Compiling with Continuations, or without? Whatever.

YOUYOU CONG, Tokyo Institute of Technology, Japan
LEO OSVALD, Purdue University, USA
GRÉGORY ESSERTEL, Purdue University, USA
TIARK ROMPF, Purdue University, USA

What makes a good compiler IR? In the context of functional languages, there has been an extensive debate on the advantages and disadvantages of continuation-passing-style (CPS). The consensus seems to be that some form of explicit continuations is necessary to model jumps in a functional style, but that they should have a 2nd-class status, separate from regular functions, to ensure efficient code generation. Building on this observation, a recent study from PLDI 2017 proposed a direct-style IR with explicit join points, which essentially represent local continuations, i.e., functions that do not return or escape. While this IR can work well in practice, as evidenced by the implementation of join points in the Glasgow Haskell Compiler (GHC), there still seems to be room for improvement, especially with regard to the way continuations are handled in the course of optimization.

In this paper, we contribute to the CPS debate by developing a novel IR with the following features. First, we integrate a control operator that resembles Felleisen's $\mathcal{C}$, eliminating certain redundant rewrites observed in the previous study. Second, we treat the non-returning and non-escaping aspects of continuations separately, allowing efficient compilation of well-behaved functions defined by the user. Third, we define a selective CPS translation of our IR, which erases control operators while preserving the meaning and typing of programs. These features enable optimizations in both direct style and full CPS, as well as in any intermediate style with selectively exposed continuations. Thus, we change the spectrum of available options from "CPS yes or no" to "as much or as little CPS as you want, when you want it".

CCS Concepts: • **Hardware → Hardware description languages and compilation**; • **Theory of computation → Control primitives**; *Functional constructs.*

Additional Key Words and Phrases: control operators, CPS translation, compiler intermediate languages

ACM Reference Format:
Youyou Cong, Leo Osvald, Grégory Essertel, and Tiark Rompf. 2019. Compiling with Continuations, or without? Whatever.. *Proc. ACM Program. Lang.* 3, ICFP, Article 79 (August 2019), 29 pages. https://doi.org/10.1145/3341643

1 INTRODUCTION

The year 2018 marked the 40th anniversary of Steele's Rabbit compiler [Steele 1978], the first compiler for Scheme. Rabbit was influential in demonstrating that a lexically-scoped dialect of LISP could be compiled efficiently. In particular, it served as an existence proof that the λ-calculus, with its scoping rules as in the mathematical formalism, is a useful foundation for practical programming languages. Rabbit also pioneered the use of a *continuation-passing style (CPS)* language as an intermediate representation. In a CPS IR, every computation receives a continuation representing what to do next. This makes both control flow and evaluation order explicit in the IR, allowing us to produce machine code without much effort. On the other hand, the presence of continuations and inflexibility of evaluation order can also be an obstacle to optimization. Thus, the birth of Rabbit marked the beginning of the great "CPS or not CPS" debate.

2475-1421/2019/8-ART79
https://doi.org/10.1145/3341643

CPS vs. Direct Style. On the CPS side, Rabbit inspired several compilers for the dialects of Scheme and SML languages, including Orbit [Kranz et al. 1986a] and the SML/NJ compiler [Appel 1992]. In these compilers, many optimizations can be described as a sequence of full β and η steps, which are always *sound* and often come with *shrinking guarantees*. While this is a direct benefit obtained from explicit continuations, they may as well make it harder to recover specific information from the source program. An instance of this is the uniform representation of ordinary functions and continuations as jumps. Since the latter are guaranteed not to escape, a compiler must avoid allocating closure records for them.

To address the downsides of CPS, Flanagan et al. [1993] proposed to replace CPS intermediate languages with Administrative Normal Form (ANF). ANF makes control flow explicit by let-binding intermediate results, rather than λ-binding them. Since an ANF-based IR remains in direct style, we can more easily identify parts of the program that are subject to optimization. Nevertheless, ANF suffers from two problems that would not arise in CPS. First, ANF is not closed under reduction, and hence requires an additional re-normalization step for subsequent optimizations. Second, ANF lacks a facility for modeling jumps, which is crucial for converting recursive functions into loops, and for representing *join points* that arise in case-like constructs.

As a remedy, Kennedy [2007] proposed to go back to CPS, but in a very specific way. The idea is to make a syntactic distinction between regular functions and continuations, and enforce continuations to behave in a 2nd-class manner. This allows us to compile continuations as jumps, and use them as a representation of join points. However, if the CPS IR is the only one on which optimizations are performed, a number of rewrites — such as common subexpression elimination, code motion, and other high-level rewrites defined by the user — would be much harder than in direct style. The problem is particularly severe in non-strict languages like Haskell, as their CPS semantics is highly involved [Okasaki et al. 1994].

To address this problem, Maurer et al. [2017] proposed to go back to direct style, but with a special form of functions for representing join points. Calls to join points are jumps, and join points "return" to their scope of declaration. Thus, there is no need to CPS-translate the source program if the purpose is to perform commuting conversions, contification, etc.; these can all be performed on their direct-style IR extended with join points.

Is This the End of Story? Maurer et al.'s IR can work well in practice, as evidenced by a corresponding implementation in the Glasgow Haskell Compiler (GHC). But in our view, informed by an attempt to re-implement the join point construct and corresponding transformation rules in the MiniScala compiler (which is used for teaching compiler construction at one of our institutions), it suffers from two kinds of inflexibility. First, the IR does not give the optimizer enough control over evaluation contexts. This results in a kind of redundancy when performing commuting conversions, and, viewed from a different angle, breaks the independence among individual rewrite rules. Second, the IR only supports optimizations in direct style. While Maurer et al. conjectured that the fixed strategy should not limit optimization opportunities, our experiment shows that a subsequent CPS pass may have a positive impact on the overall performance of the compiler.

Our Proposal. In this paper, we would like to settle the "CPS or not CPS" debate once and for all with a decisive "it depends!"

Instead of looking for a universal truth, we believe that we must provide more nuanced recommendations for compiler writers. In that sense, "CPS or not CPS" is asking the wrong question. Requirements for compilers differ wildly, and in particular, we have to consider whether we are compiling a low-level vs. high-level language, or, viewed another way, whether we are building a compiler front-end or back-end. At some point in the pipeline, a form of CPS translation is inevitable to turn functional code into basic blocks and jumps. This may well be the last step, directly emitting

assembly code, but many functional language compilers target a low-level backend that operates on a control flow graph (CFG) based IR on which additional passes of dataflow analysis (such as register allocation) are run. It is widely appreciated that CFG-based IRs in SSA-form are isomorphic to CPS [Appel 1998; Kelsey 1995]. Even in the setting of Maurer et al. [2017], who reject CPS as high-level IR, GHC compiles to `C--` or LLVM, effectively performing CPS translation as part of this lowering. Hence, for a particular optimization, "before or after CPS" seems to be a more appropriate question to ask, and the more choice an IR provides in this regard, the better job a compiler does.

We argue that the essence of "Compiling without Continuations" [Maurer et al. 2017] is not to get rid of continuations, but to be selective in *which* continuations are made explicit in the IR of a *high-level* compiler. With this key realization, we can set the whole enterprise on a firmer footing. There is no need to introduce a separate language construct to model join points; we can simply use a well-understood facility, namely *control operators*, to expose some selected continuations. That is, we want something like `call/cc`, but restricted to introduce Kennedy-style 2nd-class continuations.

Of course, `call/cc` is not the only possible control operator, so having identified the need for *a* control operator as a building block we can consider some alternatives. Semantically, Maurer-style join points behave like `call/cc` (as the paper notes in passing [Maurer et al. 2017]), in that a program silently continues after the `call/cc` block even if the continuation is not invoked explicitly. We show that, a slightly different control operator, closer to the one proposed by Felleisen et al. [1987], is more appropriate for a compiler IR. In particular, the absence of the silent fall-through property helps us avoid redundant optimization steps we mentioned earlier.

As is typical in calculi with control operators, our IR has a type- and meaning-preserving CPS translation, which converts programs into a form suitable for low-level optimizations. This allows to represent the pre- and post-CPS versions of the same program in a single IR. That is to say, we can optimize programs with no continuation (direct style), or all continuations (full CPS), or any number of continuations exposed via the control operator.

What remains to be discussed is how to distinguish continuations from ordinary functions. In our IR, we separate the *non-returning* and *non-escaping* aspects of continuations, and build a type system that keeps track of each expression's behavior. The design principle enables modelling non-escaping values that are not continuations. As a result, we can avoid unnecessary allocation of user-defined functions, eliminating one main source of inefficiency in functional programming languages. Furthermore, non-escaping values have many other practical uses, such as representation of temporary access tokens and formation of co-effect systems [Osvald et al. 2016].

Contributions. To sum up, this paper makes the following contributions:

- We present $\lambda_{\perp}^{1/2}$, a λ-calculus-based IR extended with a control operator (Section 3). In our IR, we describe a function's behavior using value / $\perp$ types and 1st / 2nd-class status. In particular, we ensure that continuations are used as jumps, and that they never escape their defining scope.
- We define a CPS translation of $\lambda_{\perp}^{1/2}$, which targets a restricted subset of the source language (Section 4). The translation is *selective* in that it only manipulates parts of the program where continuations are implicit. We then show that CPS programs follow a stack-based call discipline: function calls are always accompanied by a shrinking event of the stack.
- We show that our control operator simplifies optimization steps for case-like constructs, thanks to its flexible semantics (Section 2). We also provide examples illustrating the benefit of treating non-returning and non-escaping behaviors as independent properties (Section 6).
- We evaluate the proposed approach through two different implementations of our IR (Section 7). By running various benchmarks, we confirm that 2nd-class functions and continuations lead to faster execution and better memory usage.

2 MOTIVATION AND KEY IDEAS

In this section, we show how we use control operators to perform optimizations, highlighting the similarities and differences with Maurer et al.'s join points.

2.1 Representing Join Points

Let us begin with a simple program that uses a short-circuit boolean operator:

```
// source expression
if (e1 && e2) then e3 else e4
```

When we desugar the && operator, we obtain the following program:

```
// after desugaring
if (if e1 then e2 else false) then e3 else e4
```

Notice that the desugaring resulted in nested `if` expressions, which cannot be simplified if `e1` does not statically reduce to a boolean value. On the other hand, if we reorganize the conditional in an appropriate way, we will obtain an opportunity for simplification:

```
// commuting conversion
if e1 then (if e2 then e3 else e4)
      else (if false then e3 else e4)
```

The flattening step, called a *commuting conversion*, moved the outer `if` to the branches of the inner `if`. This exposes a new redex `if false ...`, allowing us to simplify the program to:

```
// simplification
if e1 then (if e2 then e3 else e4) else e4
```

The above program is however not optimal, in that it has duplicated occurrences of the branch `e4`. This motivates us to name the branch as follows:

```
// join points as regular functions
def j4() = e4
if e1 then (if e2 then e3 else j4()) else j4()
```

The name `j4` represents a *join point*, i.e., the place where the execution of branches joins up. Here, the join point is defined as an ordinary function, which will be heap-allocated as a closure. The allocation is however unnecessary, because the function does not escape its defining scope. Then, how can we avoid such redundant allocation?

Past Solution 1: CPS. To address the above problem, Kennedy [2007] proposed to perform optimizations in a CPS IR, where continuations are *2nd-class* citizens that never escape. In the case of our specific example, we perform optimizations after translating the source program to the following one:

```
// continuation-passing style
defcont k(x)   = ...
defcont j3()   = k(e3)
defcont j4() = k(e4)
defcont k2(x2) = if x2 then j3() else j4()
defcont j2()   = k2(e2)
defcont k1(x1) = if x1 then j2() else j4()
defcont j1()   = k1(e1)
```

The program uses a special binder `defcont` for continuations. This prevents us from capturing them in a closure or storing them in a mutable reference.

Past Solution 2: Direct Style + Explicit Join Points. While Kennedy solved the issue with allocation, the solution is still not satisfactory, because CPS makes certain class of optimizations harder to apply. For instance, in direct style, it is easy to find a nested application like `xs.map(f).map(g)`, and turn it into a single application `xs.map(g∘f)`. If we translate the original application into CPS, we would need to carefully analyze the program and spot the place to apply such rewrites.

For this reason, Maurer et al. [2017] suggest that we should work in a direct-style IR, where we have built-in facilities for representing join points. If we take this approach, our example will be rewritten to the following program:

```
// join points as special constructs
defjoin j4() = e4
if e1 then (if e2 then e3 else j4()) else j4()
```

The `defjoin` binding form introduces a join point without allocating it on the heap. This construct has been incorporated into the Glasgow Haskell Compiler (GHC) [Peyton Jones and Marlow 2002], and proven effective in reducing expensive allocations.

Our Solution: Direct Style + Control Operator. Our approach can be thought of as a combination of the two solutions discussed so far. The idea is to define join points by means of 2nd-class continuations, just like Kennedy does, while allowing optimizations in direct style, as in Maurer et al.'s IR. What makes this possible is a *control operator*, and a type system that can express non-returning and non-escaping values. For instance, the example we have been discussing is represented as follows:

```
// control operator
𝒞(k =>
  defcont j4() = k(e4)
  if e1 then (if e2 then k(e3) else j4()) else j4())
```

Operationally, the $\mathscr{C}$ operator captures and clears the surrounding context, and binds the variable `k` to a function representing the captured context. Type-wise, we classify `k` as a 2nd-class value, and assign it a type of the form $T \to \bot$, where $\bot$ represents the non-returning nature of `k`. Since the type system does not allow 2nd-class values to escape, we can safely allocate continuations on the stack, and by appropriately restricting occurrences of $\bot$ in the typing rules, we can guarantee that continuations are only used as jumps.

As the reader might have imagined, it is the optimizer's job to handle continuations via the $\mathscr{C}$ operator. In general, the optimizer inserts $\mathscr{C}$ whenever a continuation needs to be made explicit. To decide where to insert $\mathscr{C}$, we can reuse Maurer et al.'s techniques for inserting join point constructs, including the *contification* transformation [Kennedy 2007].

2.2 Optimizing and Preserving Join Points

In simple cases like the above example, the $\mathscr{C}$-based approach works almost the same way as Maurer et al.'s. Now, we look at a more complex example discussed by Maurer et al., where our IR allows for a simpler and more resilient optimization process. Consider the following (source or intermediate) program, which already has a join point inserted, but in a nested position[1]:

```
// source expression (nested pattern match)
(defjoin j(x) = BIG
 v match { case A => j(1); case B => j(2); case C => true })
match { case true  => false; case false => true }
```

If we naïvely apply a commuting conversion, the program will be rewritten to the following one:

[1]For readers unfamiliar with the Scala syntax, `e match { case pi => ei }` means matching `e` against patterns `pi`.

```
// undersired transformation (join point destroyed)
defjoin j(x) = BIG
v match {
  case A => j(1) match { case true => false; case false => true }
  case B => j(2) match { case true => false; case false => true }
  case C => false }
```

The resulting program is problematic in two ways. First, `j` is not tail-called, hence it cannot be compiled as a join point. Second, the pattern matching constructs in the `A` and `B` branches scrutinize an application of `j`, which means no further simplification is available for these constructs.

Past Solution. Maurer et al. claim that the source program should be transformed as follows:

```
// desired transformation (join point preserved)
defjoin j(x) = BIG match { case true => false; case false => true }
v match {
  case A => j(1)
  case B => j(2)
  case C => false }
```

Here, the join point is preserved, and the `match` construct inspects `BIG`, which might be a data constructor or another `match` expression. Maurer et al. arrive at this desired program in four steps, using the rewrite rules listed below:

$$E[\text{defjoin } j(x) = e1;\ e2] = \text{defjoin } j(x) = E[e1];\ E[e2] \qquad \text{(jfloat)}$$

$$E[e\ \text{match}\{p_i \to e_i\}] = e\ \text{match}\{p_i \to E[e_i]\} \qquad \text{(casefloat)}$$

$$E[j(e)] = j(e) \qquad \text{(abort)}$$

First, we use (jfloat) to move the outer `match` to the right-hand side and the body of `defjoin`.

```
defjoin j(x) = BIG match { case true  => false; case false => true }
(v match { case A => j(1); case B => j(2); case C => true })
match { case true => false; case false => true }
```

Next, we use (casefloat) to move the outer `match` to the branches of the inner `match`.

```
defjoin j(x) = BIG match { case true  => false; case false => true }
v match { case A => j(1) match { case true  => false; case false => true }
          case B => j(2) match { case true  => false; case false => true }
          case C => true match { case true  => false; case false => true }}
```

We then use (abort) to discard the context surrounding jumps.

```
defjoin j(x) = BIG match { case true  => false; case false => true }
v match { case A => j(1)
          case B => j(2)
          case C => true match { case true => false; case false => true }}
```

Lastly, we simplify the `C` branch using the standard reduction rule for pattern matching.

An important observation is that the above process involves redundant steps, namely the copying and dropping of the outer `match`. The redundancy stems from the fact that (jfloat) automatically inserts the enclosing context E into the body e_2 of the `defjoin` construct. While the extra rewrites do not affect the ultimate result, they make (jfloat) and (abort) more like a combined rule, and in fact, GHC does eagerly apply the two rules in pair when it encounters a non-tail-called join point. In our view, GHC is doing the right thing, and the formal development of Maurer et al. allows too much freedom to decide how to apply rewrite rules.

Our Solution. If we use the $\mathscr{C}$ operator, we can achieve the above optimization in fewer steps. The key is the following rewrite rule, which plays a similar role as (jfloat):

$$E[\mathscr{C}\ (\lambda j.e)] = \mathscr{C}\ (\lambda k.\text{defcont } j(x) = k(E[x]);\ e) \qquad \text{(cfloat)}$$

We start by wrapping the inner `match` around $\mathscr{C}$, and inserting a call to `k` into *non-jumping* subterms:

```
(𝒞 { k => defcont j(x) = k(BIG)
          v match { case A => j(1); case B => j(2); case C => k(true) }})
match { case true  => false; case false => true }
```

After applying (cfloat), we are very close to our goal:

```
𝒞 { k =>
  defcont j(x) = k(BIG match { case true => false; case false => true })
  v match {
    case A => j(1)
    case B => j(2)
    case C => k(true match { case true => false; case false => true })}}
```

Now, we obtain exactly what we want by simply reducing the `C` branch.

Notice that our optimization does not have the copy-and-drop steps observed in Maurer et al.'s treatment. This is because (cfloat) moves the context only to the continuation k, not to the body e. Since we can now freely choose where to inject a captured context, we no longer need to eagerly apply the (abort) rule to cancel this action. Thus, having (cfloat) makes the formal development more coherent with the actual implementation.

2.3 Further Advantages

While we have been focusing on the rewrite steps in the above example, our IR has two more advantages over Maurer et al.'s proposal. First, in our IR, we can perform optimizations both in direct style and in CPS. This is due to the succinct CPS semantics of the $\mathscr{C}$ operator, and leads to better performance of the output code. Second, we can model 2nd-class regular functions to further reduce expensive allocation. This comes from the fact that we keep track of escaping capabilities in the type system, instead of distinguishing continuations at the level of syntax. More detail on these benefits can be found in Sections 6 and 8.

3 $\lambda_{\perp}^{1/2}$: THE PROPOSED IR

We now describe $\lambda_{\perp}^{1/2}$, our proposed IR. As already touched upon in the previous section, we design the IR based on the following ideas:

- To handle continuations in direct style, we integrate a control operator $\mathscr{C}$ into our IR, and give it a semantics that is suited for compiler optimization. This helps us avoid redundant rewrites observed in Maurer et al.'s IR.
- To account for the non-returning nature of continuations, we incorporate the empty type $\perp$, and assign continuations a type of the form $T \to \perp$. This allows us to identify jumping terms by looking at their type.
- To efficiently compile non-escaping functions, including continuations *and* user-defined ones, we employ a distinction between 1st- and 2nd-class values. This enables us to avoid unnecessary allocation of closures.

3.1 Syntax

In Figure 1, we present the syntax of $\lambda_{\perp}^{1/2}$. The term language includes ordinary λ-terms as well as constants (of base type), pairs, `if`, `let`, and the $\mathscr{C}$ operator. To explicitly state whether a term can

Syntax

$$
\begin{array}{rcll}
n & ::= & 1 \mid 2 & \text{Annotations} \\
t & ::= & c \mid x^n \mid \lambda x^n.t \mid t\ t \mid (t\ {}_{n,n}\ t) \mid \mathsf{fst}\ t \mid \mathsf{snd}\ t & \text{Terms} \\
 & \mid & \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \mid \mathsf{let}\ x^n = t\ \mathsf{in}\ t \mid \mathscr{C}\ t & \\
T & ::= & B \mid T^n \to U \mid T^n \times T^n & \text{Value Types} \\
U & ::= & \bot \mid T & \text{Expression Types}
\end{array}
$$

Fig. 1. $\lambda_{\bot}^{1/2}$ Syntax

Values, Evaluation Contexts, and Single-Frame Contexts

$$
\begin{array}{rcll}
v & ::= & c \mid \lambda x^n.t \mid (v\ {}_{n,n}\ v) & \text{Values} \\
E & ::= & \Box \mid E\ t \mid v\ E \mid (E\ {}_{n,n}\ t) \mid (v\ {}_{n,n}\ E) \mid \mathsf{fst}\ E \mid \mathsf{snd}\ E & \text{Evaluation Contexts} \\
 & \mid & \mathsf{if}\ E\ \mathsf{then}\ t\ \mathsf{else}\ t \mid \mathsf{let}\ x^n = E\ \mathsf{in}\ t \mid \mathscr{C}\ E & \\
F & ::= & \Box\ t \mid v\ \Box \mid (\Box\ {}_{n,n}\ t) \mid (v\ {}_{n,n}\ \Box) \mid \mathsf{fst}\ \Box \mid \mathsf{snd}\ \Box & \text{Single-Frame Contexts} \\
 & \mid & \mathsf{if}\ \Box\ \mathsf{then}\ t\ \mathsf{else}\ t \mid \mathsf{let}\ x^n = \Box\ \mathsf{in}\ t \mid \mathscr{C}\ \Box &
\end{array}
$$

High-level Operational Semantics $\boxed{E[t] \Rightarrow E[t']}$

$$
\begin{array}{rcll}
E[(\lambda x.t)\ v] & \Rightarrow & E[t[v/x]] & \\
E[\mathsf{fst}\ (v_1\ {}_{n_1,n_2}\ v_2)] & \Rightarrow & E[v_1] & \\
E[\mathsf{snd}\ (v_1\ {}_{n_1,n_2}\ v_2)] & \Rightarrow & E[v_2] & \\
E[\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3] & \Rightarrow & E[t_2] & \\
E[\mathsf{if}\ \mathsf{false}\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3] & \Rightarrow & E[t_3] & \\
E[\mathsf{let}\ x = v\ \mathsf{in}\ t_2] & \Rightarrow & E[t_2[v/x]] & \\
E[F[\mathscr{C}\ (\lambda k.t)]] & \Rightarrow & E[\mathscr{C}\ (\lambda j.t[\lambda x.j\ F[x]/k])] & \text{if } F[\mathscr{C}\ (\lambda k.t)] : T \\
\mathscr{C}\ (\lambda k.k\ t) & \Rightarrow & t & \text{if } k \notin FV(t)
\end{array}
$$

Fig. 2. $\lambda_{\bot}^{1/2}$ High-level Operational Semantics

escape or not, we annotate certain language constructs with their class information: for instance, x^1 is a 1st-class variable, and $(v_1\ {}_{2,2}\ v_2)$ is a pair consisting of two 2nd-class values. These annotations are either given by the programmer (if the source language has support for value classification), or inserted by the compiler (via e.g. contification). Note that, when a `let` construct binds a 2nd-class variable, it serves as the `defcont` keyword from the previous section.

The type language is designed to express the returning and non-returning behavior of programs. Observe that we define value types T (which consist of base types, arrow types, and pair types) separately from expression types U (which include $\bot$). We use this distinction to constrain occurrences of jumps in programs. For instance, a function may have jumping subterms in its body, since the co-domain of a function type can be the $\bot$ type. In contrast, a pair may only be built with non-jumping terms, as both components of a pair type are value types.

3.2 Operational Semantics

The $\lambda_{\perp}^{1/2}$ language has a left-to-right, call-by-value semantics, as defined in Figure 2. We omit class annotations in the reduction rules because they do not affect the high-level behavior of programs. The only interesting rules are those dealing with the $\mathscr{C}$ operator. When the argument of $\mathscr{C}$ has reduced to a function $\lambda k.t$, one of the following reductions can happen. First, if $\mathscr{C}$ is surrounded by a non-empty context $E[F[\,]]$, it "bubbles up" by moving F inside the continuation k. We require F to have a non-$\perp$ result type to ensure the well-typedness of the reduct. Second, if $\mathscr{C}$ is invoked at top-level, and its argument has the form $\lambda k.k\ t$, the whole construct is replaced by the term t. In other words, capturing and immediately applying the continuation has no computational effect.

Comparison with Felleisen's Control Operator. As mentioned earlier, our $\mathscr{C}$ operator is inspired by Felleisen's control operator (here denoted as $\mathcal{C}$), which has the following reduction rule:

$$E[\mathcal{C}\ (\lambda k.t)] \quad \Rightarrow \quad t[\lambda x.\mathscr{A}\ E[x]/k]$$

where $\mathscr{A}$ is an aborting operation defined as $\lambda x.\mathcal{C}\ (\lambda k.x)$. There are two differences between the ways $\mathcal{C}$ and $\mathscr{C}$ behave. First, while $\mathcal{C}$ captures the whole evaluation context at one time, $\mathscr{C}$ captures one context frame per step. In an untyped language, the former reduction works perfectly, but in a typed language, it causes a mismatch between the type of the continuation (which targets $\perp$) and that of the whole program (which cannot be $\perp$). If we consume the context piece by piece, we will end up with a program of the form $\mathscr{C}\ t$, which, as we will see in Section 3.3, must have a value type.

The second difference is that, whereas $\mathcal{C}$ inserts an abort operator into the continuation, $\mathscr{C}$ does not. In a typed setting, adding an abort means allowing non-tail calls to continuations. As we saw in Section 2.2, the (jfloat) rule of Maurer et al. may bring such calls into programs, but our (cfloat) rule does not, since it gives us full control over where to inject a captured context.

3.3 Typing

In $\lambda_{\perp}^{1/2}$, whether a term is used in the right way depends on its 1st / 2nd-class status. Therefore, to obtain safety guarantees, we must explicitly handle class information in the type system. Following Osvald et al. [2016], we use a typing judgment of the form $G \vdash t :^{n} U$, which reads: under typing environment G, term t is of type U and has a n-class status.

Let us look at the typing rules in Figure 3, focusing our attention to the class information. In (TVar), we see a premise $x^{m} : T \in G^{[\leq n]}$, demanding the surface annotation m to be smaller than or equal to the concluding annotation n. This essentially allows us to use a 1st-class variable as a 2nd-class value, which is safe because 2nd-class values are used in a more restricted manner.

Another important rule is (TAbs). We see that the rule requires a 1st-class body t, making it impossible to return a 2nd-class value by a λ. We also find that the body must be typable under a restricted environment $G^{[\leq n]}$, guaranteeing that 1st-class functions do not refer to 2nd-class values through free variables.

The class distinction is also relevant in the (TCtrl) rule. The premise has a type $(T^{n} \to \perp)^{2} \to \perp$, stating that the continuation captured by $\mathscr{C}$ must be used in a 2nd-class manner. This ensures efficient compilation of continuations.

We next shift our attention to types. As mentioned earlier, we use the $\perp$ type to express whether a function returns or not. In (TCtrl), we have two occurrences of the $\perp$ type: the first one represents the non-returning nature of the captured continuation, while the second one forces the continuation to be used in the body.

A careful analysis of other rules reveals that the type system forces continuations to be used as jumps. Observe the judgments in the premises: they have an expression type only if the subject is a

Type Environments

$$\begin{array}{rcl} G & ::= & \emptyset \mid G, x^n : T \\ G^{[\leq n]} & = & \{x^m : T \in G \mid m \leq n\} \end{array}$$

Typing Rules $\boxed{G \vdash t :^n U}$

$$G \vdash c :^n B \quad \text{(TCst)}$$

$$\frac{x^m : T \in G^{[\leq n]}}{G \vdash x^m :^n T} \quad \text{(TVar)}$$

$$\frac{G^{[\leq n]}, x^m : T \vdash t :^1 U}{G \vdash \lambda x^m.t :^n T^m \to U} \quad \text{(TAbs)}$$

$$\frac{G \vdash t_1 :^2 T^m \to U \quad G \vdash t_2 :^m T}{G \vdash t_1\ t_2 :^n U} \quad \text{(TApp)}$$

$$\frac{G \vdash t_1 :^{n_1} T_1 \quad G \vdash t_2 :^{n_2} T_2}{G \vdash (t_1\ {}_{n_1,n_2}\ t_2) :^{max(n_1,n_2)} T_1^{n_1} \times T_2^{n_2}} \quad \text{(TPair)}$$

$$\frac{G \vdash t :^n T_1^{n_1} \times T_2^{n_2}}{G \vdash \mathsf{fst}\ t :^{n_1} T_1} \quad \text{(TFst)}$$

$$\frac{G \vdash t :^n T_1^{n_1} \times T_2^{n_2}}{G \vdash \mathsf{snd}\ t :^{n_2} T_2} \quad \text{(TSnd)}$$

$$\frac{G \vdash t_1 :^2 \mathsf{bool} \quad G \vdash t_2 :^n U \quad G \vdash t_3 :^n U}{G \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 :^n U} \quad \text{(TIf)}$$

$$\frac{G \vdash t_1 :^m T_1 \quad G, x^m : T_1 \vdash t_2 :^n U}{G \vdash \mathsf{let}\ x^m = t_1\ \mathsf{in}\ t_2 :^n U} \quad \text{(TLet)}$$

$$\frac{G \vdash t :^2 (T^n \to \bot)^2 \to \bot}{G \vdash \mathscr{C}\ t :^n T} \quad \text{(TCtrl)}$$

Fig. 3. $\lambda_\perp^{1/2}$ Type System

subterm appearing in a tail position (e.g., the branches of `if` and the body of `let`). This rules out uses of a jumping term as an argument to a function or an element of a pair.

3.4 Optimizations

Lastly, we present optimization rules in Figure 4. These rules can be applied to any matching part of the program at any time, according to the decision made by the optimizer. The most important one is the floating rule (cfloat) of the $\mathscr{C}$ operator. As noted earlier, the rule only inserts the captured

General Contexts

$$
\begin{array}{rcl}
C & ::= & \Box \mid \lambda x^n.C \mid C\ t \mid t\ C \mid (C\ {}_{n,n}\ t) \mid (t\ {}_{n,n}\ C) \mid \mathsf{fst}\ C \mid \mathsf{snd}\ C \\
& \mid & \mathsf{if}\ C\ \mathsf{then}\ t\ \mathsf{else}\ t \mid \mathsf{if}\ t\ \mathsf{then}\ C\ \mathsf{else}\ t \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ C \\
& \mid & \mathsf{let}\ x^n = C\ \mathsf{in}\ t \mid \mathsf{let}\ x^n = t\ \mathsf{in}\ C \mid \mathscr{C}\ C
\end{array}
$$

Optimizations $\boxed{t = t'}$

$$
\begin{array}{rll}
(\lambda x^n.t)\ v = (\mathsf{let}\ x^n = v\ \mathsf{in}\ t) & & (\text{beta}) \\
\mathsf{fst}\ (v_1\ {}_{n_1,n_2}\ v_2) = v_1 & & (\text{fst}) \\
\mathsf{snd}\ (v_1\ {}_{n_1,n_2}\ v_2) = v_2 & & (\text{snd}) \\
\mathsf{if\ true\ then}\ t_2\ \mathsf{else}\ t_3 = t_2 & & (\text{iftrue}) \\
\mathsf{if\ false\ then}\ t_2\ \mathsf{else}\ t_3 = t_3 & & (\text{iffalse}) \\
(\mathsf{let}\ x^n = v\ \mathsf{in}\ C[x^n]) = \mathsf{let}\ x^n = v\ \mathsf{in}\ C[v] & & (\text{let}) \\
F[\mathscr{C}\ (\lambda k^2.t)] = \mathscr{C}\ (\lambda j^2.\mathsf{let}\ k^2 = \lambda x^n.j^2\ F[x^n]\ \mathsf{in}\ t) & & (\text{cfloat}) \\
\quad \text{if } F \text{ has an } n\text{-class hole, and } F[\mathscr{C}\ (\lambda k^2.t)] : T & & \\
\mathscr{C}(\lambda k^2.k^2\ t) = t & & (\text{celim}) \\
\quad \text{only at top-level and if } k \notin FV(t) & &
\end{array}
$$

Fig. 4. Optimizations

context frame into the body of the continuation j, and thus helps us avoid redundant rewrites that would happen in Maurer et al.'s IR.

3.5 Type Safety

Having presented the specifications of our IR, let us discuss its metatheory. One of the properties we are interested in is type safety with regard to the semantics in Figure 2. Following Wright and Felleisen [1994], we decompose our goal into two propositions: *preservation* and *progress*.

Proposition 3.1 (Preservation). *If* $G \vdash t :^n U$ *and* $t \Rightarrow^* t'$, *then* $G \vdash t' :^n U$.

Proof. The proof is by induction on the derivation of t. We first prove preservation under single-step reduction, and then extend the result to multiple steps by induction on the length of the reduction sequence. As usual, we need substitution and inversion lemmas, and in addition to those, we need a lemma stating that 1st-class values can be judged as 2nd-class.

Lemma 3.2 (Class Conversion). *If* $G \vdash t :^1 U$, *then* $G \vdash t :^2 U$.

Now we show two sub-cases of the application case.

Case 1 (β reduction)**.** Suppose we have the following derivation:

$$
\frac{G \vdash \lambda x^m.t :^2 T^m \to U \quad G \vdash v :^m T}{G \vdash (\lambda x^m.t)\ v :^n U} \quad (\text{TApp})
$$

We must show $G \vdash t[v/x^m] :^n U$. If $n = 1$, the goal easily follows by the substitution lemma, because functions must have a 1st-class body. If $n = 2$, the goal follows by an additional appeal to Lemma 3.2.

Case 2 ($\mathscr{C}$ reduction)**.** Suppose we have the following derivation:

$$\frac{G \vdash v :^2 T_1^m \to T_2 \quad G \vdash \mathscr{C}\ (\lambda k^2.t) :^m T_1}{G \vdash v\ (\mathscr{C}\ (\lambda k^2.t)) :^n T_2} \quad \text{(TApp)}$$

We must show $G \vdash \mathscr{C}\ (\lambda j^2.t[\lambda x^m.j^2\ (v\ x^m)/k^2]) :^n T_2$. By inversion, we know that $G, k^2 : T_1^m \to \bot \vdash t :^2 \bot$. To derive what we want, j^2 must have type $T_2^n \to \bot$, and when assuming x^m is of type T_1, the function $\lambda x^m.j^2\ (v\ x^m)$ can be given the type $T_1^m \to \bot$, validating the substitution for the continuation variable[2]. This implies that the post-reduction term has the expected type.

□

Note that the preservation property can be stated for the optimization rules (Figure 4) as well, because they are essentially a variant of reduction rules that operate on open terms.

Proposition 3.3 (Progress). *If $\emptyset \vdash t :^n U$, then either t is a value, or it is a stuck term of the form $\mathscr{C}\ t$ (where t is not convertible to $\lambda k^2.k^2\ t'$), or there is some t' such that $t \Rightarrow t'$.*

Proof. The proof is again by induction on the derivation. The value cases are trivial. In the application, projection, `if`, and $\mathscr{C}$ cases, we need a canonical forms lemma, which tells us the shape of a closed value by looking at its type.

Lemma 3.4 (Canonical Forms). *Suppose $\emptyset \vdash v : T$.*

(1) If $T = bool$, then $v = true$ or $v = false$.
(2) If $T = T_1^n \to U$, then $v = \lambda x^n.t$ for some x^n and t.
(3) If $T = T_1^{n_1} \times T_2^{n_2}$, then $v = (v_1\ {}_{n_1,n_2}\ v_2)$ for some v_1 and v_2.

□

One thing we would like to note here is that, while we do not syntactically enforce the argument of $\mathscr{C}$ to be a continuation-awaiting function $\lambda k^2.t$, the canonical forms lemma guarantees that it always reduces to the expected form — remember that the argument must have an arrow type.

4 CPS TRANSLATION

Using the $\mathscr{C}$ operator, we can fruitfully perform all the direct-style optimizations that are possible in the join points IR of Maurer et al. [2017]. However, depending on the circumstances, a subsequent CPS optimization step may help us produce even more efficient code. Therefore, it is ideal to allow both direct-style and CPS optimizations in the same IR.

In this section, we present a CPS translation of $\lambda_\bot^{1/2}$. Thanks to the clean separation between returning and escaping behaviors, we can guarantee that the result of the translation is as efficient as Kennedy [2007]'s. Specifically, we define our translation based on the following principles:

- We translate programs in a *selective*[3] manner, by leaving $\mathscr{C}$-captured continuations as is.
- We give continuations a 2nd-class status, by fixing their introduction form to λk^2.

[2]The argument implicitly assumes that the top-level program is 1st-class. This does not lead to loss of generality, because the only situation in which a term cannot be judged 1st-class is when it has free variables that are 2nd-class.
[3]Our selective translation should not be confused with the selective CPS translations in the continuations literature [Asai and Uehara 2018; Nielsen et al. 2001; Rompf et al. 2009], which turn effectful terms into CPS and keeps pure terms in direct style.

Term Translation $\boxed{[\![G \vdash t :^n T]\!],\ [\![G \vdash t :^n \bot]\!]'}$

$$[\![G \vdash c :^n B]\!] = \lambda k^2.k^2\ c \tag{CCst}$$

$$\frac{x^m : T \in G^{[\leq n]}}{[\![G \vdash x^m :^n T]\!] = \lambda k^2.k^2\ x^m} \tag{CVar}$$

$$\frac{[\![G^{[\leq n]}, x^m : T_1 \vdash t :^1 T_2]\!] = t'}{[\![G \vdash \lambda x^m.t :^n T_1^m \to T_2]\!] = \lambda k^2.k^2\ (\lambda(x^m\ {}_{m,2}\ k_1^2)^2.t'\ k_1^2)} \tag{CAbs1}$$

$$\frac{[\![G^{[\leq n]}, x^m : T \vdash t :^1 U]\!]' = t'}{[\![G \vdash \lambda x^m.t :^n T_1^m \to T_2]\!] = \lambda k^2.k^2\ (\lambda x^m.[\![t]\!]')} \tag{CAbs2}$$

$$\frac{[\![G \vdash t_1 :^2 T_1^m \to T_2]\!] = t_1' \qquad [\![G \vdash t_2 :^m T_1]\!] = t_2'}{[\![G \vdash t_1\ t_2 :^n T_2]\!] = \lambda k^2.t_1'\ (\lambda v_1^2.t_2'\ (\lambda v_2^m.v_1^2\ (v_2^m\ {}_{m,2}\ (\lambda x^1.k^2\ x^1))))} \tag{CApp}$$

$$\frac{[\![G \vdash t_1 :^2 T^m \to \bot]\!] = t_1' \qquad [\![G \vdash t_2 :^m T]\!] = t_2'}{[\![G \vdash t_1\ t_2 :^n T_2]\!]' = t_1'\ (\lambda v_1^2.t_2'\ (\lambda v_2^m.v_1^2\ v_2^m))} \tag{DApp}$$

$$\frac{[\![G \vdash t_1 :^{n_1} T_1]\!] = t_1' \qquad [\![G \vdash t_2 :^{n_2} T_2]\!] = t_2'}{[\![G \vdash (t_1\ {}_{n_1,n_2}\ t_2) :^{max(n_1,n_2)} T_1^{n_1} \times T_2^{n_2}]\!] = \lambda k^2.t_1'\ (\lambda v_1^{n_1}.t_2'\ (\lambda v_2^{n_2}.k^2\ (v_1^{n_1}\ {}_{n_1,n_2}\ v_2^{n_2})))} \tag{CpPir}$$

$$\frac{[\![G \vdash t :^n T_1^{n_1} \times T_2^{n_2}]\!] = t'}{[\![G \vdash \mathsf{fst}\ t :^{n_1} T_1]\!] = \lambda k^2.t'\ (\lambda v^n.\mathsf{let}\ v_1^{n_1} = \mathsf{fst}\ v^n\ \mathsf{in}\ k^2\ v_1^{n_1})} \tag{CFst}$$

$$\frac{[\![G \vdash t :^n T_1^{n_1} \times T_2^{n_2}]\!] = t'}{[\![G \vdash \mathsf{snd}\ t :^{n_2} T_2]\!] = \lambda k^2.t'\ (\lambda v^n.\mathsf{let}\ v_2^{n_2} = \mathsf{snd}\ v^n\ \mathsf{in}\ k^2\ v_2^{n_2})} \tag{CSnd}$$

Fig. 5. CPS Translation

4.1 The Translation

In Figures 5 and 6, we define our selective CPS translation. The translation is a source-to-source mapping; more precisely, its target language is the fragment of $\lambda_{\bot}^{1/2}$ without the $\mathscr{C}$ operator. As we translate programs *selectively*, we use two kinds of translation $[\![\ ]\!]$ and $[\![\ ]\!]'$. The former takes care of terms having a value type, and generates a CPS term as the result. The latter applies to terms of type $\bot$, and produces a direct-style term. Since the type information is not available from the syntax, we define these translations on the typing judgment.

Let us walk through individual rules. Constants and variables are translated the same way as in a non-selective, call-by-value translation. Abstractions are uniformly turned into CPS, but there are two possible forms of the post-translation body. If the source abstraction is an ordinary function that returns, the target abstraction requires a function argument x^m *and* a return continuation k_1^2. If the abstraction is a jumping function that does not return, the target abstraction has a direct-style body. The two possibilities are present in the type translation $[\![\ ]\!]$ as well: the CPS counterpart of

Term Translation $\boxed{[\![G \vdash t :^n T]\!],\ [\![G \vdash t :^n \bot]\!]'}$

$$\frac{[\![G \vdash t_1 :^2 \mathsf{bool}]\!] = t_1' \qquad [\![G \vdash t_2 :^n T]\!] = t_2' \qquad [\![G \vdash t_3 :^n T]\!] = t_3'}{[\![G \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 :^n T]\!] = \lambda k^2.t_1'\ (\lambda v_1^2.\mathsf{if}\ v_1^2\ \mathsf{then}\ t_2'\ k^2\ \mathsf{else}\ t_3'\ k^2)} \quad (\textsc{CIf})$$

$$\frac{[\![G \vdash t_1 :^2 \mathsf{bool}]\!] = t_1' \qquad [\![G \vdash t_2 :^n \bot]\!]' = t_2' \qquad [\![G \vdash t_3 :^n \bot]\!]' = t_3'}{[\![G \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 :^n \bot]\!]' = t_1'\ (\lambda v_1^2.\mathsf{if}\ v_1^2\ \mathsf{then}\ t_2'\ \mathsf{else}\ t_3')} \quad (\textsc{DIf})$$

$$\frac{[\![G \vdash t_1 :^m T_1]\!] = t_1' \qquad [\![G, x^m : T_1 \vdash t_2 :^n T_2]\!] = t_2'}{[\![G \vdash \mathsf{let}\ x^m = t_1\ \mathsf{in}\ t_2 :^n T_2]\!] = \lambda k^2.t_1'\ (\lambda v_1^m.\mathsf{let}\ x^m = v_1^m\ \mathsf{in}\ t_2'\ k^2)} \quad (\textsc{CLet})$$

$$\frac{[\![G \vdash t_1 :^m T_1]\!] = t_1' \qquad [\![G, x^m : T_1 \vdash t_2 :^n T_2]\!]' = t_2'}{[\![G \vdash \mathsf{let}\ x^m = t_1\ \mathsf{in}\ t_2 :^n T_2]\!]' = t_1'\ (\lambda v_1^m.\mathsf{let}\ x^m = v_1^m\ \mathsf{in}\ t_2')} \quad (\textsc{DLet})$$

$$\frac{[\![G \vdash t :^2 (T^n \to \bot)^2 \to \bot]\!] = t'}{[\![G \vdash \mathscr{C}\ t :^n T]\!] = \lambda k^2.t'\ (\lambda v^2.v^2\ k^2)} \quad (\textsc{CCtrl})$$

Type Translation $\boxed{[\![T]\!]}$

$$\begin{array}{rcl} [\![B]\!] & = & B \\ [\![T_1^m \to T_2]\!] & = & ([\![T_1]\!]^m \times ([\![T_2]\!]^1 \to \bot)^2)^2 \to \bot \\ [\![T^m \to \bot]\!] & = & [\![T]\!]^m \to \bot \\ [\![T_1^{n_1} \times T_2^{n_2}]\!] & = & [\![T_1^{n_1}]\!] \times [\![T_2^{n_2}]\!] \end{array}$$

Fig. 6. CPS Translation (continued)

$T_1^n \to T_2$ has a subcomponent $T_2{}^1 \to \bot$ representing the return continuation, which is missing in the CPS counterpart of $T^n \to \bot$.

Similarly to abstraction, we have two rules for translating application. The first is used for ordinary function calls. The resulting term requires a return continuation k^2, and uses it to continue the execution after the application. The second rule accounts for continuation calls, i.e., jumps. Since there is no "rest of the work" after jumps, the resulting term does not take in a continuation.

Pairs and projections have one translation per construct, because they cannot inhabit the $\bot$ type. The rules for `if` and `let` shares the same pattern with those for application.

Lastly, we have a single rule for the control construct $\mathscr{C}\ t$. Recall that t is a non-returning function that demands a continuation. On the right-hand side of the translation, v^2 is a function that requires a continuation as its argument but not a return continuation. By feeding it with the top-level continuation k^2, we obtain the result of the entire program.

Before we move on, let us visit the (CApp) rule again. We see that the return continuation k^2 is η-expanded to $\lambda x^1.k^2\ x^1$. The trick is necessary for guaranteeing the stack discipline of CPS programs. Since we would like the stack to shrink at every return, we must make sure that application of a return continuation does not make the stack grow. The growing can be avoided by imposing a restriction on return continuations: they always accept a 1st-class argument that is allocated on the

heap. Note that this requirement cannot break typability of the resulting term, since every function is guaranteed to have a 1st-class body.

Remark. The CPS translation we presented above yields *administrative redexes*, i.e., applications that are not present in the source program. We could avoid these redexes by adapting the translation to a higher-order one, following the recipe of Danvy and Filinski [1992].

4.2 Properties of CPS Translation

Our CPS translation preserves the meaning and typing of programs. In this section, we sketch the proof of these properties, using the following abbreviation for readability:

$$[\![t]\!] = t' \quad \stackrel{\text{def}}{\equiv} \quad [\![G \vdash t :^n T]\!] = t'$$
$$[\![t]\!]' = t' \quad \stackrel{\text{def}}{\equiv} \quad [\![G \vdash t :^n \bot]\!]' = t'$$

Proposition 4.1 (Correctness). *Let $=$ be the least congruence relation containing the reduction rules in Figure 2 and the η rule (i.e., $\lambda x^n.t\ x^n \Rightarrow v$ if $x^n \notin FV(t)$). The following propositions hold.*

(1) If $G \vdash t :^n T$ and $t \Rightarrow^* t'$, then $[\![t]\!] = [\![t']\!]$.
(2) If $G \vdash t :^n \bot$ and $t \Rightarrow^* t'$, then $[\![t]\!]' = [\![t']\!]'$.

Proof. Similarly to the preservation property from Section 3.5, we first prove correctness with regard to single-step reduction, by cases on $t \Rightarrow t'$. This requires two lemmas, telling us how to translate terms involving substitution and plugging operations.

Lemma 4.2 (Commutativity). *Let $[\![G \vdash v :^n T]\!]_v$ be a translation on values, defined as follows:*

$$[\![G \vdash c :^n B]\!]_v = c \qquad (\mathrm{C}_v\textsc{Cst})$$

$$\frac{x^m : T \in G^{[\leq n]}}{[\![G \vdash x^m :^n T]\!]_v = x^m} \qquad (\mathrm{C}_v\textsc{Var})$$

$$\frac{[\![G^{[\leq n]}, x^m : T_1 \vdash t :^1 T_2]\!] = t'}{[\![G \vdash \lambda x^m.t :^n T_1^m \to T_2]\!]_v = \lambda(x^m\ {}_{m,2}\ k_1^2)^2.[\![t]\!]\ k_1^2} \qquad (\mathrm{C}_v\textsc{Abs1})$$

$$\frac{[\![G^{[\leq n]}, x^m : T \vdash t :^1 U]\!]' = t'}{[\![G \vdash \lambda x^m.t :^n T_1^m \to T_2]\!]_v = \lambda x^m.[\![t]\!]'} \qquad (\mathrm{C}_v\textsc{Abs2})$$

$$\frac{[\![G \vdash v_1 :^{n_1} T_1]\!]_v = v_1' \qquad [\![G \vdash v_2 :^{n_2} T_2]\!]_v = v_2'}{[\![G \vdash (v_1\ {}_{n_1,n_2}\ v_2) :^{max(n_1,n_2)} T_1^{n_1} \times T_2^{n_2}]\!]_v = (v_1'\ {}_{n_1,n_2}\ v_2')} \qquad (\mathrm{C}_v\textsc{Pair})$$

(1) If $G \vdash t :^n T$, then $[\![t[v/x]]\!] = [\![u]\!][[\![v]\!]_v/x]$.
(2) If $G \vdash t :^n \bot$, then $[\![t[v/x]]\!]' = [\![u]\!]'[[\![v]\!]_v/x]$.

Lemma 4.3 (Compositionality). *Let $[\![F]\!]$ be a translation on context frames, defined as follows:*

$$
\begin{array}{rcl}
[\![\Box\ t_2]\!] & = & \lambda v_1^2.\lambda k^2.[\![t_2]\!]\ (\lambda v_2^m.v_1^2\ ((v_2^m\ {}_{m,2}\ k^2))) \\
[\![v_1\ \Box]\!] & = & \lambda v_2^m.\lambda k^2.[\![v_1]\!]_v\ (v_2^m\ {}_{m,2}\ k^2) \\
[\![(\Box\ {}_{n_1,n_2}\ t_2)]\!] & = & \lambda v_1^{n_1}.\lambda k^2.[\![t_2]\!]\ (\lambda v_2^{n_2}.k^2\ (v_1^{n_1}\ {}_{n_1,n_2}\ v_2^{n_2})) \\
[\![(v_1\ {}_{n_1,n_2}\ \Box)]\!] & = & \lambda v_2^{n_2}.\lambda k^2.k^2\ ([\![v_1]\!]_v\ {}_{n_1,n_2}\ v_2^{n_2}) \\
[\![fst\ \Box]\!] & = & \lambda v^n.\lambda k^2.let\ v_1^{n_1} = fst\ v^n\ in\ k^2\ v_1^{n_1} \\
[\![snd\ \Box]\!] & = & \lambda v^n.\lambda k^2.let\ v_2^{n_2} = snd\ v^n\ in\ k^2\ v_2^{n_2} \\
[\![if\ \Box\ then\ t_2\ else\ t_3]\!] & = & \lambda v_1^2.\lambda k^2.if\ v_1^2\ then\ [\![t_2]\!]\ k^2\ else\ [\![t_3]\!]\ k^2 \\
[\![let\ x^m = \Box\ in\ t_2]\!] & = & \lambda x^m.\lambda k^2.[\![t_2]\!]\ k^2 \\
[\![\mathscr{C}\ \Box]\!] & = & \lambda v^2.\lambda k^2.v^2\ k^2
\end{array}
$$

We have $[\![F[t]]\!] = \lambda k^2.[\![t]\!]\ (\lambda v^n.[\![F]\!]\ v^n\ k^2)$.

Using these lemmas, we can show that the CPS image of a β / $\mathscr{C}$ redex is semantically equivalent to the CPS image of its reduct.

Case 1 (β reduction (of type T))**.**

$$
\begin{array}{ll}
[\![(\lambda x.t)\ v]\!] & \\
= \lambda k.(\lambda k.k\ (\lambda(x\ , k_1).[\![t]\!]\ k_1))\ (\lambda v_1.[\![v]\!]\ (\lambda v_2.v_1\ (v_2\ , \lambda x.k\ x))) & \text{by translation} \\
= \lambda k.(\lambda(x\ , k_1).[\![t]\!]\ k_1)\ ([\![v]\!]_v\ , \lambda x.k\ x) & \text{by } \beta \\
= \lambda k.[\![t]\!][[\![v]\!]_v/x]\ (\lambda x.k\ x) & \text{by } \beta \\
= \lambda k.[\![t]\!][[\![v]\!]_v/x]\ k & \text{by } \eta \\
= [\![t]\!][[\![v]\!]_v/x] & \text{by } \eta \\
= [\![t[v/x]]\!] & \text{by Lemma 4.2}
\end{array}
$$

Case 2 ($\mathscr{C}$ reduction (non-top level))**.**

$$
\begin{array}{ll}
[\![F[\mathscr{C}\ (\lambda k.t)]]\!] & \\
= \lambda k'.[\![(\mathscr{C}\ (\lambda k.t))]\!]\ (\lambda v.[\![F]\!]\ v\ k') & \text{Lemma 4.3} \\
= \lambda k'.(\lambda k''.(\lambda k.[\![t]\!]')\ k'')\ (\lambda v.[\![F]\!]\ v\ k') & \text{by translation and } \beta \\
= \lambda k'.(\lambda k.[\![t]\!]')\ (\lambda v.[\![F]\!]\ v\ k') & \text{by } \eta \\
= \lambda k'.(\lambda k.[\![t]\!]')\ (\lambda v.[\![F[v]]\!]\ k') & \text{by translation} \\
= \lambda k'.[\![t]\!]'[\lambda v.[\![F[v]]\!]\ k'/k] & \text{by } \beta \\
= \lambda k'.[\![t]\!]'[\lambda v.[\![F[v]]\!]\ (\lambda v_2.k'\ v_2)/k] & \text{by } \eta \\
= \lambda k'.[\![t]\!]'[\lambda v.[\![k'\ F[v]]\!]'/k] & \text{by translation} \\
= \lambda k'.[\![t]\!]'[[\![\lambda v.k'\ F[v]]\!]_v/k] & \text{by translation} \\
= \lambda k'.[\![t[\lambda v.k'\ F[v]/k]]\!]' & \text{by Lemma 4.2} \\
= \lambda k'.(\lambda j.[\![t[\lambda v.j\ F[v]/k]]\!]')\ k' & \text{by } \beta \\
= \lambda k'.(\lambda v.v\ k')\ (\lambda j.[\![t[\lambda v.j\ F[v]/k]]\!]') & \text{by } \beta \\
= \lambda k'.[\![\lambda j.t[\lambda v.j\ F[v]/k]]\!]\ (\lambda v.v\ k') & \text{by } \beta \\
= [\![\mathscr{C}\ (\lambda j.t[\lambda v.j\ F[v]/k])]\!] & \text{by translation}
\end{array}
$$

Case 3 ($\mathscr{C}$ reduction (top-level))**.**

$$
\begin{aligned}
&[\![\mathscr{C}\ (\lambda k.k\ t)]\!] \\
&= \lambda k'.[\![\lambda k.k\ t]\!]\ (\lambda v.v\ k') && \text{by translation} \\
&= \lambda k'.(\lambda v.v\ k')\ (\lambda k.[\![t]\!]\ (\lambda v_2.k\ v_2)) && \text{by } \beta \\
&= \lambda k'.[\![t]\!]\ (\lambda v_2.k'\ v_2) && \text{by } \beta \\
&= \lambda k'.[\![t]\!]\ k' && \text{by } \eta \\
&= [\![t]\!] && \text{by } \eta
\end{aligned}
$$

After proving other cases, we derive the main statement by induction on the reduction steps. □

Proposition 4.4 (Type Preservation).
(1) If $G \vdash t :^n T$, *then* $[\![G]\!] \vdash [\![t]\!] :^n ([\![T]\!]^n \to \bot)^2 \to \bot$.
(2) If $G \vdash t :^n \bot$, *then* $[\![G]\!] \vdash [\![t]\!]' :^n \bot$.

Proof. By induction on the derivation of t. We show some representative cases.

Case 1 (TApp)**.**

Sub-Case 1 ($t_1\ t_2 :^n T_2$)**.** We must show

$$[\![G]\!] \vdash \lambda k^2.[\![t_1]\!]\ (\lambda v_1^2.[\![t_2]\!]\ (\lambda v_2^m.v_1^2\ (v_2^m\ {}_{m,2}\ (\lambda x^1.k^2\ x^1)))) :^n ([\![T_2]\!]^n \to \bot)^2 \to \bot$$

By the induction hypothesis, we have

$$[\![G]\!] \vdash [\![t_1]\!] :^2 ((([\![T_1]\!]^m \times ([\![T_2]\!]^1 \to \bot)^2)^2 \to \bot)^2 \to \bot)^2 \to \bot \text{ , and}$$

$$[\![G]\!] \vdash [\![t_2]\!] :^m ([\![T_2]\!]^m \to \bot)^2 \to \bot$$

It is not hard to see that the application of v_1^2 to the value-continuation pair has type $\bot$; remember that the η expansion on the top-level continuation k^2 is type safe. This implies that the CPS-translated function and argument are passed a correct continuation.

Sub-Case 2 ($t_1\ t_2 :^n \bot$)**.** We must show

$$[\![G]\!]' \vdash [\![t_1]\!]\ (\lambda v_1^2.[\![t_2]\!]\ (\lambda v_2^m.v_1^2\ v_2^m)) :^n \bot$$

By the induction hypothesis, we have

$$[\![G]\!] \vdash [\![t_1]\!] :^2 ((([\![T_1]\!]^m \to \bot)^2 \to \bot)^2 \to \bot \text{ , and } [\![G]\!] \vdash [\![t_2]\!] :^m ([\![T_1]\!]^m \to \bot)^2 \to \bot$$

The goal easily follows by (TAbs) and (TApp).

Case 2 (TCtrl)**.** We must show

$$[\![G]\!] \vdash \lambda k^2.[\![t]\!]\ (\lambda v^n.v^n\ k^2) :^n ([\![T]\!]^n \to \bot)^2 \to \bot$$

By the induction hypothesis, we know that

$$[\![G]\!] \vdash [\![t]\!] :^2 ((([\![T]\!]^n \to \bot)^2 \to \bot)^2 \to \bot)^2 \to \bot$$

Assuming $v^2 :^2 ([\![T]\!]^n \to \bot)^2 \to \bot$, we can easily see that the application $[\![t]\!]\ (\lambda v^n.v^n\ k^2)$ is well-typed. This immediately implies the goal.

□

Values

$$v \quad ::= \quad c \mid a \mid (v \;{}_{n,n}\; v)$$

Evaluation Contexts

$$\begin{array}{rcl} E & ::= & \Box \mid \mathsf{Arg}(t, \rho, E) \mid \mathsf{Fun}(v, E) \\ & \mid & \mathsf{Pair1}(t, n, \rho, E) \mid \mathsf{Pair2}(v, n, \rho, E) \mid \mathsf{Fst}(\rho, E) \mid \mathsf{Snd}(\rho, E) \\ & \mid & \mathsf{If}(t, t, \rho, E) \mid \mathsf{Let}(x^m, t_2, \rho, E) \end{array}$$

Stores

$$\begin{array}{lcll} \rho & ::= & \emptyset \mid \rho, x^n : v & \text{Environments} \\ H & ::= & \emptyset \mid H, a : \langle \rho, i, \lambda x^m.t \rangle & \text{Heaps} \\ S & ::= & \emptyset \mid S, a : \langle \rho, i, \lambda x^m.t \rangle & \text{Stacks} \\ S^{[\le a]} & = & \{b : \_ \in S \mid b \le a\} & \end{array}$$

Fig. 7. Values, Evaluation Contexts, and Stores for Low-Level Semantics

4.3 Stack Discipline of CPS Programs

In the previous subsection, we examined the soundness of our CPS translation with regard to the high-level semantics and typing. Now, we focus our attention to the low-level behavior of CPS programs, and show that they follow a stack discipline.

In direct-style programs, it is easy to see that calls and returns make the stack grow and shrink, but in CPS, this matching is harder to spot, because every function application is a tail call. Working with our IR involves further complications, as we do not syntactically distinguish continuations, and we have 2nd-class functions taking in 2nd-class arguments.

Fortunately, we can recover the stack discipline by exploiting the following observation: every time we call a function that accepts a 1st-class argument, we can revert the stack to the one with which the function was defined. This relies on the fact that 1st-class values can never refer to 2nd-class values, and that every function is tail-called in CPS programs.

To formally analyze the stack behavior of CPS programs, we define a low-level operational semantics for the post-CPS language. Figure 7 shows the refined notion of values, evaluation contexts, and stores. In the low-level semantics, a value is either a constant or an address, which we use to indirectly access closures. These are both stored in the value environment ρ. A closure $\langle \rho, i, \lambda x^m.t \rangle$ carries a value environment ρ and a stack pointer i. To avoid unnecessary allocation, we only store 1st-class closures on the heap, and keep 2nd-class closures on the stack. Note that heaps and stacks are both ordered, i.e., larger addresses appear later in the binding sequence.

The low-level semantics is defined in the style of an abstract machine, as shown in Figure 8. A configuration contains a subset of the following elements: a term t, an environment ρ, a heap H, a stack S, an evaluation context E, and a number n representing the 1st/2nd-class status of t. Among the transition rules, *evals*- and *evale*-rules define how to search for a redex, whereas *cont*-rules tell us what to do when we obtain a value. Note that the separation between *evals* and *evale* comes from the syntactic invariant of CPS programs: application, `if`, and `let` may only appear in a tail position.

Turning the viewpoint to the stack space, there are four rules that are particularly important. Let us first look at the *evale*-rules of abstraction. We see that the generated closure keeps the largest stack address a in the current stack S, helping us remember which 2nd-class references were available at the function definition time. We next observe the *cont*-rules dealing with a Fun-context,

Operational Semantics $\boxed{\langle t;\ \rho;\ H;\ S;\ E\rangle_{evals} \Rightarrow v,\ \ \langle t;\ \rho;\ H;\ S;\ E;\ n\rangle_{evale} \Rightarrow v}$

$$
\begin{array}{rcl}
\langle t_1\ t_2;\ \rho;\ H;\ S;\ E\rangle_{evals} & \Rightarrow & \langle t_1;\ \rho;\ H;\ S;\ \mathsf{Arg}(t_2,\rho,E);\ 2\rangle_{evale} \\
\langle \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3;\ \rho;\ H;\ S;\ E\rangle_{evals} & \Rightarrow & \langle t_1;\ \rho;\ H;\ S;\ \mathsf{If}(t_2,t_3,\rho,E);\ 2\rangle_{evale} \\
\langle \mathsf{let}\ x^m = t_1\ \mathsf{in}\ t_2;\ \rho;\ H;\ S;\ E\rangle_{evals} & \Rightarrow & \langle t_1;\ \rho;\ H;\ S;\ \mathsf{Let}(x^m,t_2,\rho,E);\ m\rangle_{evale} \\
 & & \\
\langle c;\ \rho;\ H;\ S;\ E;\ n\rangle_{evale} & \Rightarrow & \langle E;\ c;\ H;\ S;\ n\rangle_{cont} \\
\langle x^m;\ \rho;\ H;\ S;\ E;\ n\rangle_{evale} & \Rightarrow & \langle E;\ v;\ H;\ S;\ n\rangle_{cont} \\
 & & \text{if } x^m : v \in \rho \\
\langle \lambda x^m.t;\ \rho;\ H;\ S;\ E;\ 1\rangle_{evale} & \Rightarrow & \langle E;\ a_H+1;\ (H, a_H+1 : \langle \rho, a_S, \lambda x^m.t\rangle);\ S;\ 1\rangle_{cont} \\
 & & \text{where } a_H/a_S \text{ are the largest addresses in } H/S \\
\langle \lambda x^m.t;\ \rho;\ H;\ S;\ E;\ 2\rangle_{evale} & \Rightarrow & \langle E;\ a_S+1;\ H;\ (S, a_S+1 : \langle \rho, a_S, \lambda x^m.t\rangle);\ 2\rangle_{cont} \\
 & & \text{where } a_S \text{ is the largest address in } S \\
\langle (t_1\ {}_{n_1,n_2}\ t_2);\ \rho;\ H;\ S;\ E;\ n\rangle_{evale} & \Rightarrow & \langle t_1;\ \rho;\ H;\ S;\ \mathsf{Pair1}(t_2,n_2,\rho,E);\ n_1\rangle_{evale} \\
 & & \text{where } n = max(n_1,n_2) \\
\langle \mathsf{fst}\ t;\ \rho;\ H;\ S;\ E;\ n\rangle_{evale} & \Rightarrow & \langle t;\ \rho;\ H;\ S;\ \mathsf{Fst}(\rho,E);\ n\rangle_{evale} \\
\langle \mathsf{snd}\ t;\ \rho;\ H;\ S;\ E;\ n\rangle_{evale} & \Rightarrow & \langle t;\ \rho;\ H;\ S;\ \mathsf{Snd}(\rho,E);\ n\rangle_{evale} \\
 & & \\
\langle \square;\ v;\ H;\ S;\ n\rangle_{cont} & \Rightarrow & v \\
\langle \mathsf{Arg}(t_2,\rho,E);\ a;\ H;\ S;\ n\rangle_{cont} & \Rightarrow & \langle t_2;\ \rho;\ H;\ S;\ \mathsf{Fun}(a,E);\ m\rangle_{evale} \\
 & & \text{if } a : \langle \rho', a', \lambda x^m.t\rangle \in H/S \\
\langle \mathsf{Fun}(a,E);\ v;\ H;\ S;\ 1\rangle_{cont} & \Rightarrow & \langle t;\ \rho', x^1 : v;\ H;\ S^{[\leq i]};\ E\rangle_{evals} \\
 & & \text{if } a : \left\langle \rho', i, \lambda x^1.t \right\rangle \in H/S \\
\langle \mathsf{Fun}(a,E);\ v;\ H;\ S;\ 2\rangle_{cont} & \Rightarrow & \langle t;\ \rho', x^2 : v;\ H;\ S;\ E\rangle_{evals} \\
 & & \text{if } a : \left\langle \rho', i, \lambda x^2.t \right\rangle \in H/S \\
\langle \mathsf{Pair1}(t_2,n_2,\rho,E);\ v_1;\ H;\ S;\ n_1\rangle_{cont} & \Rightarrow & \langle t_2;\ \rho;\ H;\ S;\ \mathsf{Pair2}(v_1,n_1,\rho,E);\ n_2\rangle_{evale} \\
\langle \mathsf{Pair2}(v_1,n_1,\rho,E);\ v_2;\ H;\ S;\ n_2\rangle_{cont} & \Rightarrow & \langle E;\ (v_1\ {}_{n_1,n_2}\ v_2);\ H;\ S;\ max(n_1,n_2)\rangle_{cont} \\
\langle \mathsf{Fst}(\rho,E);\ (v_1\ {}_{n_1,n_2}\ v_2);\ H;\ S;\ n\rangle_{cont} & \Rightarrow & \langle E;\ v_1;\ H;\ S;\ n_1\rangle_{cont} \\
\langle \mathsf{Snd}(\rho,E);\ (v_1\ {}_{n_1,n_2}\ v_2);\ H;\ S;\ n\rangle_{cont} & \Rightarrow & \langle E;\ v_2;\ H;\ S;\ n_2\rangle_{cont} \\
\langle \mathsf{If}(t_2,t_3,\rho,E);\ \mathsf{true};\ H;\ S;\ 2\rangle_{cont} & \Rightarrow & \langle t_2;\ \rho;\ H;\ S;\ E\rangle_{evals} \\
\langle \mathsf{If}(t_2,t_3,\rho,E);\ \mathsf{false};\ H;\ S;\ 2\rangle_{cont} & \Rightarrow & \langle t_3;\ \rho;\ H;\ S;\ E\rangle_{evals} \\
\langle \mathsf{Let}(x^m,t_2,\rho,E);\ v;\ H;\ S;\ m\rangle_{cont} & \Rightarrow & \langle t_2;\ \rho, x^m : v;\ H;\ S;\ E\rangle_{evals}
\end{array}
$$

Fig. 8. Low-level Operational Semantics of CPS Programs

$$
\begin{array}{rcl}
\langle \mathscr{C}\ t;\ \rho;\ H;\ S;\ E;\ n\rangle_{eval} & \Rightarrow & \langle t;\ \rho;\ H;\ S;\ \mathsf{Ctrl}(E,n);\ 2\rangle_{eval} \\
\\
\langle \mathsf{Fun}(a,E);\ v;\ H;\ S;\ 1\rangle_{cont} & \Rightarrow & \langle t;\ \rho', x^1 : v;\ H;\ S;\ E;\ 1\rangle_{eval} \\
 & & \text{if } a : \left\langle \rho', i, \lambda x^1.t \right\rangle \in H/S \\
\langle \mathsf{Ctrl}(E[F],n);\ a;\ H;\ S;\ 2\rangle_{cont} & \Rightarrow & \langle \mathscr{C}\ (\lambda j^2.t);\ (\rho, k^2 : a_S + 1);\ H; \\
 & & (S, a_S + 1 : \left\langle \rho, a_S, \lambda x^n.j^2\ F[x^n] \right\rangle);\ E;\ 2\rangle \\
 & & \text{if } a : \left\langle \rho, i, \lambda k^2.t \right\rangle \in S \\
\langle \mathsf{Ctrl}(\square,n);\ a;\ H;\ S;\ 2\rangle_{cont} & \Rightarrow & \langle t;\ \rho;\ H;\ S;\ \square;\ n\rangle_{eval} \\
 & & \text{if } a : \left\langle \rho, i, \lambda k^2.k^2\ t \right\rangle \in S \text{ and } k^2 \notin FV(t)
\end{array}
$$

Fig. 9. Low-level Operational Semantics of Direct-Style Programs (excerpt)

which let us evaluate a function's body with an appropriate stack. When the argument is 1st-class, we know that it cannot be stored on the stack, nor can it access stack-allocated values through free variables. This means, the function will never use any pre-existing stack memory beyond its own defining scope. Therefore, we discard all the closures whose address is larger than i. If the argument is 2nd-class, on the other hand, we use the current stack to allow arbitrary 2nd-class references from the argument.

Using this semantics, we can easily check that CPS programs follow a stack discipline. Suppose we have an application $(\lambda x^1.x^1)$ true in the source language. The CPS translation will convert the application into the following term:

$$
\begin{array}{l}
\lambda k_0^2.(\lambda k_1^2.k_1^2\ (\lambda(x^1\ {}_{1,2}\ k_2^2)^2.(\lambda k_3^2.k_3^2\ x^1)\ k_2^2)) \\
\quad (\lambda v_1^2.(\lambda k_4^2.k_4^2\ \mathsf{true})\ (\lambda v_2^1.v_1^2\ (v_2^1\ {}_{1,2}\ (\lambda y^1.k_0^2\ y^1))))
\end{array}
$$

The translation has generated five continuation variables, which means five continuation closures will be allocated on the stack when the program is run with an initial continuation k (which serves as the return continuation of the application). However, at the last step of evaluation, where k is called with true, we can reset the stack to an empty one, because the stack was empty when k was defined. It is important to note that this example follows the literal translation rules as given in Figures 5 and 6, which introduce extraneous administrative redexes. As noted at the end of Section 4.1, these administrative redexes can be eliminated using standard techniques [Danvy and Filinski 1992].

As a final remark, since local continuations are also allocated on the stack, a CPS program may use more stack space compared to its direct-style counterpart. However, the difference must be a constant factor, because the CPS translation introduces at most one continuation per language construct. Furthermore, it is possible to remove the extra stack overhead by identifying purely local continuations (via a syntactic check) and compiling them into direct jumps.

4.4 From CPS Semantics to Direct-Style Semantics

It is straightforward to adjust the low-level semantics to one that accounts for direct-style terms. First, we merge *evals*- and *evale*-rules into a single set of *eval*-rules, allowing non-tail occurrences

of application, `if`, and `let` constructs. Second, in the β reduction rule with a 1st-class argument, we do not shrink the stack but keep all the 2nd-class bindings for the rest of the computation. Third, we add new transition rules for the $\mathscr{C}$ operator. We show these changes in Figure 9. Note that we can expect shrinking events of the stack in this setting, too, although it is not observable from the transition rules.

4.5 Equivalence of Semantics

We have defined three semantics: a high-level semantics for direct-style terms (Figure 2), a low-level semantics for direct-style terms (Figure 9), and a low-level semantics for CPS terms (Figure 8). We would naturally like to show that these semantics are all equivalent, i.e., they always produce the same result for a given program.

Let us begin by comparing the low-level semantics for direct-style and CPS programs, restricting ourselves to the $\mathscr{C}$-free fragment. If we ignore the *evals* / *evale*-distinction, the two semantics basically consist of the same set of transition rules. The only difference is in the β rule for functions receiving a 1st-class argument, which involves stack adjustment in the CPS semantics. This does not affect the observational behavior however; we may just have more elements in the stack when using the direct-style semantics.

We next discuss the high-level and low-level semantics for direct-style programs. It is easy to see that the semantics work the same for conditionals and projections. In the case of β, `let`, and $\mathscr{C}$ reductions, the high-level semantics immediately applies substitution, whereas the low-level semantics extends the environment with a new binding. Again, these treatment has the same effect on the program's behavior.

5 EXTENSIONS

So far, we have integrated into $\lambda_{\perp}^{1/2}$ a minimal set of language features and rewrite rules. To make the IR more realistic, we must support more advanced computation forms, typing facilities, and optimization techniques. In this section, we describe some of the interesting extensions.

Data Types and Recursion. Conditionals, which we included in our IR, are the simplest language construct to which we can apply commuting conversions. If we generalize them to pattern matching on data types, we must allow join point functions to take constructor arguments as parameters, instead of uniformly representing them as thunks.

Data types also give rise to the notion of recursion. It is instructive to show what the low-level semantics of a `let rec` construct would look like, especially in the case where the recursive function is 2nd-class:

$$\langle \mathsf{LetRec}(f^2, t, n, \rho, E);\ a;\ H;\ S;\ 1\rangle_{cont} \Rightarrow \langle t;\ \rho, f^2 : a;\ H;\ (S, a : \left\langle (\rho', f^2 : a), a, \lambda x^m.t' \right\rangle);\ E;\ n\rangle_{eval} \quad \text{if } a : \langle \rho', a - 1, \lambda x^m.t' \rangle \in S$$

When evaluating the body t, we bind the address a of f^2 to a closure whose stack pointer is also a. This is because the function's body refers to the function itself, which is stored in a.

Control Constructs. While the $\mathscr{C}$ operator is useful for optimization purposes, it is not powerful enough to express other control constructs found in the mainstream languages. For instance, semi-corountines and generators (such as `yield` and `foreach`) require 2nd-class *delimited* continuations, which means we need a control delimiter for the $\mathscr{C}$ operator. In the case of proper coroutines, continuations are stored in mutable references, hence we must relax the (TCtrl) rule so that it can introduce *1st-class* continuations.

Polymorphism. Shifting our attention to the type system, one important extension would be supporting parametric polymorphism in the style of System F [Girard et al. 1989]. To enable reasoning about programs in the presence of type variables, we must make sure that they can never be instantiated to the $\bot$ type. This restriction makes the selective CPS translation and its type preservation proof easy to scale.

Subtyping. Another type-side extension to look at is subtyping, especially its interaction with 1st / 2nd classes. As we showed in Section 3.5, 1st-class values can always be treated as 2nd-class, hence we can incorporate this conversion into a subtyping relation. It is also possible to allow conversion from $\bot$ to a value type, which, in essence, amounts to adopting the abort operator. However, this conversion must be explicit at the level of syntax, because otherwise the selective CPS translation may yield different output for the same expression depending the typing derivation.

Tail-Call Optimization. We can recognize and optimize source-level tail calls even after the CPS translation, via a simple syntactic analysis. For a given function call site, we need to check if all 2nd-class arguments (including the continuation) are also arguments of the caller. If this is the case, none of the arguments can have been allocated by the caller; they all must have existed before the caller stack frame was created. Therefore, we can shrink the stack and allow the callee to overwrite the caller stack frame. This check is similar to Kennedy [2007], but accounts for the fact that, in our approach, a tail call may need to pass freshly stack-allocated arguments, in which case we need to preserve the current stack frame.

6 CASE STUDY

6.1 Control-Based Language Features

Even in a language that does not provide continuations at the surface level, having 2nd-class continuations in the IR provides tangible benefits in supporting control features such as exceptions, `return` statements, and loops with `break` and `continue`. In particular, the combination with explicit 2nd-class functions enables some useful programming idioms. Consider the following `foreach` function for list traversal:

```
def foreach(xs: List[A])(@local f: A => Unit): Unit = {
  var xs1 = xs; while (xs1 != Nil) { f(xs1.head); xs1 = xs1.tail }
}
```

Observe that the argument `f` has an annotation `@local`, which is a Scala keyword for declaring 2nd-class values. Using `foreach`, we can build other useful functions such as `find`:

```
def find[A](xs: List[A])(@local p: A => Boolean) = {
  xs foreach (x => if (p(x)) return Some(x))
  return None
}
```

Invocations of both `find` and `foreach` never need to heap-allocate a closure for the argument functions, as they are guaranteed not to escape. Moreover, we can use an explicit `return` statement to exit `find` non-locally from within the function passed to `foreach`. This is safe because `return` is a 2nd-class function. Using `return` in a 1st-class context where it might escape is prohibited by the type system. This gain in expressiveness is due to exposing 2nd-class values at the user level, and is a direct benefit of separating non-returning and non-escaping aspects of continuations.

6.2 Recursive Join Points vs. Non-Local Returns

Let us compare this implementation with an example from Maurer et al. [2017]. Maurer et al. introduced *recursive join points* as a means to gain more optimization opportunities. For example,

in the program below, `find` is implemented using an auxiliary `go` function, which can be contified as a recursive join point:

```
def find[A](xs: List[A])(p: A => Boolean): Option[A] = {
  def go(xs: List[A]): Option[A] = xs match {
    case x :: xs => if (p(x)) Some(x) else go(xs)
    case Nil => None
  }
  go(xs)
}
```

This allows us to compile `go` into a jump, but more exciting optimizations are possible when `find` is used in the following way, as part of another function `any`:

```
def any[A](xs: List[A])(p: A => Boolean): Boolean = {
  find(xs)(p) match {
    case Some(_) => true
    case None => false
  }
}
```

When inlining `find`, the `match` expression within `any` can be cancelled by exploiting the join-point nature of `go`:

```
def any[A](xs: List[A])(p: A => Boolean): Boolean = {
  def go(xs: List[A]): Boolean = xs match {
    case x :: xs => if (p(x)) true else go(xs)
    case Nil => false
  }
  go(xs)
}
```

In our model, we can do the same, but we are also considerably more flexible. Let us implement `any` using our `foreach`-based `find` function:

```
def find[A](xs: List[A])(p: A => Boolean): Option[A] = {
  xs foreach (x => if (p(x)) return Some(x))
  return None
}
def any[A](xs: List[A])(p: A => Boolean): Boolean = {
  find(xs)(p) match {
    case Some(_) => true
    case None => false
  }
}
```

We first inline `find`, making the implementation of `return` explicit using $\mathscr{C}$:

```
def any[A](xs: List[A])(p: A => Boolean): Boolean = {
  (𝒞 \ (k => xs foreach (x => if (p(x)) k(Some(x))); k(None))) match {
    case Some(_) => true
    case None => false
  }
}
```

Then, we contract the $\mathscr{C}$ redex by absorbing the `match` expression into the continuation:

```
def any[A](xs: List[A])(p: A => Boolean): Boolean = {
  𝒞(k => xs foreach (x => if (p) k(true)); k(false))
```

```
}
```

Observe that we do not rely on an exposed structurally-recursive traversal; put differently, there is no need to inline `foreach`, or know anything about its implementation. Therefore, our approach works for any iterable structure (trees, hash maps, etc.) that has a `foreach` method, not just singly-linked lists.

7 EVALUATION

We implemented our approach in two very different compilers, the LMS runtime code generation and DSL compiler framework [Rompf and Odersky 2010], and the MiniScala Scala to native compiler used for teaching compiler construction at Purdue University. These implementations illustrate the flexibility of our approach and highlight how to adapt to the respective design goals and constraints.

7.1 LMS: Pattern Matching, Regexp and Automata

LMS (Lightweight Modular Staging) [Rompf and Odersky 2010] is a DSL compiler framework, embedded as a library in Scala. LMS features a graph-based IR. All intermediate results are given a name, but execution order is flexible up to control dependencies expressed in the graph [Rompf 2012]. User-defined rewrites in direct style are pervasive, and for end-to-end performance of user programs, such rewrites are even more important than in GHC. Likewise, direct-style optimizations such as common subexpression elimination and various code motion techniques are implemented as part of most LMS compiler pipelines. Since LMS emits source code (typically in Scala or C), performing a global CPS translation would be impractical.

We have extended LMS with a local control operator, as well as support for 2nd-class continuations that map to labels and goto statements in C. To use these new facilities, we have also modified a pre-existing regexp compiler [Rompf 2016; Rompf et al. 2013]. The original version would emit top-level C functions for each DFA state along with a driver loop (Figure 10, left). The new version emits local labels and gotos instead (Figure 10, right), leading to an overall 23% speedup on a mix of regex benchmarks.

7.2 MiniScala

MiniScala is a compiler from a restricted subset of Scala to x86 assembly. The initial implementation was based on the L^3 (Lisp-Like Language) compiler developed by Michel Schinz at EPFL. The intermediate representation is directly modeled after Kennedy's paper, including 2nd-class continuations and the overall optimization strategy, but without the union-find-based graph representation. We have replaced the IR with ours, while being faithful to Kennedy's design of performing optimizations after CPS. We have verified that optimization outcomes remain the same. But going beyond the old IR, we can now model 2nd-class functions that are not continuations, and even make them accessible at the surface level.

In this experiment, we used 2nd-class functions to replace heap allocations with stack allocations. We ported a series of benchmarks from the Computer Languages Benchmarks Game[4] and Scala Native[5]. The former set of benchmarks is compelling as it is used frequently in the literature [Adams et al. 2014; Gerakios et al. 2014; Kalibera et al. 2014; Morandat et al. 2012; Redondo and Ortin 2013]. The latter is also interesting because Scala Native shares much of the same goal as we do: supporting cheap allocation of objects on par with languages such as C and Rust, which compile to native code.

[4] https://benchmarksgame.alioth.debian.org/
[5] https://github.com/scala-native/scala-native

```
// generated code (old)
DFAState x13 = { x1, FALSE };
DFAState x17 = { x4, FALSE };
DFAState x10 = { x7, FALSE };
DFAState x12 = { x1, TRUE };
DFAState* x1(char x2) {
  if (x2 == 'A') return &x17;
  else           return &x13;
}
DFAState* x7(char x8) {
  if (x8 == 'A')      return &x10;
  else if (x8 == 'B') return &x12;
  else                return &x13;
}
DFAState* x4(char x5) {
  if (x5 == 'A') return &x10;
  else           return &x13;
}
DFAState* start = &x13;
// driver loop
int match(char *str) {
  DFAState* state = start;
  while (*str)
    state = state->func(*str++);
  return state->flag;
}
```

```
// generated code (new)
int match(char *str) {
  int flag = FALSE;
  char x2; // arg for x1
  char x8; // arg for x7
  char x5; // arg for x4

  // start: x1, false
  x1: if (!*str) return flag;
      x2 = *str++;
      if (x2 == 'A') {
        flag = FALSE; goto x4;
      } else {
        flag = FALSE; goto x1
      };

  x7: ... // elided

  x4: if (!*str) return flag;
      x5 = *str++;
      if (x5 == 'A') {
        flag = FALSE; goto x7;
      } else {
        flag = FALSE; goto x1
      };
}
```

Fig. 10. C code generated by LMS regexp matcher for regexp `'AAB'`

We did not change the algorithms from the reference implementation. Nevertheless, we sometimes had to change the code slightly to introduce 2nd-class functions. In each benchmark, we measured the amount of heap and stack memory consumed by the program, using a medium-sized input that makes the program run for a few seconds. The results are shown in Table 11. We see an asymptotically significant improvement in memory usage in the `pidigits` and `list` benchmarks. A particularly interesting fact is that, in the `pidigits` benchmark, the CPS translation enabled us to replace all dynamic allocations of big integers by allocation of 2nd-class arrays of integers. In contrast, the improvement is less remarkable in the `bounce` benchmark. This is because the code has variables stored in an array, as well as higher-order functions using those variables, which cannot be made 2nd-class. For the same reason, the `storage` benchmark shows no improvement at all.

8 INSTANTIATION CHOICES AND DISCUSSION

An appealing aspect of our IR is that it leaves many optimization choices open. In this section, we discuss various possible options and their respective advantages.

Naming Intermediate Results. We can easily change the IR to flatten out expressions and require all intermediate results to be named as in ANF. Assigning names is, of course, useful for referring to things. For example, many dataflow analyses rely on identifiers for expressions. In particular, common subexpression elimination and global value numbering are basically just hash-consing in a graph-based IR [Click 1995; Ershov 1958]. On the other hand, retaining nested expressions can be

	benchmark	heap (B/#)	stack (B/#)	% stack-alloc
Benchmarks Game	fannkuchredux	60588/15129	0	0
	fannkuchredux2	0	60588/15129	100
	pidigits	30852336/2302070	0	0
	pidigits2	500/33	30761576/2279472	99
Scala Native	bounce	24232/5557	0	0
	bounce2	4208/553	20024/5004	82
	list	22732/5655	0	0
	list2	0	42408/2144	100
	storage	139060/8192	0	0
	storage2	117220/6827	21844/1336	15
	towers	84/6	0	0
	towers2	0	84/6	100

Fig. 11. Memory profile for baseline and modified (suffix "2") benchmark programs. For both heap and stack memory we show the total allocation amount in bytes (B) and the number of allocations (#).

useful for simplicity. Instruction selection and register assignment can work very well in a setting where expression nesting defines lifetime.

Naming Control Points. We can also change the IR after CPS to require all continuations to be named. This is again useful in dataflow analysis, because named local continuations correspond to basic blocks and continuation arguments correspond to ϕ functions in SSA [Kelsey 1995]. On the other hand, the ability to use continuations anonymously can be beneficial as well. In particular, if a function or continuation is used anonymously, we know that it is used only in one place, without tracking all uses of an identifier in a program.

Direct-Style First, Then CPS. Since our IR supports both direct-style and CPS programs, it can be used to build either compiler front-ends that feed into an external downstrown compiler (as was the case with LMS in Section 7.1), or compiler back-ends based on dataflow analysis and low-level optimizations (as in MiniScala in Section 7.2). In our view, it is beneficial if optimizations can be done either before or after CPS, or both, in the same base IR.

9 RELATED WORK

Compiling with Continuations. The idea of using a CPS IR first appeared in the Rabbit compiler [Steele 1978], and was later incorporated into the Orbit compiler [Kranz et al. 1986a] with several refinements for improving memory usage. These early systems already recognized the need for distinguishing between lambdas representing functions and those representing continuations; the Orbit retrospective [Kranz et al. 1986b] contains an illuminating historical perspective. CPS IRs are also adopted into dialects of the SML language [Appel 1992; Kennedy 2007]. In particular, Kennedy [2007] solved the issue with additional redexes and expensive allocation by employing two-level λ abstractions and a special binder for continuations.

Compiling without Continuations. Meanwhile, the past decades have also seen various arguments for non-CPS IRs. Flanagan et al. [1993] showed that the effect of naïve CPS and accompanying simplification / optimization can be achieved by the ANF translation. As noted earlier, working in a direct-style IR is convenient for rewriting particular patterns of expressions, because the structure of the source program remains more or less the same. Also, leaving evaluation order unspecified is a key benefit when compiling lazy languages like Haskell. By adding join points in the style

of Maurer et al. [2017], we can further avoid code duplication that would result from careless treatment of case-like constructs.

Monadic intermediate languages [Benton et al. 1998; Benton 1993] are another popular choice for compilers. Like ANF, monadic IRs use let constructs to make control flow explicit, but unlike ANF, they allow nested let constructs, which are flattened via commuting conversions when necessary. Monadic IRs distinguish between pure values and effectful computations, and this distinction is used to decide whether it is safe to eliminate a let binding or an exception handler.

Compiling with Some Continuations. In our IR, we use $\mathscr{C}$ as a means to selectively expose continuations. For a similar purpose, the Moby compiler [Reppy 2001] uses a *local CPS conversion* to convert non-tail calls into CPS, improving the performance of nested loops. Another related device can be found in Sequent Core [Downen et al. 2016], an IR based on sequent calculus. Sequent Core has special binders for abstracting and creating continuations, and interestingly, they work more like our $\mathscr{C}$ operator than Maurer et al.'s join points. Specifically, when there are nested conditionals, the continuation binder pushes the outer conditional only into the definition of the join point. This is an advantage of being able to handle certain continuations explicitly.

1st- and 2nd-Class Values. Osvald et al. [2016] re-introduced 2nd-class functions into modern languages, and showed a number of practical applications such as capabilities and co-effects. They also formally proved that 2nd-class values follow a strict stack discipline, and conjectured that this could make them cheaper to implement, although their system, implemented as a compiler plug-in for Scala, does not provide 2nd-class values at the level of a compiler IR.

10 CONCLUSIONS

Over the past decades, the debate on "CPS or not CPS" has attracted great attention in the programming languages community, exploring various perspectives on compiler design. In this paper, we claim that the right question to ask is "how much CPS would you like", and propose an IR in which one can compile with an arbitrary number of continuations, using a carefully designed control operator and an optional CPS translation.

With the ability to perform direct-style and CPS optimizations in the same IR, the compiler writer can make design decisions in a more flexible way, possibly changing those decisions in a later phase of development. We intend to explore the benefits of this flexibility when combined with language features and optimizing transformations we did not address in this paper. We also hope that our proposal will stimulate further advances in the CPS debate.

ACKNOWLEDGMENTS

We gratefully acknowledge the anonymous reviewers for their valuable feedback, which improved the paper in various ways. This work was supported in part by NSF awards 1553471 and 1564207, DOE award DE-SC0018050, as well as gifts from Google, Facebook, and VMware.

REFERENCES

Keith Adams, Jason Evans, Bertrand Maher, Guilherme Ottoni, Andrew Paroski, Brett Simmers, Edwin Smith, and Owen Yamauchi. 2014. The HipHop Virtual Machine. In *Proceedings of the 29th ACM International Conference on Object-Oriented Programming Systems Languages, and Applications (OOPSLA '14)*. ACM, New York, NY, USA, 777–790. https://doi.org/10.1145/2660193.2660199

Andrew W. Appel. 1992. *Compiling with Continuations.* Cambridge University Press.

Andrew W. Appel. 1998. SSA is Functional Programming. *SIGPLAN Notices* 33, 4 (1998), 17–20.

Kenichi Asai and Chihiro Uehara. 2018. Selective CPS transformation for shift and reset. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation.* ACM, 40–52.

Nick Benton, Andrew Kennedy, and George Russell. 1998. Compiling Standard ML to Java Bytecodes. In *ICFP*. ACM, 129–140.

Peter Nicholas Benton. 1993. *Strictness Analysis of Lazy Functional Programs*. Ph.D. Dissertation. University of Cambridge.

Cliff Click. 1995. Global code motion/global value numbering. *ACM Sigplan Notices* 30, 6 (1995), 246–257.

Oliver Danvy and Andrzex Filinski. 1992. Representing control: A study of the CPS transformation. *Mathematical structures in computer science* 2, 4 (1992), 361–391.

Paul Downen, Luke Maurer, Zena M. Ariola, and Simon Peyton Jones. 2016. Sequent calculus as a compiler intermediate language. In *ICFP*. ACM, 74–88.

Andrei P. Ershov. 1958. On programming of arithmetic operations. *Commun. ACM* 1, 8 (1958), 3–6.

Matthias Felleisen, Daniel P Friedman, Eugene Kohlbecker, and Bruce Duba. 1987. A syntactic theory of sequential control. *Theoretical computer science* 52, 3 (1987), 205–237.

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *PLDI*. ACM, 237–247.

Prodromos Gerakios, Nikolaos Papaspyrou, and Konstantinos Sagonas. 2014. Static safety guarantees for a low-level multithreaded language with regions. *Science of Computer Programming* 80 (2014), 223–263. https://doi.org/10.1016/j.scico.2013.06.005

Jean-Yves Girard, Paul Taylor, and Yves Lafont. 1989. *Proofs and types*. Vol. 7. Cambridge University Press.

Tomas Kalibera, Petr Maj, Floréal Morandat, and Jan Vitek. 2014. A fast abstract syntax tree interpreter for R. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '14)*. ACM, New York, NY, USA, 89–102. https://doi.org/10.1145/2576195.2576205

Richard Kelsey. 1995. A Correspondence between Continuation Passing Style and Static Single Assignment Form. In *Intermediate Representations Workshop*. ACM, 13–23.

Andrew Kennedy. 2007. Compiling with continuations, continued. In *ICFP*. ACM, 177–190.

David A. Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, and James Philbin. 1986a. ORBIT: an optimizing compiler for scheme. In *SIGPLAN Symposium on Compiler Construction*. ACM, 219–233.

David A. Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. 1986b. Orbit: an optimizing compiler for scheme (with retrospective). In *Best of PLDI*. ACM, 175–191.

Luke Maurer, Paul Downen, Zena M. Ariola, and Simon L. Peyton Jones. 2017. Compiling without continuations. In *PLDI*. ACM, 482–494.

Floréal Morandat, Brandon Hill, Leo Osvald, and Jan Vitek. 2012. Evaluating the Design of the R Language: Objects and Functions for Data Analysis. In *ECOOP 2012 — Object-Oriented Programming: 26th European Conference. Proceedings (Lecture Notes in Computer Science)*. Springer Berlin Heidelberg, Germany, 104–131.

Lasse R Nielsen et al. 2001. A selective CPS transformation. *Electronic Notes in Theoretical Computer Science* 45 (2001), 311–331.

Chris Okasaki, Peter Lee, and David Tarditi. 1994. Call-by-need and continuation-passing style. *Lisp and Symbolic Computation* 7, 1 (1994), 57–81.

Leo Osvald, Grégory M. Essertel, Xilun Wu, Lilliam I. González Alayón, and Tiark Rompf. 2016. Gentrification gone too far? affordable 2nd-class values for fun and (co-)effect. In *OOPSLA*. ACM, 234–251.

Simon L. Peyton Jones and Simon Marlow. 2002. Secrets of the Glasgow Haskell Compiler inliner. *J. Funct. Program.* 12, 4&5 (2002), 393–433.

Jose M. Redondo and Francisco Ortin. 2013. Efficient support of dynamic inheritance for class- and prototype-based languages. *Journal of Systems and Software* 86, 2 (2013), 278–301. https://doi.org/10.1016/j.jss.2012.08.016

John Reppy. 2001. Local CPS conversion in a direct-style compiler. In *Proceedings of the Third ACM SIGPLAN Workshop on Continuations (CW '01)*. 13–22.

Tiark Rompf. 2012. *Lightweight Modular Staging and Embedded Compilers: Abstraction Without Regret for High-Level High-Performance Programming*. Ph.D. Dissertation. EPFL. https://doi.org/10.5075/epfl-thesis-5456

Tiark Rompf. 2016. The Essence of Multi-stage Evaluation in LMS. In *A List of Successes That Can Change the World (Lecture Notes in Computer Science)*, Vol. 9600. Springer, 318–335.

Tiark Rompf, Ingo Maier, and Martin Odersky. 2009. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In *ACM Sigplan Notices*, Vol. 44. ACM, 317–328.

Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Conference on Generative programming and component engineering (GPCE)*. 127–136. https://doi.org/10.1145/1868294.1868314

Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. 2013. Optimizing Data Structures in High-Level Programs *(POPL)*.

Guy L. Steele, Jr. 1978. *Rabbit: A compiler for Scheme*. Technical Report. Massachusetts Institute of Technology.

Andrew K Wright and Matthias Felleisen. 1994. A syntactic approach to type soundness. *Information and computation* 115, 1 (1994), 38–94.