inr } y \Rightarrow [\![m_2]\!]K'\}.$$
    By Theorem 5.9, we know $[\![v]\!]$ reduces to either some $\text{inl } w$ or $\text{inr } w$. WLOG, supposing the former, it then suffices to show that $[\![m_1]\!]K\{y \mapsto w\}$ is semantically equivalent to $\text{join } j\ x = m' \text{ in } [\![m_1]\!]K'\{y \mapsto w\}$, which holds by the induction hypothesis on m_1.
  - Otherwise, both the left- and right-hand sides are
    $$\text{join } j\ x = m' \text{ in case } [\![v]\!] \text{ of } \{\text{inl } y \Rightarrow [\![m_1]\!]K'; \text{inr } y \Rightarrow [\![m_2]\!]K'\},$$
    which are equivalent by Theorem 5.9, using Lemma 4.5 for the translated subterms. □

Finally, we have all pieces to prove the main theorem: the translation under continuation K is semantically equivalent to plugging into K. The equivalence we want follows from instantiating by the empty continuation.

THEOREM 5.21. [Soundness:soundCC] *Suppose that v and m are terms containing no join points and jumps, and K_1, K_2 are continuations that may contain join points and jumps.*

- *If* $\Gamma \vdash v : A$ *then* $\Gamma \vDash v \sim [\![v]\!] : A$.
- *If* $\Gamma \mid \cdot \vdash m : B_1, \Gamma \mid \Delta \vdash K_1 : B_1 \Rightarrow B_2, \Gamma \mid \Delta \vdash K_2 : B_1 \Rightarrow B_2$, *and* $\Gamma \mid \Delta \vDash K_1 \sim K_2 : B_1 \Rightarrow B_2$, *then* $\Gamma \mid \Delta \vDash K_1[m] \sim [\![m]\!]K_2 : B_2$.

PROOF. By mutual induction on the typing derivations of v and m.

- **Case** T-VAR: Holds by logical equivalence of substitution maps.
- **Case** T-UNIT: Trivial.
- **Cases** T-FORCE, T-FUN, T-RET, and T-PROD: Hold by Lemma 5.15 and the induction hypotheses.
- **Cases** T-LEFT, T-RIGHT, T-THUNK, T-APP, T-FST, and T-SND: Hold by the induction hypotheses.
- **Case** T-LET: The goal is to show that $K[\text{let } x \leftarrow n \text{ in } m]$ is equivalent to $[\![\text{let } x \leftarrow n \text{ in } m]\!]K$, which holds by the following chain of equivalences.

$$
\begin{aligned}
K[\text{let } x \leftarrow n \text{ in } m] &\sim \text{let } x \leftarrow n \text{ in } K[m] && (\textit{by Lemma 5.17}) \\
&\sim \text{let } x \leftarrow n \text{ in } [\![m]\!]K = (\text{let } x \leftarrow \Box \text{ in } [\![m]\!]K)[n] && (\textit{by IH on } m) \\
&\sim [\![n]\!](\text{let } x \leftarrow \Box \text{ in } [\![m]\!]K) = [\![\text{let } x \leftarrow n \text{ in } m]\!]K && (\textit{by IH on } n)
\end{aligned}
$$

- **Case** T-CASE: The goal is to show that $K[\text{case } v \text{ of } \{\text{inl } y \Rightarrow m_1; \text{inr } y \Rightarrow m_2\}]$ is equivalent to $[\![\text{case } v \text{ of } \{\text{inl } y \Rightarrow m_1; \text{inr } y \Rightarrow m_2\}]\!]K$. By Lemma 5.18, the left-hand side is equivalent to

case v of $\{\mathsf{inl}\ y \Rightarrow K[m_1]; \mathsf{inr}\ y \Rightarrow K[m_2]\}$. Furthermore, by the induction hypotheses on the subterms, this is equivalent to case $[\![v]\!]$ of $\{\mathsf{inl}\ y \Rightarrow [\![m_1]\!]K; \mathsf{inr}\ y \Rightarrow [\![m_2]\!]K\}$. It then suffices to prove this equivalent to the right-hand side. There are two subcases depending on what K is.

- If $K \equiv \square :: k_1 :: \ldots :: k_i$ or $K \equiv (\mathsf{let}\ x \leftarrow \square\ \mathsf{in}\ \mathsf{jump}\ j\ v) :: k_1 :: \ldots :: k_i$, then the right-hand side is case $[\![v]\!]$ of $\{\mathsf{inl}\ y \Rightarrow [\![m_1]\!]K; \mathsf{inr}\ y \Rightarrow [\![m_2]\!]K\}$, and we are done.
- Otherwise, $K \equiv (\mathsf{let}\ x \leftarrow \square\ \mathsf{in}\ m') :: k_1 :: \ldots :: k_i$. Letting $K' \equiv (\mathsf{let}\ x \leftarrow \square\ \mathsf{in}\ \mathsf{jump}\ j\ x) :: k_1 :: \ldots :: k_i$, the right-hand side is

$$\mathsf{join}\ j\ x = m'\ \mathsf{in\ case}\ [\![v]\!]\ \mathsf{of}\ \{\mathsf{inl}\ y \Rightarrow [\![m_1]\!]K'; \mathsf{inr}\ y \Rightarrow [\![m_2]\!]K'\}.$$

  By Theorem 5.9, we know $[\![v]\!]$ reduces to either some $\mathsf{inl}\ w$ or $\mathsf{inr}\ w$. WLOG, supposing the former, it then suffices to show that $[\![m_1]\!]K\{y \mapsto w\}$ is equivalent to $\mathsf{join}\ j\ x = m'\ \mathsf{in}\ [\![m_1]\!]K'\{y \mapsto w\}$. Extending the substitution map with $y \mapsto w$ (noting that y is only free in m_1), this equivalence holds by Lemma 5.20. □

Corollary 5.22. [Soundness:soundCCnil] *Suppose m contains no join points and jumps. If $\Gamma \mid \cdot \vdash m : B$ then $\Gamma \mid \cdot \vDash m \sim [\![m]\!]\square : B$.*

At last we are able to prove Theorem 5.1, restated below, with one more minor lemma.

Lemma 5.23 (Equivalent ground values are equal). [Soundness:$\mathscr{V}$.ground] *If $(v, w) \in [\![T]\!]$ then $v = w$, proven by induction on T.*

Theorem 5.24. [Soundness:retGround] *Given m such that $\cdot \mid \cdot \vdash m : \mathsf{F}\ T$, if $m \Downarrow \mathsf{return}\ v$, then $[\![m]\!]\square \Downarrow \mathsf{return}\ v$.*

Proof. From Corollary 5.22, we have $\cdot \mid \cdot \vDash m \sim [\![m]\!]\square : \mathsf{F}\ T$. Instantiating with empty substitution maps and empty join stacks, we have $(m, [\![m]\!]\square) \in *FT$. By inversion, we know that there exists some $(w_1, w_2) \in T$ such that $m \Downarrow \mathsf{return}\ w_1$ and $[\![m]\!]\square \Downarrow \mathsf{return}\ w_2$. By Corollary 3.3, we have $v = w_1$, and by Lemma 5.23, we have $w_1 = w_2$. Therefore, $[\![m]\!]\square \Downarrow \mathsf{return}\ v$. □

6 Discussion

6.1 Case-of-case

In the lambda calculus with conditional expressions, the *case-of-case* transformation commutes a conditional whose scrutinee is another conditional. The naïve transformation directly moves the outer conditional into the branches of the inner, duplicating the code in its branches.

$$\begin{aligned}&\mathsf{if}\ (\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3)\ \mathsf{then}\ e_4\ \mathsf{else}\ e_5\\ &\Longrightarrow \mathsf{if}\ e_1\ \mathsf{then}\ (\mathsf{if}\ e_2\ \mathsf{then}\ e_4\ \mathsf{else}\ e_5)\ \mathsf{else}\ (\mathsf{if}\ e_3\ \mathsf{then}\ e_4\ \mathsf{else}\ e_5)\end{aligned} \quad [8]$$

Let-binding the outer branches avoids this duplication. Notably, doing so doesn't touch the direct scrutinization of e_2 and e_3, so if they are literal true or false values, then those branches can be inlined to just x or y.

$$\begin{aligned}&\mathsf{if}\ (\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3)\ \mathsf{then}\ e_4\ \mathsf{else}\ e_5\\ &\Longrightarrow \mathsf{let}\ x = e_4\ \mathsf{in\ let}\ y = e_5\ \mathsf{in\ if}\ e_1\ \mathsf{then}\ (\mathsf{if}\ e_2\ \mathsf{then}\ x\ \mathsf{else}\ y)\ \mathsf{else}\ (\mathsf{if}\ e_3\ \mathsf{then}\ x\ \mathsf{else}\ y)\end{aligned} \quad [9]$$

If we translate the left-hand lambda expression to a CBPV term, both CBV and CBN translations yield a term of the following shape.

$$\mathsf{let}\ y \leftarrow (\mathsf{let}\ x \leftarrow m_1\ \mathsf{in\ if}\ x\ \mathsf{then}\ m_2\ \mathsf{else}\ m_3)\ \mathsf{in\ if}\ y\ \mathsf{then}\ m_4\ \mathsf{else}\ m_5 \quad [10]$$

CC-normalizing this term moves the outer let binding into the branches of the conditional on x, with jumps to avoid duplication.

$$\mathsf{let}\ x \leftarrow m_1\ \mathsf{in\ join}\ j\ y = \mathsf{if}\ y\ \mathsf{then}\ m_4\ \mathsf{else}\ m_5\ \mathsf{in}$$

$$\text{if } x \text{ then } (\text{let } y \leftarrow m_2 \text{ in jump } j\ y) \text{ else } (\text{let } y \leftarrow m_3 \text{ in jump } j\ y) \qquad [11]$$

Unfortunately, even if m_2 is the computation return true, inlining it will only reduce the branch down to jump j true, whose evaluation will still need to jump to the conditional in the join point that could otherwise have been inlined away statically. What we would like is to create separate join points for m_4 and m_5 and jump to them individually to permit the kind of inlining above, similar to the right-hand side of Equation 9.

$$\begin{aligned}&\text{let } x \leftarrow m_1 \text{ in join } j_4 = m_4 \text{ in join } j_5 = m_5 \text{ in}\\ &\quad \text{if } x \text{ then } (\text{let } y \leftarrow m_2 \text{ in if } y \text{ then jump } j_4 \text{ else jump } j_5)\\ &\quad\quad \text{else } (\text{let } y \leftarrow m_3 \text{ in if } y \text{ then jump } j_4 \text{ else jump } j_5)\end{aligned} \qquad [12]$$

Therefore, we prove a semantic equivalence between Equation 11 and Equation 12, generalizing to case expressions. Consequently, a transformation from the former to the latter is valid, and we recover the desired case-of-case transformation.

Lemma 6.1 (Case-of-case). [`Commutation.caseCase`] *Suppose the following hold:*

- $\Gamma \vdash v : A_3 + A_4$
- $\Gamma, y_1 : A_1 \mid \Delta \vdash m_1 : B$
- $\Gamma, y_2 : A_2 \mid \Delta \vdash m_2 : B$
- $\Gamma, y_3 : A_3 \mid \cdot \vdash m_3 : \mathsf{F}\,(A_1 + A_2)$
- $\Gamma, y_4 : A_4 \mid \cdot \vdash m_4 : \mathsf{F}\,(A_1 + A_2)$

Then under contexts Γ, Δ*, the computation*

$$\begin{aligned}&\text{join } j\ x = \text{case } x \text{ of } \{\text{inl } y_1 \Rightarrow m_1; \text{inr } y_2 \Rightarrow m_2\} \text{ in}\\ &\quad \text{case } v \text{ of } \{\text{inl } y_3 \Rightarrow (\text{let } x \leftarrow m_3 \text{ in jump } j\ x); \text{inr } y_4 \Rightarrow (\text{let } x \leftarrow m_3 \text{ in jump } j\ x)\}\end{aligned}$$

is semantically equivalent at type B *to the computation*

$$\begin{aligned}&\text{join } j_1\ y_1 = m_1 \text{ in join } j_2\ y_2 = m_2 \text{ in}\\ &\quad \text{case } v \text{ of } \{\text{inl } y_3 \Rightarrow (\text{let } x \leftarrow m_3 \text{ in case } x \text{ of } \{\text{inl } y_1 \Rightarrow \text{jump } j_1\ y_1; \text{inr } y_2 \Rightarrow \text{jump } j_2\}\ y_2);\\ &\qquad\qquad \text{inr } y_4 \Rightarrow (\text{let } x \leftarrow m_4 \text{ in case } x \text{ of } \{\text{inl } y_1 \Rightarrow \text{jump } j_1\ y_1; \text{inr } y_2 \Rightarrow \text{jump } j_2\}\ y_2)\}\end{aligned}$$

This transformation applies generally to case-of-case-of-case and so forth. The translation to CBPV gives join points that contain join points that contain case expressions and so forth. Unnesting these join points with join-join followed by Lemma 6.1 yields the desired transformation.

6.2 Inlining and CCNF

Recall that new opportunities for inlining revealed by CC-normalization preserve CCNF, since they only occur in tail positions. However, subsequent inlinings that force direct thunks may violate CCNF. Consider the following sequence of a commuting conversion, followed by an inlining that reduces a function, followed by an inlining that forces a thunk.

$$\begin{aligned}&(\text{let } x \leftarrow n_1 \text{ in } (\lambda y'.\, \text{let } y \leftarrow y'! \text{ in } m_1))\,\{\text{let } z \leftarrow n_2 \text{ in } m_2\}\\ &\Longrightarrow \text{let } x \leftarrow n_1 \text{ in } ((\lambda y'.\, \text{let } y \leftarrow y'! \text{ in } m_1)\,\{\text{let } z \leftarrow n_2 \text{ in } m_2\}) && (\textit{commute})\\ &\Longrightarrow \text{let } x \leftarrow n_1 \text{ in } (\text{let } y \leftarrow \{\text{let } z \leftarrow n_2 \text{ in } m_2\}! \text{ in } m_1) && (\text{E-app})\\ &\Longrightarrow \text{let } x \leftarrow n_1 \text{ in } (\text{let } y \leftarrow (\text{let } z \leftarrow n_2 \text{ in } m_2) \text{ in } m_1) && (\text{E-force})\end{aligned}$$

The resulting term after commuting and inlining once is still in CCNF. But if we force the thunk, we end up with a nested let expression, which isn't in CCNF. Therefore, renormalization may be required only after forcing thunks that appear in n positions as a result of inlining.

6.3 Commuting constructors

Our notion of commuting conversions only involves elimination forms, as they make up evaluation contexts E and tail contexts L. Other work [Forster et al. 2019; Levy 2003] also consider equations which commuting elimination forms with introduction forms, specifically computation constructors in tail position, listed below. In particular, the first equation is part of Levy's *sequencing laws*.

$$\mathsf{let}\ x \leftarrow n\ \mathsf{in}\ \lambda y.\, m \iff \lambda y.\, \mathsf{let}\ x \leftarrow n\ \mathsf{in}\ m$$

$$\mathsf{case}\ v\ \mathsf{of}\ \{\mathsf{inl}\ y \Rightarrow \lambda x.\, m_1; \mathsf{inr}\ z \Rightarrow \lambda x.\, m_2\} \iff \lambda x.\, \mathsf{case}\ v\ \mathsf{of}\ \{\mathsf{inl}\ y \Rightarrow m_1; \mathsf{inr}\ z \Rightarrow m_2\}$$

$$\mathsf{let}\ x \leftarrow n\ \mathsf{in}\ \langle m_1, m_2\rangle \iff \langle \mathsf{let}\ x \leftarrow n\ \mathsf{in}\ m_1, \mathsf{let}\ x \leftarrow n\ \mathsf{in}\ m_2\rangle$$

$$\mathsf{case}\ v\ \mathsf{of}\ \{\mathsf{inl}\ y \Rightarrow \langle m_1, m_2\rangle; \mathsf{inr}\ z \Rightarrow \langle m_3, m_4\rangle\} \iff$$
$$\langle \mathsf{case}\ v\ \mathsf{of}\ \{\mathsf{inl}\ y \Rightarrow m_1; \mathsf{inr}\ z \Rightarrow m_3\}, \mathsf{case}\ v\ \mathsf{of}\ \{\mathsf{inl}\ y \Rightarrow m_2; \mathsf{inr}\ z \Rightarrow m_4\}\rangle$$

We don't consider these because it's unclear which direction to pick and what benefits they provide. Going left to right, the case equations only apply when the branches are both exactly an introduction form, and these naïve pair equations duplicate code, so typing and evaluation would need to be extended to permit creating join points from the components and jumping from within pairs. Going right to left, the pair equations only apply when the projections both eliminate exactly the same thing. Either way, they are sensitive to slight syntactic variation, *e.g.* permuted let bindings. Furthermore, neither direction appears to reveal optimization opportunities.

6.4 Extensions and future work

Effects. Like Forster et al. [2019], the CBPV we consider doesn't include effects. However, by inspection, we can see that commuting conversions are effect safe. For instance, considering $\mathsf{let}\ y \leftarrow (\mathsf{let}\ x \leftarrow n_1\ \mathsf{in}\ n_2)\ \mathsf{in}\ m$, if n_1, n_2, and m contained effects, they would be executed in that very order according to the evaluation semantics both before and after commutation.

Not all semantically equivalent computations preserve effect order. Given n_1 and n_2 where neither x nor y are free, $\mathsf{let}\ x \leftarrow n_1\ \mathsf{in}\ \mathsf{let}\ y \leftarrow n_2\ \mathsf{in}\ m$ is semantically equivalent to $\mathsf{let}\ y \leftarrow n_2\ \mathsf{in}\ \mathsf{let}\ x \leftarrow n_1\ \mathsf{in}\ m$, but would swap the order of effects in n_1 and n_2. We believe that CC-normalization doesn't perform extraneous reordering transformations, but rigorously proving it effect safe requires incorporating effects into the logical relation.

Stack usage. Commuting conversions optimize stack usage. For the CBV lambda calculus, where the A-reductions of ANF are commuting conversions, Bowman [2024] views ANF as the A-normalization of monadic normal form (MNF), the syntactic form of the monadic meta-language by Moggi [1991] that binds intermediate computations. He shows that first taking A-reduction steps from MNF to ANF optimizes stack usage in comparison to evaluating the original MNF term under stack machine semantics.

Similarly, we can see that our CBPV commuting conversions may also reduce stack usage. For instance, considering $\mathsf{let}\ y \leftarrow (\mathsf{let}\ x \leftarrow n_1\ \mathsf{in}\ n_2)\ \mathsf{in}\ m$, stack machine evaluation would first push $\mathsf{let}\ y \leftarrow \square\ \mathsf{in}\ m$ onto the stack, followed by $\mathsf{let}\ x \leftarrow \square\ \mathsf{in}\ n_2$, before evaluating n_1 and popping these two stack frames. In contrast, evaluating $\mathsf{let}\ x \leftarrow n_1\ \mathsf{in}\ \mathsf{let}\ y \leftarrow n_2\ \mathsf{in}\ m$ pushes $\mathsf{let}\ x \leftarrow \square\ \mathsf{in}\ \mathsf{let}\ y \leftarrow n_2\ \mathsf{in}\ m$ onto the stack, evaluates n, pops this stack frame, then pushes $\mathsf{let}\ y \leftarrow \square\ \mathsf{in}\ m$ next, using one fewer total stack frame.

Because our translation is a single-pass transformation that uses a continuation rather than a sequence of commuting conversions, which traverses the term each time, showing that CC-normalized computations optimize stack usage is nontrivial. It may require going through an intermediate big-step semantics to track maximum stack usage, similar to how Chen et al. [2025] use big-step to track space and time usage in their cost model of CBPV.

Compilation to assembly. One way to reap the benefits of the way commuting conversions unnest computations is to compile onwards to a lower-level assembly-like language. A potential compilation target is the lower-level "linearized CBPV" proposed by New [2019], which is a stack machine language with explicit push/pop and jump operations. We conjecture that CC-normalization prior to compilation to this language would optimize stack usage, and that our jumps can be compiled directly to its jumps.

7 Related Work

Forster et al. [2019] mechanize in Rocq a vast amount of metatheory for call-by-push-value, including normalization and observational equivalence. Our work builds on their design of logical equivalence between terms. They show a number of commuting conversions as semantic equivalences, namely commuting let-bound let expressions (Lemma 8.4, Equation 5) and function application of let expressions (Lemma 8.6, Equation 1), but are not comprehensive in listing all possible commuting conversions systematically as we have done.

Our join points and their typing judgements with a separate join point typing context are inspired by Maurer et al. [2017], who add join points and jumps to System F, and implement their system in the Haskell compiler GHC. We simplify the addition by only allowing jumps in tail position, while they permit jumps in evaluation contexts. They perform their optimizations equation by equation, which yields intermediate steps that require this permissiveness, such as the following.

$$\begin{aligned}(\mathsf{join}\ j\ x = m\ \mathsf{in}\ \mathsf{jump}\ j\ v)\ w &\Longrightarrow \mathsf{join}\ j\ x = m\ w\ \mathsf{in}\ (\mathsf{jump}\ j\ v)\ w \\ &\Longrightarrow \mathsf{join}\ j\ x = m\ w\ \mathsf{in}\ \mathsf{jump}\ j\ v\end{aligned}$$

Because we present instead a single-pass algorithm, we never produce such intermediate steps that need to be typeable, so we are able to eliminate them from our syntax outright. Doing so also simplifies typing: their jumps need to be typeable with arbitrary types, which they implement using type polymorphism, while our jumps are always in tail position and fixed to the return type of the join point they jump to.

Their work on join points is in contrast to earlier work by Kennedy [2007], who argues for CPS instead of direct-style using second-class continuations, and from whom we borrow the terminology *CC-normalization*. Cong et al. [2019] combine ideas from both using control operators that bind second-class continuations, using them as join points. Their system allows for both direct style and continuation-passing style optimizations; this is not directly applicable to CBPV, which is already in direct style by virtue of binding intermediate computations. However, the corresponding dual of CBPV is stack-passing style [New 2023], and it would be interesting to see whether a similar technique could be applied. The Zydeco project is ongoing work implementing a compiler from CBPV to stack-passing style [Jiang et al. 2025].

8 Conclusion

In this paper, we looked at commuting conversions for call-by-push-value, which are syntactic transformations that preserve evaluation. By commuting evaluation contexts into tail positions, we unnest computations and expose inlining opportunities. We have identified a commuting conversion normal form for CBPV and presented a single-pass transformation into CCNF. To avoid code duplication without incurring additional closures, we used a separate join point construct that avoids creating new thunks. We have shown that the translation preserves not just typing but also evaluation, using a logical equivalence to prove that the translation of a term is equivalent to itself. Our results are entirely mechanized in Lean 4.

9 Data-Availability Statement

We will submit an artifact for evaluation consisting of our Lean 4 proof development, provided as supplementary materials for the paper submission.

References

William J. Bowman. 2024. A Low-Level Look at A-Normal Form. *Proc. ACM Program. Lang.* 8, OOPSLA2 (Oct. 2024), 165–191. doi:10.1145/3689717

Zhuo Chen, Johannes Åman Pohjola, and Christine Rizkallah. 2025. A Verified Cost Model for Call-by-Push-Value. In *Interactive Theorem Proving* (Reykjavik, Iceland) *(ITP 2025)*. Springer International Publishing, Cham, Switzerland, Article 7, 19 pages. doi:10.4230/LIPIcs.ITP.2025.7

Youyou Cong, Leo Osvald, Grégory M. Essertel, and Tiark Rompf. 2019. Compiling with continuations, or without? whatever. *Proc. ACM Program. Lang.* 3, ICFP, Article 79 (July 2019), 28 pages. doi:10.1145/3341643

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *International Conference on Automated Deduction (Lecture Notes in Computer Science, Vol. 9195)*. Springer International Publishing, Cham, Switzerland, 378–388. doi:10.1007/978-3-319-21401-6_26

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The essence of compiling with continuations. In *ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, Albuquerque, NY, USA, 237–247. doi:10.1145/173262.155113

Yannick Forster, Steven Schäfer, Simon Spies, and Kathrin Stark. 2019. Call-by-push-value in Coq: operational, equational, and denotational theory. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Cascais, Portugal) *(CPP 2019)*. Association for Computing Machinery, New York, NY, USA, 1180–131. doi:10.1145/3293880.3294097

Dmitri Garbuzov, William Mansky, Christine Rizkallah, and Steve Zdancewic. 2018. Structural Operational Semantics for Control Flow Graph Machines. https://arxiv.org/abs/1805.05400

Yuchen Jiang, Max S. New, Tingting Ding, Runze Xue, Yuxuan Xia, and Nathan Varner. 2025. *zydeco-lang/zydeco: v0.2.2*. zydeco-lang. doi:10.5281/zenodo.15756916

Andrew Kennedy. 2007. Compiling with continuations, continued. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming* (Freiburg, Germany) *(ICFP '07)*. Association for Computing Machinery, New York, NY, USA, 177–190. doi:10.1145/1291151.1291179

Paul Blain Levy. 2003. *Call-By-Push-Value: A Functional/Imperative Synthesis*. Springer Dordrecht, Dordrecht, Netherlands. doi:10.1007/978-94-007-0954-6

Luke Maurer, Zena Ariola, Paul Downen, and Simon Peyton Jones. 2017. Compiling without continuations. In *ACM Conference on Programming Languages Design and Implementation (PLDI'17)*. Association for Computing Machinery, New York, NY, USA, 482–494. doi:10.1145/3062341.3062380

Eugenio Moggi. 1991. Notions of computation and monads. *Information and Computation* 93, 1 (1991), 55–92. doi:10.1016/0890-5401(91)90052-4

Greg Morrisett, Karl Crary, Neal Glew, and David Walker. 2002. Stack-based typed assembly language. *Journal of Functional Programming* 12, 1 (2002), 43–88. doi:10.1017/S0956796801004178

Max S. New. 2019. From Call-by-push-value to Stack-Based TAL? https://maxsnew.com/docs/cbpv-stal-lola-2019.pdf

Max S. New. 2023. Compiling with Call-by-push-value. https://coalg.org/calco-mfps-2023/slides/compiling-with-cbpv-1.pdf

Christine Rizkallah, Dmitri Garbuzov, and Steve Zdancewic. 2018. A Formal Equational Theory for Call-By-Push-Value. In *Interactive Theorem Proving*, Jeremy Avigad and Assia Mahboubi (Eds.). Springer International Publishing, Cham, Switzerland, 523–541. doi:10.1007/978-3-319-94821-8_31

Zachary J. Sullivan, Paul Downen, and Zena M. Ariola. 2023. Closure Conversion in Little Pieces. In *Proceedings of the 25th International Symposium on Principles and Practice of Declarative Programming* (Lisboa, Portugal) *(PPDP 2023)*. Association for Computing Machinery, New York, NY, USA, Article 10, 13 pages. doi:10.1145/3610612.3610622