Compiling with Continuations, or without? Whatever.

Youyou Cong[1,2], Leo Osvald[2], Grégory Essertel[2], Tiark Rompf[2]
[1]Ochanomizu University, [2]Purdue University

Abstract

What makes a good compiler IR? In the context of functional languages, there has been an extensive debate about the merits and drawbacks 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, separately from regular functions, to ensure efficient code generation.

In particular, the most recent contribution to this debate from PLDI 2017 proposes an explicit join-point construct to model labels and jumps in a direct-style IR. We develop this idea further and propose an IR based on exposing continuations *selectively* through a control operator, similar to call/cc, but suitably restricted for a compiler IR. We further propose a type system that tracks no-escape and no-return behavior, serving as a basis for a type-directed CPS transform that maintains these properties as well as the call-stack behavior of direct style. Our IR supports both direct style and full CPS, as well as any number of selective continuations in-between. Hence, we change the fundamental question from "CPS yes or no" to "as much or as little CPS as you like".

1 Introduction

The year 2018 marks the 40th anniversary of Steele's Rabbit compiler [31], the first compiler for Scheme. Rabbit was influential in demonstrating that a lexically-scoped dialect of LISP could be compiled efficiently, and therefore it was an existence proof that λ-calculus with scoping rules as in the mathematical formalism is a useful foundation for practical programming languages. Rabbit also pioneered the use of a low-level intermediate language based on Scheme in continuation-passing-style (CPS), which maps relatively closely to machine language. In the intervening forty years, there has been an extensive debate about the relative merits and drawbacks of CPS in intermediate languages.

The Great CPS Debate Rabbit inspired much follow-on work, including Orbit [17] and Appel's SML/NJ compiler, the latter documented at great length in his influential Book "Compiling with Continuations" [2]. CPS makes all aspects of control flow and evaluation order explicit and uniform. In terms of optimizations, CPS renders unrestricted $\beta\eta$-reduction sound, which is not sound in direct-style representations of call-by-value languages. Many optimizations can be described as well-behaved sequences of β and η steps, often with *shrinking* guarantees. While all function calls become jumps in CPS, the downside of uniformity is that certain specialized information from the source program is no longer explicit and has to be recovered by sophisticated analysis. In particular, avoiding allocation of closure records for continuations that represent normal jumps is of paramount concern.

To address the downsides of CPS, including complexity and the need for additional analysis and simplification, Flanagan et al. [10] proposed to replace CPS intermediate languages with Administrative Normal Form (ANF), which makes the evaluation order explicit through let-binding all intermediate values but leaves the program otherwise in direct style. ANF is closely related to monadic intermediate languages based on Moggi's computational λ-calculus [20]. While many CPS optimizations carry over to ANF, the downside of not providing a facility to model jumps is that some optimizations that are natural in CPS have no obvious equivalent. Key examples are the conversion of recursive functions to native loops and the representation of *join points*, which arise in nested conditionals, pattern matching, and short-circuit boolean operators. These join points need to be represented as regular lambdas in ANF, again requiring additional analysis to ensure efficient code generation.

As a remedy, and based on the observation that both proper functions and continuations are needed in an IR and that they should not be confused, Kennedy [16] proposed to go back to CPS but in a very specific way that imposes a syntactic distinction between normal functions and continuations, ensuring that continuations behave in a restricted 2nd-class way that allows efficient code generation as jumps. This works very well for join points and similar optimizations. However, if this is the only IR on which optimizations are performed, some are harder to do than in direct style. In particular for pure and non-strict languages, fixing the evaluation order is an obstacle for common-subexpression elimination, code motion, as well as for high-level user-defined rewrites that may match on nested function applications.

To address this final issue, Maurer et al. [19] propose to go back to direct style but add a special form of functions to model jumps, uniformly called join points. Calls to join points are jumps, and join points "return" to their scope of declaration. Thus, there is no need to CPS-transform normal code if the goal is to perform commuting conversions, contification, etc., which can all be performed on this direct-style IR extended with join points.

Is this the End of the Story? Almost, except for perhaps one detail: while Maurer et al. [19]'s join points are similar to standard continuations, they behave differently in subtle ways that makes them in our view less ideally suited for certain optimizations. Instead, we propose to add standard 2nd-class continuations to a direct-style IR, with optional CPS conversion that preserves all desirable properties.

Preprint, November 2018.

Looking Back, what Have we Learned? Before we go into details about our proposed approach we would like to settle the great “CPS good or bad” debate once and for all with a decisive “it depends!”

Instead of looking for a universal truth, we believe that more nuanced recommendations for compiler writers are needed. In that sense, “CPS good or bad” 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 conversion 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 CFG-based IR on which additional passes of dataflow analysis, register allocation, etc., are run. It is widely appreciated that CFG-based IRs in SSA-form are isomorphic to CPS [3, 14]. Even in the setting of Maurer et al. [19] who reject CPS as high-level IR, GHC compiles to `C--` or LLVM, effectively performing CPS conversion as part of this lowering. Hence, for a particular optimization, “before vs. after CPS” seems to be a more appropriate question, and the more choice an IR provides the better.

Contributions We believe that the essence of “Compiling without Continuations” [19] 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 and somewhat ad-hoc language construct to model join points; instead, we can use a well-understood facility to expose only some selected continuations: *control operators*. That is, we want something like `call/cc` in our IR, 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 similary to `call/cc` (as the paper notes in passing [19]) in that the programs silently continues after the `call/cc` block even if the continuation is never invoked. We show that a slightly different control operator, closer to Felleisen’s $\mathscr{C}$ [9] and without the silent fall-through property, is more suited for an IR and solves a crucial code duplication issue with Maurer-style join points. This code duplication issue, which we describe in Section 2, is all the more troubling, as joint points are meant to *solve* code duplication issues in the first place.

Control operators are erased during CPS translation. More specifically, we propose a *selective* CPS translation that only manipulates parts of the program where continuations are implicit. This allows us to represent the same program before and after CPS conversion in the same IR. The CPS conversion maintains all desired properties, including 2nd-class nature of continuations and a stack-based call discipline. The key benefit is that programs can be optimized in direct style, in full CPS, and anywhere in-between with only some continuations made explicit.

While it is entirely possible to follow Kennedy and Maurer et al. in distinguishing continuations and functions syntactically, we propose to make this distinction in the type system. Clearly, we can introduce separate types for continuations and functions, like in the typing of `call/cc` in SML [12]. But we find it especially useful to separate out two aspects of continuations: first, their *non-returning* behavior, and second, their *non-escaping property*. Treating these two aspects separately enables us to also model non-escaping regular functions (and other values) in our IR. These can be allocated on the stack, further reducing closure allocation, which is a main source of inefficiency in functional languages. Moreover, such non-escaping 2nd-class values can serve as temporary access tokens and form the basis for co-effect systems to model checked exceptions and other effects [22].

To sum up, we characterize our key ideas as follows:

- Control operator as introduction form for current continuation, translated away during CPS transform
- No-return functions and no-escape values identified in the type system, modeled as orthogonal concepts

Our approach has the following desirable properties:

- Similar lightweight design, and supports all optimizations of Maurer-style join points
- Grounded firmly in well-understood theory (control operators and CPS transform)
- Practical benefits w.r.t. code duplication (less other compiler magic required to get desired results)
- Same IR before and after CPS, preserve all relevant properties (especially stack vs. heap allocation, and call-stack discipline). Pick and chose when to perform optimizations
- Easily scale to non-local returns, exceptions, and if desired, 1st-class and delimited continuations
- Benefits of 2nd-class regular functions for stack allocation, and beyond (co-effect-systems)

The paper is structured as follows:

- We discuss our motivation and key ideas (Section 2).
- We present the formal development: syntax, type system, optimizations (Section 3), selective CPS transform (Section 4), extensions (Section 5).
- We evaluate our approach qualitatively based on examples (Section 6) and quantitatively based on experiments with two very different compilers (Section 7).
- We discuss specific instantiation choices (Section 8) and related work (Section 9).

2 Motivation and Key Ideas

Representing Join Points One of the recurring questions discussed in the CPS debate is how to optimize nested conditionals and case analysis [16, 19]. These occur frequently in intermediate forms of code after inlining but also directly as the result of desugaring short-circuit boolean operators:

```
// source expression (0)
if (e1 && e2) then e4 else e5
// desugaring (1)
if (if e1 then e2 else false) then e4 else e5
// commuting conversion (2)
if e1 then (if e2 then e4 else e5)
       else (if false then e4 else e5)
// simplification (3)
if e1 then (if e2 then e4 else e5) else e5
```

Observe that the commuting conversion exposes a new redex `if false ...`, giving us an additional simplification opportunity. The resulting program is however not optimal, in that it has duplicated occurrences of the branch `e5`. This motivates us to introduce a *join point*, by defining `j5` as a function that computes `e5` when called:

```
// direct style, regular function as join point
def j5() = e5
if e1 then (if e2 then e4 else j5()) else j5()
```

This solves the code duplication issue, but the program is still unsatisfactory, because the join point will be heap-allocated as a closure. For this reason, Kennedy [16] proposed to perform optimizations in CPS programs, where continuations are bound by a distinct binder (`defcont` in the program below) and are given a *2nd-class* status (i.e., no escaping ability):

```
// continuation-passing style
defcont k(x)   = ...
defcont j4()   = k(e4); defcont j5() = k(e5)
defcont k2(x2) = if x2 then j4() else j5()
defcont j2()   = k2(e2)
defcont k1(x2) = if x1 then j2() else j5()
defcont j1()   = k1(e1)
```

On the other hand, CPS may limit applicable simplifications in certain circumstances, especially high-level rewrites of the sort that operate on nested expressions with multiple function invocations, e.g., to rewrite `xs.map(f).map(g)` into `xs.map(g∘f)`. To achieve efficiency without losing such optimization opportunities, Maurer et al. [19] augment their IR with a new binding construct for join points (`defjoin` below), which allows the compiler to represent this pattern in a direct-style IR, such as the IR of the Glasgow Haskell Compiler (GHC) [23]:

```
// direct style, explicit join point
defjoin j5() = e5
if e1 then (if e2 then e4 else j5()) else j5()
```

As we now treat join points differently from functions, we have to make sure that it is always safe to regard them as mere jumps. This means that `defjoin`-bound variables cannot be captured by functions, and must be tail-called in a fully applied form, just as with `defcont`.

Our approach can be thought of as a combination of the two solutions discussed above: we define join points by means of Kennedy-style 2nd-class continuations, but at the same time, we allow direct-style optimizations in the style of Maurer et al. What makes this possible is a *control operator*, as well as a type system that can express non-returning computations and non-escaping values:

```
// control operator with selective CPS translation
𝒞 { k =>
  defcont j5() = k(e5)
  if e1 then (if e2 then k(e4) else j5()) else j5()
}
```

Operationally, the $\mathscr{C}$ operator captures and clears the entire surrounding context, and binds the variable `k` to a functional representation of the captured context. Typewise, we classify `k` as a 2nd-class value, and assign it a type of the form $T \to \bot$, where $\bot$ is an empty type with no inhabitant. 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 may only be used as jumps.

As we mentioned earlier, $\mathscr{C}$ can be erased via a selective CPS translation. This opens the possibility to first simplify programs in direct style, and then apply further optimizations in CPS. Also, CPS programs are guaranteed to run efficiently, thanks to the 2nd-class status of continuations.

Optimizing and Preserving Join Points In the simple cases like the above example, the $\mathscr{C}$-based optimization works almost the same way as Maurer et al.'s approach. Now, we look at a more complex example from Maurer et al.'s paper, where our proposed IR allows for a simpler optimization process. Consider the following source program which already has a join point inserted, but in a nested position:

```
// source expression
{ 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 join points were treated like regular functions, commuting conversion would lead to the following program that is problematic in two ways. First, `j` is not tail-called, hence it cannot be compiled as a join point. Second, the case expressions inspect an application of `j`, which means we cannot apply any further optimization.

```
// undesired transformation (join point destroyed)
def 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
}
```

What Maurer et al. propose instead is the following:

```
// desired transformation (join point preserved)
def 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 case expression inspects `BIG`, which is potentially a data constructor or a case expression. However, it is not obvious how to arrive at this transformation and others in the general case.

What Is the Problem? What exactly makes the optimization hard to apply? The answer becomes clear when we look at the proposed optimization rules:

```
// Maurer et al. optimization rules
E[ defjoin j(x) = e1; e2 ] = defjoin j(x) = E[e1]; E[e2] (jfloat)
E[ j(x) ] = j(x) (jmp) // if j is a join point
```

In Maurer et al.'s system, the `defjoin` construct is manipulated via the (jfloat) rule, which wraps both the definition `e1` and the body `e2` around the context `E`. If `e2` jumps to `j`, evaluation finishes with `E[e1]`, and if `e2` returns normally, evaluation finishes with `E[e2]`. Readers familiar with control operators might have noticed that the rule is similar to the reduction rule of `call/cc`. This optimization rule, however, yields a program that is again problematic in two ways – much like the one declared as undesired above!

```
// (jfloat) yields:
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 }
```

First, `j` is not tail-called and hence is no longer obviously a join point. Second, the `A` and `B` branches scrutinize a join-point application, which cannot be a datatype constructor, therefore the copied pattern matching context only makes the program bigger. In addition, this intermediate step needs to be well-typed, so the typing rules for join points are quite complex. To obtain what we want, we need to transform the `A` and `B` cases back to `j(1)` and `j(2)` through subsequent applications of (jmp). Moreover, we have to carefully analyze the program to decide whether applying (jfloat) in the first place may be beneficial.

Solution Our control operator has a similar, but slightly different, semantics. This yields an optimization rule that gives us the desired program in a more straightforward way:

```
// our optimization rule:
E[ 𝒞 { k => t }) ] = 𝒞 { j => defcont k(x) = j(E[ x ]); t } (cfloat)
```

In our representation of the example, $\mathscr{C}$ is already used to represent the join point:

```
// source expression
{ 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 }
```

And we obtain the following optimized version:

```
// (cfloat) yields:
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 })
} }
```

Observe that the (cfloat) rule does not copy the captured continuation to the body `t`, since the $\mathscr{C}$ operator requires all uses of `k` to be explicit. This allows us to inject the outer context *when necessary*, instead of injecting it everywhere and then removing it later. In the above example, we *only* need to look at `k(BIG)` and `k(true)` to decide whether it makes sense to pull the outer `case` into the $\mathscr{C}$ block – we can completely ignore the other case branches `j(1)` and `j(2)`. In constrast, Maurer et al.'s formulation needs to consider all possible future opportunities for (jmp) when deciding where to apply (jfloat). Thus, we obtain the expected result more reliably and with less effort.

Furthermore, since we distinguish between escaping and non-escaping values based on their types (instead of syntax), we can stack-allocate both continuations and any user-defined, non-escaping functions. This gives us optimization opportunities that are not available in Maurer et al.'s system.

3 Formal Development

We now describe our IR, which we call $\lambda_{\perp}^{1/2}$. Before going into detail, let us summarize three key ideas of the language, which we roughly discussed in the previous section:

- To account for the non-returning nature of continuations, we incorporate an empty type $\perp$, and assign continuations a type of the form $T \rightarrow \perp$. This allows us to identify jumping terms by looking at their type.
- To efficiently compile continuations, we distinguish between 1st- and 2nd-class values, and classify continuations as 2nd-class. Roughly speaking, 1st-class values can escape, whereas 2nd-class values cannot. Formally, the classification is defined by two rules shown in Section 3.3.
- To simplify optimization steps, we use a control operator $\mathscr{C}$ whose reduction inserts no implicit call of the captured continuation. That means, our operator resembles Felleisen's $\mathscr{C}$ [9], instead of `call/cc` (which has a close connection to (jfloat) of Maurer et al. [19]):

$$E[\mathscr{C}\ (\lambda k.t)] = t[\lambda x.\text{abort}\ (E[x])/k]$$
$$E[\text{callcc}\ (\lambda k.t)] = E[t[\lambda x.\text{abort}\ (E[x])/k]]$$

Syntax

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

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

Values and Evaluation Contexts

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

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

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

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

3.1 Syntax

In Figure 1, we present the syntax of $\lambda_\bot^{1/2}$. The term language includes ordinary λ-terms as well as constants (of base type), pairs, `if`, `let`, and the $\mathscr{C}$ operator. Some of these language constructs have a class annotation: e.g., the annotation on variables tells us which class of value they denote, and those on pairs carry the class information of their first and second elements. Types consist of base types B and arrow types $T^m \to U$. The domain again has a class annotation, and the co-domain may be the empty type $\bot$, in which case the arrow type is inhabited by non-returning functions.

3.2 Operational Semantics

The $\lambda_\bot^{1/2}$ language has a call-by-value semantics, which is shown in Figure 2. Note that value classes do not affect the high-level behavior of programs, so we omit them in the reduction rules. Most rules are completely standard, except the last one: when the argument of $\mathscr{C}$ has evaluated to a function $\lambda k.t$, we evaluate the body t in an empty context $\square$, with k bound to the continuation $\lambda x.E[x]$ surrounding the $\mathscr{C}$ construct. This rule is slightly different from that

Type Environments

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

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

$$G \vdash c :^n B \qquad (\textsc{TCst})$$

$$\frac{x^m : T \in G^{[\le n]}}{G \vdash x^m :^n T} \qquad (\textsc{TVar})$$

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

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

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

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

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

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

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

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

Figure 3. $\lambda_\bot^{1/2}$: Type System

of Felleisen's $\mathscr{C}$, in that the captured continuation is not surrounded by an abort operator. However, we can still rule out returning continuations by exploiting the $\bot$ type, as we will see in the next subsection.

3.3 Type System

The type system of $\lambda_\bot^{1/2}$ (Figure 3) assigns both types and 1st vs. 2nd classes to terms. One of the key rules is (TAbs), which encodes two restrictions on value classification:

- Functions may only return a 1st-class value
- 1st-class functions should not refer to 2nd-class values through free variables

These restrictions are imposed by requiring a 1st-class body t, and by constraining variable occurrences via the $G^{[\le n]}$ operation.

Optimizations $\boxed{t = t'}$

$$
\begin{aligned}
(\lambda x^m.t)\ t_2 &= (\mathsf{let}\ x^m = t_2\ \mathsf{in}\ t) \\
\mathsf{fst}\ (v_1\ {}_{n_1,n_2}\ v_2) &= v_1 \\
\mathsf{snd}\ (v_1\ {}_{n_1,n_2}\ v_2) &= v_2 \\
\mathsf{if\ true\ then}\ t_2\ \mathsf{else}\ t_3 &= t_2 \\
\mathsf{if\ false\ then}\ t_2\ \mathsf{else}\ t_3 &= t_3 \\
(\mathsf{let}\ x^m = v\ \mathsf{in}\ t) &= t[v/x^m] \\
E[\mathscr{C}(\lambda k^2.t)] &= \mathscr{C}(\lambda j^2.\mathsf{let}\ k^2 = \lambda x^n.j^2\ E[x^n]\ \mathsf{in}\ t) \\
&\quad \text{if } E \text{ has an } n\text{-class hole} \\
\mathscr{C}(\lambda k^2.k^2\ t) &= t
\end{aligned}
$$

Figure 4. Optimizations

Another rule we would like to elaborate on is (TCtrl). It says, the argument of $\mathscr{C}$ must be a non-returning function taking in a 2nd-class continuation. The second occurrence of $\bot$ makes it impossible to return the captured continuation to the top level, which is exactly what we expect.

The type system allows continuations to be used only as jumps. This is guaranteed by constraining occurrences of non-returning terms in the typing rules. Notice that, any subterm that can be a hole of an evaluation context – such as the function and argument parts of an application, and the definition part of a `let` expression – must have a non-$\bot$ type (i.e., a value type T). Under this principle, all we can do with a continuation application is to return the value as the final result; we cannot pass the value to some other function, or `let`-bind the value for future use.

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 of $\mathscr{C}$: here, E is one single context frame, such as $\square\ t$ or $v\ \square$, and we identify k^2 with the composition of E and the outer context frames j^2. As noted above, the floating rule does not duplicate the captured context, without necessitating additional rewrites needed in the language of Maurer et al. [19]. The other rule for $\mathscr{C}$ represents idempotency: capturing and immediately applying the continuation has no computational effect.

4 CPS Translation

As we saw in the previous sections, many optimizations can be fruitfully performed in direct style with join points and the $\mathscr{C}$ operator. However, depending on the circumstances, lowering to CPS after direct-style optimizations are exhausted may still be an important step (see Section 8 for further discussion). In that case, we must make sure that the result after the CPS translation is as efficient as Kennedy's; in particular, we want to compile continuations to jumps, instead of heap-allocating them.

Fortunately, the type system helps us CPS translate programs in a desired manner. Since returning and non-returning functions, as well as escaping and non-escaping values are cleanly separated, we can use the type system to guide a *selective* CPS translation. The idea is to leave $\mathscr{C}$-captured continuations as is, and make all implicit continuations explicit. A key observation is that, using the value class distinction, we can guarantee the desired stack behavior of CPS programs. Thus, our system allows early levels of the compiler, and even the source program, to deal as little or as much with continuations as desired (by means of $\mathscr{C}$ and non-escaping functions), while being sure that the generated code corresponds to what a low-level, 2nd-class CPS translation in the style of Kennedy would produce.

4.1 The Translation

In Figures 5 and 6, we show the selective CPS translation of $\lambda_\bot^{1/2}$. Notice that there are two kinds of translation: $[\![t]\!]$ and $[\![t]\!]'$. The former turns jump-free terms (of type T) into CPS, whereas the latter manipulates jumping terms (of type $\bot$) without introducing a continuation.

Let us go through individual rules. Constants and variables are translated the same way as in a non-selective, call-by-value translation. Abstractions have two rules: the first rule is the standard, non-curried translation, while the second one uses the non-CPS translation for the body t. Correspondingly, we have different translations for function types $T_1^n \to T_2$ and $T^n \to \bot$: the former has two extra arrows in the image of the translation, whereas the latter does not. The translation of $T_1^n \to T_2$ further helps us maintain an invariant: the return continuation of a function always receives a 1st-class argument. This intuitively makes sense, because every function has a 1st-class body. As we will see in Section 4.2, the invariant guarantees a preferable stack behavior of CPS programs.

Similarly to abstraction, we have two rules for translating application. The first rule is used for ordinary function calls. Notice that the return continuation k^2 has been η-expanded. This trick is needed to force the continuation passed to v_1^2 to have type $[\![T_2]\!]^1 \to \bot$. That is, if k^2 is a continuation expecting a 2nd-class value (which happens when the source application is 2nd-class), we wrap it around a lambda binding x^1, and use x^1 as a 2nd-class value in the application $k^2\ x^1$ (which is type safe). The second rule of application is used for jumps of type $\bot$. The result does not take a continuation, and has the same type $\bot$ as the original application.

The translation of $\mathscr{C}t$ feeds the continuation-demanding value v^2 (which represents the result of evaluating t) with the top-level continuation k^2. Note that v^2 does not require a return continuation, because it is a $\bot$-returning function.

Term Translation $\boxed{\llbracket t \rrbracket, \llbracket t \rrbracket'}$

$$\begin{array}{rcll}
\llbracket c \rrbracket &=& \lambda k^2.k^2\ c & \\
\llbracket x^m \rrbracket &=& \lambda k^2.k^2\ x^m & \\
\llbracket \lambda x^m.t \rrbracket &=& \lambda k^2.k^2\ (\lambda (x^m\ {}_{m,2}\, k_1^2)^2.\llbracket t \rrbracket\ k_1^2) & \text{if } t : T \\
\llbracket \lambda x^m.t \rrbracket &=& \lambda k^2.k^2\ (\lambda x^m.\llbracket t \rrbracket') & \text{if } t : \bot \\
\llbracket t_1\ t_2 \rrbracket &=& \lambda k^2.\llbracket t_1 \rrbracket\ (\lambda v_1^2.\llbracket t_2 \rrbracket\ (\lambda v_2^m.v_1^2\ (v_2^m\ {}_{m,2}\, (\lambda x^1.k^2\ x^1)))) & \text{if } t_1 : T_1^m \to T_2 \\
\llbracket t_1\ t_2 \rrbracket' &=& \llbracket t_1 \rrbracket\ (\lambda v_1^2.\llbracket t_2 \rrbracket\ (\lambda v_2^m.v_1^2\ v_2^m)) & \text{if } t_1 : T_1^m \to \bot \\
\llbracket (t_1\ {}_{n_1,n_2}\, t_2) \rrbracket &=& \lambda k^2.\llbracket t_1 \rrbracket\ (\lambda v_1^{n_1}.\llbracket t_2 \rrbracket\ (\lambda v_2^{n_2}.k^2\ (v_1^{n_1}\ {}_{n_1,n_2}\, v_2^{n_2}))) & \\
\llbracket \mathsf{fst}\ t \rrbracket &=& \lambda k^2.\llbracket t \rrbracket\ (\lambda v^{n_1}.k^2\ v^{n_1}) & \text{if } t : T_1^{n_1} \times T_2^{n_2} \\
\llbracket \mathsf{snd}\ t \rrbracket &=& \lambda k^2.\llbracket t \rrbracket\ (\lambda v^{n_2}.k^2\ v^{n_2}) & \text{if } t : T_1^{n_1} \times T_2^{n_2} \\
\llbracket \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \rrbracket &=& \lambda k^2.\llbracket t_1 \rrbracket\ (\lambda v_1^2.\mathsf{if}\ v_1^2\ \mathsf{then}\ \llbracket t_2 \rrbracket k^2\ \mathsf{else}\ \llbracket t_3 \rrbracket k^2) & \text{if } t_2, t_3 : T \\
\llbracket \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \rrbracket' &=& \llbracket t_1 \rrbracket\ (\lambda v_1^2.\mathsf{if}\ v_1^2\ \mathsf{then}\ \llbracket t_2 \rrbracket'\ \mathsf{else}\ \llbracket t_3 \rrbracket') & \text{if } t_2, t_3 : \bot \\
\llbracket \mathsf{let}\ x^m = t_1\ \mathsf{in}\ t_2 \rrbracket &=& \lambda k^2.\llbracket t_1 \rrbracket\ (\lambda v_1^m.\mathsf{let}\ x^m = v_1^m\ \mathsf{in}\ \llbracket t_2 \rrbracket\ k^2) & \\
\llbracket \mathsf{let}\ x^m = t_1\ \mathsf{in}\ t_2 \rrbracket' &=& \llbracket t_1 \rrbracket\ (\lambda v_1^m.\mathsf{let}\ x^m = v_1^m\ \mathsf{in}\ \llbracket t_2 \rrbracket') & \\
\llbracket \mathscr{C} t \rrbracket &=& \lambda k^2.\llbracket t \rrbracket\ (\lambda v^2.v^2\ k^2) &
\end{array}$$

Figure 5. CPS Translation on Terms

Type Translation $\boxed{\llbracket T \rrbracket}$

$$\begin{array}{rcl}
\llbracket B \rrbracket &=& B \\
\llbracket T_1^n \to T_2 \rrbracket &=& (\llbracket T_1 \rrbracket^n \times (\llbracket T_2 \rrbracket^1 \to \bot)^2)^2 \to \bot \\
\llbracket T^n \to \bot \rrbracket &=& \llbracket T \rrbracket^n \to \bot \\
\llbracket T_1^{n_1} \times T_2^{n_2} \rrbracket &=& \llbracket T_1^{n_1} \rrbracket \times \llbracket T_2^{n_2} \rrbracket
\end{array}$$

Figure 6. CPS Translation on Types

As we can see from these rules, our CPS translation leaves administrative redexes, but they can be eliminated during the translation using standard techniques [7].

An important property that holds of our CPS translation is that it preserves typing, i.e., well-typed programs are converted to well-typed programs.

Proposition 4.1.

1. *If* $G \vdash t :^n T$, $\llbracket G \rrbracket \vdash \llbracket t \rrbracket :^n (\llbracket T \rrbracket^n \to \bot)^2 \to \bot$.
2. *If* $G \vdash t :^n \bot$, $\llbracket G \rrbracket \vdash \llbracket t \rrbracket' :^n \bot$.

Proof. By induction on the structure of t. □

4.2 Properties of CPS Programs

Having defined a CPS translation, we next observe the stack behavior of CPS programs. 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 less clear because there is no distinction between calls and returns. Compared to Kennedy, our situation is more complicated: we do not identify continuations syntactically, and furthermore, we may have user-defined non-returning functions that do not need heap allocation, or 2nd-class functions taking in other 2nd-class functions.

Fortunately, a simple observation guides our model. When we call a function (in CPS) that receives only 1st-class arguments, the arguments cannot be stored on the stack, nor can they 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, when jumping to the function's body, we can shrink the stack by removing any bindings introduced after the function was defined. Note that this adjustment is safe only if the function does not return, which is always the case in the post-CPS world.

By contrast, calling a function that receives at least one 2nd-class argument means that the argument is stored somewhere on the stack, and potentially refers to any stack-stored values that are visible to the caller. In this case, we need to grow the stack and push the argument to the current address.

In our language, continuations are classified as 2nd-class functions, therefore we may use more stack space in a CPS program than its direct-style counterpart. However, the difference must be a constant factor, because the CPS translation introduces at most one continuation per language construct, and furthermore, calls to a return continuation remove unnecessary bindings from the stack – remember that return continuations receive a 1st-class argument.

Values

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

Evaluation Contexts

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

Stores

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

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

$$\begin{array}{rcl}
\langle t_1\ t_2;\ \rho;\ H;\ S;\ E;\ n\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;\ n\rangle_{evals} & \Rightarrow & \langle t_1;\ \rho;\ H;\ S;\ \mathsf{If}(t_2, t_3, n, \rho, E);\ 2\rangle_{evale} \\
\langle \mathsf{let}\ x^m = t_1\ \mathsf{in}\ t_2;\ \rho;\ H;\ S;\ E;\ n\rangle_{evals} & \Rightarrow & \langle t_1;\ \rho;\ H;\ S;\ \mathsf{Let}(x^m, t_2, n, \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 heap/stack 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 stack 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, n, \rho, E);\ n_1\rangle_{evale} \\
\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} \\
\langle \mathsf{Fun}(a, E);\ v;\ H;\ S;\ 1\rangle_{cont} & \Rightarrow & \langle t;\ \rho', x^1 : v;\ H;\ S^{[\le i]};\ E;\ 1\rangle_{evals} \quad \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;\ 1\rangle_{evals} \quad \text{if } a : \left\langle \rho', i, \lambda x^2.t \right\rangle \in H/S \\
\langle \mathsf{Pair1}(t_2, n_2, n, \rho, E);\ v_1;\ H;\ S;\ n_1\rangle_{cont} & \Rightarrow & \langle t_2;\ \rho;\ H;\ S;\ \mathsf{Pair2}(v_1, n_1, n, \rho, E);\ n_2\rangle_{evale} \\
\langle \mathsf{Pair2}(v_1, n_1, n, \rho, E);\ v_2;\ H;\ S;\ n_2\rangle_{cont} & \Rightarrow & \langle E;\ (v_1\ {}_{n_1},{}_{n_2}\ v_2);\ H;\ S;\ n\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, n, \rho, E);\ \mathsf{true};\ H;\ S;\ 2\rangle_{cont} & \Rightarrow & \langle t_2;\ \rho;\ H;\ S;\ E;\ n\rangle_{evals} \\
\langle \mathsf{If}(t_2, t_3, n, \rho, E);\ \mathsf{false};\ H;\ S;\ 2\rangle_{cont} & \Rightarrow & \langle t_3;\ \rho;\ H;\ S;\ E;\ n\rangle_{evals} \\
\langle \mathsf{Let}(x^m, t_2, n, \rho, E);\ v;\ H;\ S;\ m\rangle_{cont} & \Rightarrow & \langle t_2;\ \rho, x^m : v;\ H;\ S;\ E;\ n\rangle_{evals}
\end{array}$$

Figure 7. Low-level Operational Semantics of CPS Programs

To formally analyze the stack behavior of CPS programs, we define a low-level operational semantics of the post-CPS language. The semantics has two features. First, it uses different stores for different kinds of values. Second, it only accounts for programs where every application is a tail-call.

In Figure 7, we present the semantics in the style of an abstract machine. A configuration consists of six elements: a term t, three stores ρ, H, S, an evaluation context E, and a number n representing the class of t. Among the three kinds of stores, ρ is an environment that maps variables to constants and addresses, H is a heap that maps addresses to 1st-class closures, and S is a stack that maps addresses to 2nd-class closures. We assume that heaps and stacks are both ordered, that is, larger addresses appear later in the

$$\langle \mathscr{C}\ t;\ \rho;\ H;\ S;\ E;\ n\rangle_{eval} \Rightarrow \langle t;\ \rho;\ H;\ S;\ \mathsf{Ctrl}(E,n);\ 2\rangle_{eval}$$

$$\langle \mathsf{Fun}(a,E);\ v;\ H;\ S;\ 2\rangle_{cont} \Rightarrow \langle t;\ \rho', x^2 : v;\ H;\ S;\ E;\ 1\rangle_{eval} \qquad \text{if } a : \left\langle \rho', i, \lambda x^2.t \right\rangle \in H/S$$

$$\langle \mathsf{Ctrl}(E,n);\ a;\ H;\ S;\ 2\rangle_{cont} \Rightarrow \langle t;\ (\rho', k^2 : a_S + 1);\ H;\ (S, a_S + 1 : \langle \rho, a_S, \lambda x^n.E[x^n]\rangle);\ \square;\ 2\rangle_{eval} \qquad \text{if } a : \left\langle \rho', i, \lambda k^2.t \right\rangle \in S$$

Figure 8. Low-level Operational Semantics of Direct-Style $\lambda_\perp^{1/2}$ (excerpt)

binding sequence. A closure $\langle \rho, i, \lambda x^m.t\rangle$ carries a whole environment ρ and a stack pointer i, where the latter is used to adjust the stack when the function is called.

Trasition rules are typically defined by an eval-function, which tells us what to do when given a program of a certain shape, and a cont-function, which tells us how to proceed when given a value in a certain evaluation context. In our semantics, we split the eval-function into two: *evals* for application, `if`, and `let` in tail context, and *evale* for other expresions. This allows us to exclude non-tail calls: for instance, it is impossible to pass an application to a function, because arguments are evaluated by the *evale* function.

From the viewpoint of stack space, there are four rules that are particularly important. Two of them are the *evale*-rules of abstraction: we can see that the generated closure keeps the largest stack address a in the current stack S, helping us remember which 2nd-class references are available at the function definition time. The other key rules are the *cont*-rules dealing with a Fun-context, which let us evaluate a function's body with an appropriate stack. If the argument is 1st-class, we use a shrunken stack that only contains bindings within the function's defining scope. If the argument is 2nd-class, we use the whole stack to allow arbitrary 2nd-class references from the argument.

Using this semantics, as well as the definition of the CPS translation, we can easily check that CPS programs do not use too much stack space compared to direct-style ones. 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:

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

The translation has generated five continuation variables, which means five continuations 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 an empty one, because the stack was empty when k was defined. Thus, by exploiting the class information of arguments, we can guarantee the desired stack behavior of programs.

The CPS semantics can also be turned into a semantics of $\lambda_\perp^{1/2}$ in a straightforward manner. This is done in three steps. First, we merge *evals* and *evale* functions into one single function *eval*, and thus allow any nesting function application. Second, in the β-reduction rule with a 2nd-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 $\mathscr{C}$ constructs. These changes are shown in Figure 8. Note that we can expect the shrinking event of the stack in this language, too, although it is not observable from the transition rules.

5 Extensions

While the core language we presented, $\lambda_\perp^{1/2}$, captures most essential features of a functional compiler IR, a realistic implementation may require additional features. We describe relevant extensions of the formal model next.

Recursion We can support recursive functions in $\lambda_\perp^{1/2}$ in a straightforward way by extending the semantics of `let` expressions to `letrec`. The only complication is in the low-level semantics: we must store a closure with an appropriately updated environment and stack pointer that includes.

Polymorphism Extending the language with parametric polymorphism in the style of System F is also straightforward. The key is to only allow abstraction over value types T but not $\perp$. Preventing instantiation of type variables with $\perp$ is necessary to apply the type-directed selective CPS conversion and sufficient to guarantee its type preservation property.

Subtyping Another interesting extension to consider is subtyping, especially its interaction with class conversion. We can always regard a 1st-class term as a 2nd-class one [22] and hence incorporate this conversion into a standard subtyping relation. It is also possible to allow conversion from $t : \perp$ to $t : T$, by dropping the continuation via $\mathscr{C}\ (\lambda k^2.t)$. However, this conversion needs to remain explicit. For the same reason as with parametric polymorphism, we need to keep $\perp$ distinct from value types T.

First-Class and Delimited Continuations While this paper follows Kennedy [16], Maurer et al. [19] and others in assuming a language that does not support first-class continuations at the user level, it is not hard to extend the internal language in this direction. The main change is to adapt the typing rules for $\mathscr{C}$ to support a version that introduces

1st-class continuations. We can also go another step and add control operators shift and reset for 1st-class *delimited* continuations [6]. In this case, function types need to be extended to include the type of the delimited context, and the selective CPS transform needs to be extended to deal with context types as described by Rompf et al. [28].

6 Case Study

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 loops with break and continue, return statements, or exceptions. Especially the combination with explicit 2nd-class functions enables some useful programming idioms. Consider the following foreach function to traverse a list:

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

Note that the function argument f is a 2nd-class value, here denoted by the @local annotation. We can build other useful functions on top of foreach, e.g., find:

```
def find[A](xs: List[A])(@local p: A => Boolean) = {
  xs foreach (x => if (p) 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 to not 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.

Recursive Join Points vs. Non-Local Returns Let us compare this implementation with an example from Maurer et al. [19]. A key observation of Maurer et al. is that join points can be made *recursive* and thus lead to new optimization opportunities. For example, in the program below, find is implemented using an auxiliary rec function that can be contified as a recursive join point:

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

This allows to compile rec into a jump, but more exciting optimizations are possible when find is used as follows, 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
} }
```

Here, find can be inlined into any, eliminating the match expression within any can be cancelled by exploiting the join-point nature of rec:

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

In our model, we can do the same, but we are also considerably more flexible. We consider the same implementation of find based on foreach as shown above and use it to implement any:

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

Next, we 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
  }
}
```

And now, we contract without reasoning about loops, contification, or anything else, just 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))
}
```

Since we do not have to reason about exposed recursion or loops, our approach works for any iterable structure (trees, hash maps, etc.), 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 [29] and the MiniScala Scala to native compiler used for teaching compiler construction at our institution. These two implementations illustrate the flexibility of the

```
// 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
      };
}
```

Figure 9. C code generated by LMS regexp matcher for regexp 'AAB'

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) [29] is a DSL compiler framework, embedded as library in Scala. LMS features a graph-based IR. All intermediate results are named but execution order is flexible up to control dependencies expressed in the graph [26]. User-defined rewrites are pervasive and as important or even more so for end-to-end performance of user programs than in GHC. LMS emits source code, typically Scala or C, and hence a global CPS transform would be impractical.

We have extended LMS with a local control operator and support for 2nd-class continuations that map to labels and goto statements in C. We have modified a pre-existing regexp compiler [27, 30] to use this new facility. The original version would emit top-level C functions for each DFA state along with a driver loop (Figure 9, left). The new version emits local labels and gotos instead (Figure 9, right), which leads to an overall 23% speedup on a mix of regex benchmarks.

7.2 MiniScala

MiniScala is a compiler from a restricted Scala subset 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 but remain 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 on the surface level.

In this experiment, we use these 2nd-class functions to replace heap allocations with stack allocations. We ported a series of benchmarks from the Computer Languages Benchmarks Game[1] and Scala Native[2]. The former set of benchmarks is compelling because it is used frequently in the literature [1, 11, 13, 21, 24]. 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 or Rust, which compile to native code.

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 both heap and stack memory used by the program on a medium-sized input (i.e., long enough that it takes a few seconds for a program to complete). The results are shown in Table 10. The two benchmarks where the improvement in memory usage is asymptotically significant are `pidigits` and `list`. Another noteworthy result is for the `bounce` benchmark, in which neither the variables that are part of a (mutable) generator state nor the higher-order functions using them could be made second-class because the former are stored in an array, therefore we see only slight improvements there. The `storage` benchmark shows no improvements at all because of the same reason, although this is amplified by the frequency of such stores. Other benchmarks do not show noticeable gains.

8 Instantiation Choices and Discussion

An appealing aspect of our IR is that, as presented, it leaves many choices open.

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 useful to refer to things. For example, many dataflow analyses rely on identifiers for expressions. In a graph-based IR, common subexpression eliminatino (CSE) and global value numbering (GVN) are basically just hash-consing. On the hand, retaining nested expressions can be useful for simplicity. Instruction selection and register assignment can work very well in a setting where expression nesting defines lifetime.

[1]https://benchmarksgame.alioth.debian.org/
[2]https://github.com/scala-native/scala-native

	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

Figure 10. 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 (#).

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 ontinuation arguments correspond to ϕ functions in SSA. Assembly code generation needs to emit named labels anyways. On the other hand, the ability to use continuations anonymously can also be beneficial. 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 can be used with or without CPS conversion, 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 optimization (as in MiniScala in Section 7.2), or both. 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

Rabbit [31] and Orbit [17] were breakthroughs in compiler design that paved the way for CPS based intermediate languages. Even these early systems recognized the need to have some form of internal distinction between lambdas representing functions and those representing continuations. The Orbit retrospective [18] contains an illuminating historical perspective.

The question of whether a compiler IR should be CPS or not has been lively discussed for decades. Appel [2] argued for CPS by showing the advantages of fixing the evaluation order, as well as the sound reduction behavior of CPS programs. Although naïve CPS would generate extra redexes and require expensive allocation, subsequent studies have solved these problems by means of administrative reduction and a special binder for continuations [16].

On the other hand, Flanagan et al. [10] showed that the effect of naïve CPS and accompagning simplification / optimization can be achieved by a direct-style translation called the ANF translation. ANF makes control flow explicit by `let`-binding every intermediate result, but keeps evaluation order implicit, which is preferable in call-by-need languages such as Haskell. One downside of direct-style IRs is the code duplication issue that arises in case-like constructs, but it has proven avoidable by adding join points in the style of Maurer et al. [19].

The idea of making certain continuations explicit via $\mathscr{C}$ is similar to Reppy's local CPS conversion (LCPS) [25]. LCPS is local in the sense that it only converts non-tail calls into CPS. Reppy implemented the translation in the Moby compiler, and successfully improved performance of nested loops.

Osvald et al. [22] 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, implementes as a compiler plug-in for Scala, does not provide 2nd-class values at the level of a compiler IR.

Further related work is concerned with realistic compilation by program rransformation [15], and using sequent calculus as the basis for an intermediate language [8]. The importance of shrinking reductions and suitable implementation techniques especially in CPS have also been a subject of discussion in the great CPS debate [4, 5].

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 claimed that the right question to ask is "before or after CPS", i.e., we should allow both direct-style and CPS optimizations and leave any further decision to the compiler designer, with the flexibility of changing such decisions later. Based on this idea, we designed an IR that has a carefully chosen control operator, and a facility for expressing non-returning / non-escaping functions. We then gave a selective CPS translation of our IR, which generates efficient programs with a desired stack behavior. To show the efficiency and flexibility of our approach, we also implemented our IR and CPS translation in different compilers, and observed significant performance and memory usage improvements in two benchmarks.

References

[1] 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

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

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

[4] Nick Benton, Andrew Kennedy, Sam Lindley, and Claudio V. Russo. 2004. Shrinking Reductions in SML.NET. In *IFL (Lecture Notes in Computer Science)*, Vol. 3474. Springer, 142–159.

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

[6] Olivier Danvy and Andrzej Filinski. 1990. Abstracting control. In *Proc. LFP'90*. 151–160.

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

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

[9] 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.

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

[11] 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

[12] Robert Harper, Bruce F. Duba, and David B. MacQueen. 1993. Typing First-Class Continuations in ML. *J. Funct. Program.* 3, 4 (1993), 465–484.

[13] 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

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

[15] Richard Kelsey and Paul Hudak. 1989. Realistic Compilation by Program Transformation. In *POPL*. 281–292.

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

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

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

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

[20] Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (1991), 55–92.

[21] 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.

[22] 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.

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

[24] 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

[25] 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.

[26] 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

[27] 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.

[28] Tiark Rompf, Ingo Maier, and Martin Odersky. 2009. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In *ICFP*. 317–328.

[29] 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

[30] 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)*.

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