One Step at a Time

A Functional Derivation of Small-Step Evaluators from Big-Step Counterparts

Ferdinand Vesely[1,2(✉)] and Kathleen Fisher[1]

[1] Tufts University, Medford, USA
{fvesely,kfisher}@eecs.tufts.edu
[2] Swansea University, Swansea, UK
f.vesely@swansea.ac.uk

Abstract. Big-step and small-step are two popular flavors of operational semantics. Big-step is often seen as a more natural transcription of informal descriptions, as well as being more convenient for some applications such as interpreter generation or optimization verification. Small-step allows reasoning about non-terminating computations, concurrency and interactions. It is also generally preferred for reasoning about type systems. Instead of having to manually specify equivalent semantics in both styles for different applications, it would be useful to choose one and derive the other in a systematic or, preferably, automatic way.

Transformations of small-step semantics into big-step have been investigated in various forms by Danvy and others. However, it appears that a corresponding transformation from big-step to small-step semantics has not had the same attention. We present a fully automated transformation that maps big-step evaluators written in direct style to their small-step counterparts. Many of the steps in the transformation, which include CPS-conversion, defunctionalisation, and various continuation manipulations, mirror those used by Danvy and his co-authors. For many standard languages, including those with either call-by-value or call-by-need and those with state, the transformation produces small-step semantics that are close in style to handwritten ones. We evaluate the applicability and correctness of the approach on 20 languages with a range of features.

Keywords: Structural operational semantics · Big-step semantics · Small-step semantics · Interpreters · Transformation · Continuation-passing style · Functional programming

1 Introduction

Operational semantics allow language designers to precisely and concisely specify the meaning of programs. Such semantics support formal type soundness proofs [29], give rise (sometimes automatically) to simple interpreters [15,27] and debuggers [14], and document the correct behavior for compilers. There are

L. Caires (Ed.): ESOP 2019, LNCS 11423, pp. 205–231, 2019.
https://doi.org/10.1007/978-3-030-17184-1_8

two popular approaches for defining operational semantics: big-step and small-step. *Big-step semantics* (also referred to as *natural* or *evaluation* semantics) relate initial program configurations directly to final results in one "big" evaluation step. In contrast, *small-step semantics* relate intermediate configurations consisting of the term currently being evaluated and auxiliary information. The initial configuration corresponds to the entire program, and the final result, if there is one, can be obtained by taking the transitive-reflexive closure of the small-step relation. Thus, computation progresses as a series of "small steps."

The two styles have different strengths and weaknesses, making them suitable for different purposes. For example, big-step semantics naturally correspond to definitional interpreters [23], meaning many big-step semantics can essentially be transliterated into a reasonably efficient interpreter in a functional language. Big-step semantics are also more convenient for verifying program optimizations and compilation – using big-step, semantic preservation can be verified (for terminating programs) by induction on the derivation [20,22].

In contrast, small-step semantics are often better suited for stepping through the evaluation of an example program, and for devising a type system and proving its soundness via the classic syntactic method using progress and preservation proofs [29]. As a result, researchers sometimes develop multiple semantic specifications and then argue for their equivalence [3,20,21]. In an ideal situation, the specifier writes down a single specification and then derives the others.

Approaches to deriving big-step semantics from a small-step variant have been investigated on multiple occasions, starting from semantics specified as either interpreters or rules [4,7,10,12,13]. An obvious question is: what about the reverse direction?

This paper presents a systematic, mechanised transformation from a big-step interpreter into its small-step counterpart. The overall transformation consists of multiple stages performed on an interpreter written in a functional programming language. For the most part, the individual transformations are well known. The key steps in this transformation are to explicitly represent control flow as *continuations*, to defunctionalise these continuations to obtain a datatype of reified continuations, to "tear off" recursive calls to the interpreter, and then to return the reified continuations, which represent the rest of the computation. This process effectively produces a stepping function. The remaining work consists of finding translations from the reified continuations to equivalent terms in the source language. If such a term cannot be found, we introduce a new term constructor. These new constructors correspond to the intermediate auxiliary forms commonly found in handwritten small-step definitions.

We define the transformations on our *evaluator definition language* – an extension of λ-calculus with call-by-value semantics. The language is untyped and, crucially, includes tagged values (variants) and a case analysis construct for building and analysing object language terms. Our algorithm takes as input a big-step interpreter written in this language in the usual style: a main function performing case analysis on a top-level term constructor and recursively calling itself or auxiliary functions. As output, we return the resulting small-step

interpreter which we can "pretty-print" as a set of small-step rules in the usual style. Hence our algorithm provides a fully automated path from a restricted class of big-step semantic specifications written as interpreters to corresponding small-step versions.

To evaluate our algorithm, we have applied it to 20 different languages with various features, including languages based on call-by-name and call-by-value λ-calculi, as well as a core imperative language. We extend these base languages with conditionals, loops, and exceptions.

We make the following contributions:

- We present a multi-stage, automated transformation that maps any deterministic big-step evaluator into a small-step counterpart. Section 2 gives an overview of this process. Each stage in the transformation is performed on our *evaluator definition language* – an extended call-by-value λ-calculus. Each stage in the transformation is familiar and principled. Section 4 gives a detailed description.
- We have implemented the transformation process in Haskell and evaluate it on a suite of 20 representative languages in Section 5. We argue that the resulting small-step evaluation rules closely mirror what one would expect from a manually written small-step specification.
- We observe that the same process with minimal modifications can be used to transform a big-step semantics into its *pretty-big-step* [6] counterpart.

2 Overview

In this section, we provide an overview of the transformation steps on a simple example language. The diagram in Fig. 1 shows the transformation pipeline. As the initial step, we first convert the input big-step evaluator into continuation-passing style (CPS). We limit the conversion to the *eval* function itself and leave all other functions in direct style. The resulting continuations take a value as input and advance the computation. In the generalization step, we modify these continuations so that they take an arbitrary term and evaluate it to a value before continuing as before. With this modification, each continuation handles both the general non-value case and the value case itself. The next stage lifts a carefully chosen set of free variables as arguments to continuations, which allows us to define all of them at the same scope level. After generalization and argument lifting, we can invoke continuations directly to switch control, instead of passing them as arguments to the *eval* function. Next we defunctionalize the continuations, converting them into a set of tagged values together with an *apply* function capturing their meaning. This transformation enables the next step, in which we remove recursive tail-calls to *apply*. This allows us to interrupt the interpreter and make it return a continuation or a term: effectively, it yields a stepping function, which is the essence of a small-step semantics. The remainder of the pipeline converts continuations to terms, performs simplifications, and then converts the CPS evaluator back to direct style to obtain the final small-step interpreter. This interpreter can be pretty-printed as a set of small-step rules.

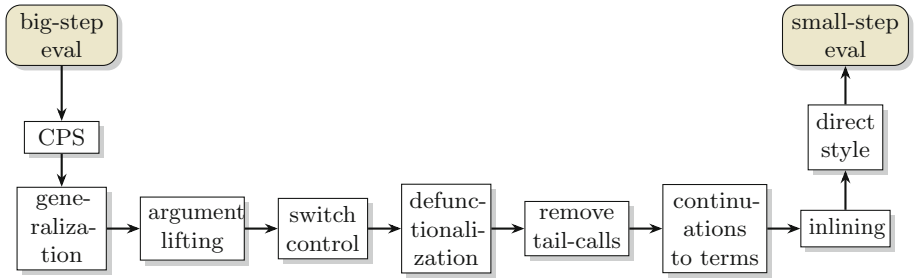

Fig. 1. Transformation overview

Our example language is a λ-calculus with call-by-value semantics. Fig. 2 gives its syntax and big-step rules. We use environments to give meaning to variables. The only values in this language are closures, formed by packaging a λ-abstraction with an environment.

$$v ::= \mathbf{clo}(x, e, \rho)$$
$$e ::= \mathbf{var}(x) \mid \mathbf{val}(v) \mid \mathbf{lam}(x, e) \mid \mathbf{app}(e_1, e_2)$$

$$x \in \mathit{Var} \qquad \rho \in \mathit{Env} = \mathit{Var} \to \mathit{Val}$$

$$\frac{}{\rho \vdash \mathbf{val}(v) \Downarrow v} \qquad \frac{\rho(x) = v}{\rho \vdash \mathbf{var}(x) \Downarrow v}$$

$$\frac{}{\rho \vdash \mathbf{lam}(x, e) \Downarrow \mathbf{clo}(x, e, \rho)}$$

$$\frac{\rho \vdash e_1 \Downarrow \mathbf{clo}(x, e, \rho') \qquad \rho \vdash e_2 \Downarrow v_2 \qquad \rho'[x \mapsto v_2] \vdash e \Downarrow v}{\rho \vdash \mathbf{app}(e_1, e_2) \Downarrow v}$$

Fig. 2. Example: Call-by-value λ-calculus, abstract syntax and big-step semantics

We will now give a series of interpreters to illustrate the transformation process. We formally define the syntax of the meta-language in which we write these interpreters in Section 3, but we believe for readers familiar with functional programming the language is intuitive enough to not require a full explanation at this point. Shaded text highlights (often small) changes to subsequent interpreters.

Big-Step Evaluator. We start with an interpreter corresponding directly to the big-step semantics given in Fig. 2. We represent environments as functions – the empty environment returns an error for any variable. The body of the *eval* function consists of a pattern match on the top-level language term. Function abstractions are evaluated to closures by packaging them with the current environment. The only term that requires recursive calls to *eval* is application: both its arguments are evaluated in the current environment, and then its first argument is pattern-matched against a closure, the body of which is then evaluated to a value in an extended environment using a third recursive call to *eval*.

```
let empty = λx. error() in
let update x v ρ = λx'. let xx' = (== x x') in if xx' then v else (ρ x') in
let rec eval e ρ =
  case e of {
    val(v) → v |
    var(x) → let v = (ρ x) in v |
    lam(x, e') → clo(x, e', ρ) |
    app(e1, e2) →
        let v1 = (eval e1 ρ) in
        let v2 = (eval e2 ρ) in
        case v1 of {
          clo(x, e', ρ') →
              let ρ'' = (update x v2 ρ') in
              let v = (eval e' ρ'') in
              v
        }
  }
```

CPS Conversion. Our first transformation introduces a continuation argument to *eval*, capturing the "rest of the computation" [9,26,28]. Instead of returning the resulting value directly, *eval* will pass it to the continuation. For our example we need to introduce three continuations – all of them in the case for **app**. The continuation $kapp_1$ captures what remains to be done after evaluating the first argument of **app**, $kapp_2$ captures the computation remaining after evaluating the second argument, and $kclo_1$ the computation remaining after the closure body is fully evaluated. This final continuation simply applies the top-level continuation to the resulting value and might seem redundant; however, its utility will become apparent in the following step. Note that the CPS conversion is limited to the *eval* function, leaving any other functions in the program intact.

```
let rec eval e ρ k =
  case e of {
    val(v) → (k v) |
    var(x) → let v = (ρ x) in (k v) |
    lam(x, e') → (k clo(x, e', ρ)) |
    app(e1, e2) →
      letcont kapp1 v1 =
          letcont kapp2 v2 =
            case v1 of {
              clo(x, e', ρ') →
                  let ρ'' = (update x v2 ρ') in
                  letcont kclo1 v = (k v) in
                  (eval e' ρ'' (λv. (kclo1 v)))
            } in
            (eval e2 ρ (λv2. (kapp2 v2))) in
        (eval e1 ρ (λv1. (kapp1 v1)))
  }
```

Generalization. Next, we modify the continuation definitions so that they handle both the case when the term is a value (the original case) and the case where it is still a term that needs to be evaluated. To achieve this goal, we introduce a case analysis on the input. If the continuation's argument is a value, the evaluation will proceed as before. Otherwise it will call *eval* with itself as the continuation argument. Intuitively, the latter case will correspond to a congruence rule in the resulting small-step semantics and we refer to these as *congruence cases* in the rest of this paper.

```
let rec eval e ρ k = case e of {
    val(v) → (k val(v)) |
    var(x) → let v = (ρ x) in (k val(v)) |
    lam(x, e′) → (k val(clo(x, e′, ρ))) |
    app(e1, e2) →
        letcont kapp1 e1 =
          case e1 of {
            val(v1) →
              ...
                        case v1 of {
                          clo(x, e′, ρ′) →
                              let ρ″ = (update x v2 ρ′) in
                              letcont kclo1 e =
                                case e of {
                                  val(v) → (k val(v)) |
                                  ELSE(e) → (eval e ρ″ (λe′. (kclo1 e′)))
                                } in
                              (eval e′ ρ″ (λv. (kclo1 v)))
              ...
            ELSE(e1) → (eval e1 ρ (λe1′. (kapp1 e1′)))
          } in
        (eval e1 ρ (λv1. (kapp1 v1)))
}
```

Argument Lifting. The free variables inside each continuation can be divided into those that depend on the top-level term and those that parameterize the evaluation. The former category contains variables dependent on subterms of the top-level term, either by standing for a subterm itself, or by being derived from it. In our example, for $kapp_1$, it is the variable e_2, i.e., the right argument of **app**, for $kapp_2$, the variable v_1 as the value resulting from evaluating the left argument, and for $kclo_1$ it is the environment obtained by extending the closure's environment by binding the closure variable to the operand value (ρ'' derived from v_2). We lift variables that fall into the first category, that is, variables derived from the input term. We leave variables that parametrize the evaluation, such as the input environment or the store, unlifted. The rationale is that, eventually, we want the continuations to act as term constructors and they need to carry information not contained in arguments passed to *eval*.

```
let rec eval e ρ k = case e of {
    ...
    app(e1, e2) →
      letcont kapp1 e2 e1 =
                ...
                letcont kapp2 v1 e2 =
                  ...
                              letcont kclo1 ρ′ e =
                                case e of {
                                  val(v) → (k val(v)) |
                                  ELSE(e) → (eval e ρ′ (λe′. (kclo1 ρ′ e′)))
                                } in
                              (eval e′ ρ″ (λv. (kclo1 ρ″ v)))
                          } |
                  ELSE(e2) → (eval e2 ρ (λe2′. (kapp2 v1 e2′)))
                } in
              (eval e2 ρ (λv2. (kapp2 v1 v2))) |
        ELSE(e1) → (eval e1 ρ (λe1′. (kapp1 e2 e1′)))
      } in
      (eval e1 ρ (λv1. (kapp1 e2 v1)))
}
```

Continuations Switch Control. Since continuations now handle the full evaluation of their argument themselves, they can be used to switch stages in the evaluation of a term. Observe how in the resulting evaluator below, the evaluation of an **app** term progresses through stages initiated by $kapp_1$, $kapp_2$, and finally $kclo_1$.

```
let rec eval e ρ k = case e of {
    ...
    app(e1, e2) →
      letcont kapp1 e2 e1 =
        ...
              letcont kapp2 v1 e2 =
                ...
                letcont kclo1 ρ′ e =
                  ...
                in (kclo1 ρ″ e′)
                ...
              in (kapp2 v1 e2) |
        ...
      in (kapp1 e2 e1)
}
```

Defunctionalization. In the next step, we defunctionalize continuations. For each continuation, we introduce a constructor with the corresponding number of arguments. The *apply* function gives the meaning of each defunctionalized continuation.

```
let rec apply eval e_k ρ k = case e_k of {
    kapp1(e_2, e_1) →
        case e_1 of {
          val(v_1) → (apply eval kapp2(v_1, e_2) ρ k) |
          ELSE(e_1) → (eval e_1 ρ (λe'_1. (apply eval kapp1(e_2, e'_1) ρ k)))
        } |
    kapp2(v_1, e_2) →
        case e_2 of {
          val(v_2) →
              case v_1 of {
                clo(x, e', ρ') →
                    let ρ'' = (update x v_2 ρ')
                    in (apply eval kclo1(ρ'', e') ρ k)
              } |
          ELSE(e_2) → (eval e_2 ρ (λe'_2. (apply eval kapp2(v_1, e'_2) ρ k)))
        } |
    kclo1(ρ', e) →
        case e of {
          val(v) → (k val(v)) |
          ELSE(e) → (eval e ρ' (λe'. (apply eval kclo1(ρ', e') ρ k)))
        }
  } in
let rec eval e ρ k = case e of {
    val(v) → (k val(v)) |
    var(x) → let v = (ρ x) in (k val(v)) |
    lam(x, e') → (k val(clo(x, e', ρ))) |
    app(e_1, e_2) → (apply eval kapp1(e_2, e_1) ρ k)
  }
```

Remove Tail-Calls. We can now move from a recursive evaluator to a stepping function by modifying the continuation arguments passed to *eval* in congruence cases. Instead of calling *apply* on the defunctionalized continuation, we return the defunctionalized continuation itself. Note, that we leave intact those calls to *apply* that switch control between different continuations (e.g., in the definition of *eval*).

```
let rec apply eval e_k ρ k = case e_k of {
    kapp1(e_2, e_1) →
        case e_1 of {
          val(v_1) → (apply eval kapp2(v_1, e_2) ρ k) |
          ELSE(e_1) → (eval e_1 ρ (λe'_1. (k kapp1(e_2, e'_1))))
        } |
    kapp2(v_1, e_2) →
        case e_2 of {
          val(v_2) → ... (apply eval kclo1(ρ'', e') ρ k) |
          ELSE(e_2) → (eval e_2 ρ (λe'_2. (k kapp2(v_1, e'_2))))
        } |
    kclo1(ρ', e) →
```

```
      case e of {
        val(v) → (k val(v)) |
        ELSE(e) → (eval e ρ′ (λe′. (k kclo1(ρ′, e′))))
      }
} in ...
```

Convert Continuations into Terms. At this point, we have a stepping function that returns either a term or a continuation, but we want a function returning only terms. The most straightforward approach to achieving this goal would be to introduce a term constructor for each defunctionalized continuation constructor. However, many of these continuation constructors can be trivially expressed using constructors already present in the object language. We want to avoid introducing redundant terms, so we aim to reuse existing constructors as much as possible. In our example we observe that **kapp1**(e_2, e_1) corresponds to **app**(e_1, e_2), while **kapp2**(v_1, e_2) to **app**(**val**$(v_1), e_2)$. We might also observe that **kclo1**(ρ', e) would correspond to **app**(**clo**$(x, e, \rho), val(v_2))$ if $\rho' =$ update $x\ v_2\ \rho$. Our current implementation doesn't handle such cases, however, and so we introduce **kclo1** as a new term constructor.

```
let rec apply eval e_k ρ k = case e_k of {
  kapp1(e_2, e_1) →
      case e_1 of {
        val(v_1) → (apply eval kapp2(v_1, e_2) ρ k) |
        ELSE(e_1) → (eval e_1 ρ (λe′_1. (k app(e′_1, e_2))))
      } |
  kapp2(v_1, e_2) →
      case e_2 of {
        val(v_2) →
            case v_1 of {
              clo(x, e′, ρ′) → let ρ″ = (update x v_2 ρ′) in kclo1(ρ″, e′)
            } |
        ELSE(e_2) → (eval e_2 ρ (λe′_2. (k app(val(v_1), e′_2))))
      } |
  kclo1(ρ′, e) →
      case e of {
        val(v) → (k val(v)) |
        ELSE(e) → (eval e ρ′ (λe′. (k kclo1(ρ′, e′))))
      }
} in
let rec eval e ρ k = case e of {
  ...
  kclo1(ρ′, e′) → (apply eval kclo1(ρ′, e′) ρ k)
}
```

Inlining and Simplification. Next, we eliminate the *apply* function by inlining its applications and simplifying the result. At this point we have obtained a small-step interpreter in continuation-passing style.

```
let rec eval e ρ k = case e of {
  ...
  app(e1, e2) →
      case e1 of {
        val(v1) →
            case e2 of {
              val(v2) →
                  case v1 of {
                    clo(x, e′, ρ′) → let ρ″ = (update x v2 ρ′) in kclo1(ρ″, e′)
                  } |
              ELSE(e2) → (eval e2 ρ (λe2′. (k app(val(v1), e2′))))
            } |
        ELSE(e1) → (eval e1 ρ (λe1′. (k app(e1′, e2))))
      } |
  kclo1(ρ′, e′) →
      case e′ of {
        val(v) → (k val(v)) |
        ELSE(e) → (eval e ρ′ (λe′. (k kclo1(ρ′, e′))))
      }
}
```

Convert to Direct Style and Remove the Value Case. The final transformation is to convert our small-step interpreter back to direct style. Moreover, we also remove the value case $\mathsf{val}(v) \to \mathsf{val}(v)$ as we, usually, do not want values to step.

```
let rec eval e ρ = case e of {
  var(x) → let v = (ρ x) in val(v) |
  lam(x, e′) → val(clo(x, e′, ρ)) |
  app(e1, e2) →
      case e1 of {
        val(v1) →
            case e2 of {
              val(v2) →
                  case v1 of {
                    clo(x, e′, ρ′) → let ρ″ = (update x v2 ρ′) in kclo1(ρ″, e′)
                  } |
              ELSE(e2) → let e2′ = (eval e2 ρ) in app(val(v1), e2′)
            } |
        ELSE(e1) → let e1′ = (eval e1 ρ) in app(e1′, e2)
      } |
  kclo1(ρ′, e′) →
      case e′ of {
        val(v) → val(v) |
        ELSE(e) → let e′ = (eval e ρ′) in kclo1(ρ′, e′)
      }
}
```

Small-Step Evaluator. Fig. 3 shows the small-step rules corresponding to our last interpreter. Barring the introduction of the **kclo1** constructor, the resulting semantics is essentially identical to one we would write manually.

$$1\ \frac{v = \rho\ x}{\rho \vdash \mathbf{var}(x) \to \mathbf{val}(v)} \qquad 2\ \frac{}{\rho \vdash \mathbf{lam}(x, e') \to \mathbf{val}(\mathbf{clo}(x, e', \rho))}$$

$$3\ \frac{\rho'' = \mathsf{update}\ x\ v_2\ \rho'}{\rho \vdash \mathbf{app}(\mathbf{val}(\mathbf{clo}(x, e', \rho')), \mathbf{val}(v_2)) \to \mathbf{kclo1}(\rho'', e')}$$

$$4\ \frac{\rho \vdash e_2 \to e_2'}{\rho \vdash \mathbf{app}(\mathbf{val}(v_1), e_2) \to \mathbf{app}(\mathbf{val}(v_1), e_2')} \qquad 5\ \frac{\rho \vdash e_1 \to e_1'}{\rho \vdash \mathbf{app}(e_1, e_2) \to \mathbf{app}(e_1', e_2)}$$

$$6\ \frac{}{\rho \vdash \mathbf{kclo1}(\rho', \mathbf{val}(v)) \to \mathbf{val}(v)} \qquad 7\ \frac{\rho' \vdash e \to e'}{\rho \vdash \mathbf{kclo1}(\rho', e) \to \mathbf{kclo1}(\rho', e')}$$

Fig. 3. Resulting small-step semantics

3 Big-Step Specifications

We define our transformations on an untyped extended λ-calculus with call-by-value semantics that allows the straightforward definition of big- and small-step interpreters. We call this language an *evaluator definition language* (EDL).

3.1 Evaluator Definition Language

Table 1 gives the syntax of EDL. We choose to restrict ourselves to A-normal form, which greatly simplifies our partial CPS conversion without compromising readability. Our language has the usual call-by-value semantics, with arguments being evaluated left-to-right. All of the examples of the previous section were written in this language.

Our language has 3 forms of let-binding constructs: the usual (optionally recursive) **let**, a let-construct for evaluator definition, and a let-construct for defining continuations. The behavior of all three constructs is the same, however, we treat them differently during the transformations. The **leteval** construct also comes with the additional static restriction that it may appear only once (i.e., there can be only one evaluator). The **leteval** and **letcont** forms are recursive by default, while **let** has an optional **rec** specifier to create a recursive binding. For simplicity, our language does not offer implicit mutual recursion, so mutual recursion has to be made explicit by inserting additional arguments. We do this when we generate the *apply* function during defunctionalization.

Notation and Presentation. We use vector notation to denote syntactic lists belonging to a particular sort. For example, $\vec{e}$ and $\vec{ae}$ are lists of elements of, respectively, *Expr* and *AExpr*, while $\vec{x}$ is a list of variables. Separators can be spaces (e.g., function arguments) or commas (e.g., constructor arguments or configuration components). We expect the actual separator to be clear from the context. Similarly for lists of expressions: $\vec{e}$, $\vec{ae}$, etc. In let bindings, $f\ x_1\ \ldots\ x_n = e$ and $f = \lambda x_1\ \ldots\ x_n.\ e$ are both syntactic sugar for $f = \lambda x_1.\ \ldots\ \lambda x_n.\ e$.

Table 1. Syntax of the evaluator definition language.

Expr ∋ e ::=	**let** $bn = ce$ **in** e	(let-binding)
	| **let rec** $bn = ce$ **in** e	(recursive let-binding)
	| **leteval** $x = ce$ **in** e	(evaluator definition)
	| **letcont** $k = ce$ **in** e	(continuation definition)
	| ce	
CExpr ∋ ce ::=	$(ae\ ae\ \ldots)$	(application)
	| **case** ae **of** { cas | … | cas }	(pattern matching)
	| **if** ae **then** e **else** e	(conditional)
	| ae	
AExpr ∋ ae ::=	v | op	(value, operator)
	| x | k	(variable, continuation variable)
	| $\boldsymbol{\lambda} bn.\ e$	(λ-abstraction)
	| $c(ae, \ldots, ae)$	(constructor application)
	| $\langle\ ae, \ldots, ae\ \rangle$	(configuration expression)
Binder ∋ bn ::=	x | $\langle\ x, \ldots, x\ \rangle$	(variable, configuration)
Case ∋ cas ::=	$c(x, \ldots, x) \to e$	(constructor pattern)
	| **ELSE**$(x) \to e$	(default pattern)
Value ∋ v ::=	n | b | $c(v,\ldots,v)$ | $\langle\ v,\ldots,v\ \rangle$ | **abs**$(\boldsymbol{\lambda} x.e, \rho)$	

4 Transformation Steps

In this section, we formally define each of the transformation steps informally described in Section 2. For each transformation function, we list only the most relevant cases; the remaining cases trivially recurse on the A-normal form (ANF) abstract syntax. We annotate functions with E, CE, and AE to indicate the corresponding ANF syntactic classes. We omit annotations when a function only operates on a single syntactic class. For readability, we annotate meta-variables to hint at their intended use – ρ stands for read-only entities (such as environments), whereas σ stands for read-write or "state-like" entities of a configuration (e.g., stores or exception states). These can be mixed with our notation for syntactic lists, so, for example, $\vec{x}^{\sigma}$ is a sequence of variables referring to state-like entities, while $\vec{ae}^{\rho}$ is a sequence of a-expressions corresponding to read-only entities.

4.1 CPS Conversion

The first stage of the process is a *partial* CPS conversion [8,25] to make control flow in the evaluator explicit. We limit this transformation to the main evaluator function, i.e., only the function *eval* will take an additional continuation argument and will pass results to it. Because our input language is already in ANF, the conversion is relatively easy to express. In particular, applications of the evaluator are always **let**-bound to a variable (or appear in a tail position),

which makes constructing the current continuation straightforward. Below are the relevant clauses of the conversion. For this transformation we assume the following easily checkable properties:

- The evaluator name is globally unique.
- The evaluator is *never* applied partially.
- All bound variables are distinct.

The conversion is defined as three mutually recursive functions with the following signatures:

$$\begin{aligned}\mathsf{cps}_E &: \mathit{Expr} \to (\mathit{CExpr} \to \mathit{Expr}) \to \mathit{Expr}\\ \mathsf{cps}_{CE} &: \mathit{CExpr} \to (\mathit{CExpr} \to \mathit{Expr}) \to \mathit{Expr}\\ \mathsf{cps}_{AE} &: \mathit{AExpr} \to \mathit{AExpr}\end{aligned}$$

In the equations, $\mathcal{K}$, $\mathcal{I}$, $\mathcal{A}_k : \mathit{CExpr} \to \mathit{Expr}$ are meta-continuations; $\mathcal{I}$ injects a *CExpr* into *Expr*.

$$\begin{aligned}&\mathsf{cps}_E\big[\mathbf{leteval}\ \mathit{eval}\ \vec{bn} = e_1\ \mathbf{in}\ e_2\big]\ \mathcal{K} =\\ &\qquad \mathbf{leteval}\ \mathit{eval}\ \vec{bn}\ k = \big(\mathsf{cps}_E\big[e_1\big]\ \mathcal{A}_k\big)\ \mathbf{in}\ \big(\mathsf{cps}_E\big[e_2\big]\ \mathcal{K}\big)\\ &\qquad \text{where } k \text{ is a fresh continuation variable}\end{aligned}$$

$$\begin{aligned}&\mathsf{cps}_E\big[\mathbf{let}\ bn = (\mathit{eval}\ ae_1\ \vec{ae})\ \mathbf{in}\ e\big]\ \mathcal{K} =\\ &\qquad \mathbf{letcont}\ k\ bn = \big(\mathsf{cps}_E\ \big[e\big]\ \mathcal{K}\big)\ \mathbf{in}\ \mathsf{cps}_{CE}\big[(\mathit{eval}\ ae_1\ \vec{ae})\big]\ \mathcal{A}_k\\ &\qquad \text{where } k \text{ is a fresh continuation variable}\end{aligned}$$

$$\begin{aligned}&\mathsf{cps}_E\ \big[\mathbf{let}\ bn = ce\ \mathbf{in}\ e\ \big]\ \mathcal{K} =\\ &\qquad \mathsf{renorm}\big[\mathbf{let'}\ bn = \big(\mathsf{cps}_{CE}\ \big[ce\big]\ \mathcal{I}\big)\ \mathbf{in}\ \big(\mathsf{cps}_E\ \big[e\big]\ \mathcal{K}\big)\big]\end{aligned}$$

$$\begin{aligned}&\mathsf{cps}_{CE}\ \big[(\mathit{eval}\ ae_1\ \vec{ae})\big]\ \mathcal{K} = \ (\mathit{eval}\ \big(\mathsf{cps}_{AE}\ \big[ae_1\big]\big)\ \big(\mathsf{cps}_{AE}\ \big[\vec{ae}\big]\big)\ (\boldsymbol{\lambda} x.\ \mathcal{K}\big[x\big]))\\ &\qquad \text{where } x \text{ is a fresh variable}\end{aligned}$$

$$\mathsf{cps}_{CE}\ \big[ae\big]\ \mathcal{K} = \mathcal{K}\big(\mathsf{cps}_{AE}\ \big[ae\big]\big)$$

$$\mathsf{cps}_{AE}\ \big[\boldsymbol{\lambda} x.e\big] = \boldsymbol{\lambda} x.\ \big(\mathsf{cps}_E\ \big[e\big]\ \mathcal{I}\big)$$

$$\mathsf{cps}_{AE}\ \big[ae\big] = ae$$

where for any k, $\mathcal{A}_k$ is defined as

$$\begin{aligned}\mathcal{A}_k\big[ae\big] &= k\ ae\\ \mathcal{A}_k\big[ce\big] &= \mathbf{let}\ x = ce\ \mathbf{in}\ k\ x \quad \text{where } x \text{ is fresh}\end{aligned}$$

and

$$\begin{aligned}&\mathsf{renorm}\big[\mathbf{let'}\ x = ce\ \mathbf{in}\ e\big] = \mathbf{let}\ x = ce\ \mathbf{in}\ e\\ &\mathsf{renorm}\big[\mathbf{let'}\ x = (\mathbf{let}\ x' = ce\ \mathbf{in}\ e')\ \mathbf{in}\ e\big] =\\ &\qquad \mathbf{let}\ x' = ce\ \mathbf{in}\ \mathsf{renorm}\big[\mathbf{let'}\ x = e'\ \mathbf{in}\ e\big]\end{aligned}$$

In the above equations, **let**' is a pseudo-construct used to make renormalization more readable. In essence, it is a non-ANF version of **let** where the bound expression is generalized to *Expr*. Note that renorm only works correctly if $x' \notin \mathsf{fv}(e)$, which is implied by our assumption that all bound variables are distinct.

4.2 Generalization of Continuations

The continuations resulting from the above CPS conversion expect to be applied to value terms. The next step is to generalize (or "lift") the continuations so that they recursively call the evaluator to evaluate non-value arguments. In other words, assuming the term type can be factored into values and computations $V + C$, we convert each continuation k with the type $V \to V$ into a continuation $k' : V + C \to V$ using the following schema:

$$\textbf{let rec } k'\ t = \textbf{case } t \textbf{ of } inl\ v \to k\ v \mid inr\ c \to eval\ c\ k'$$

The recursive clauses will correspond to congruence rules in the resulting small-step semantics.

The transformation works by finding the unique application site of the continuation and then inserting the corresponding call to *eval* in the non-value case.

$$\begin{aligned}
&\mathsf{gencont}_E\big[\textbf{letcont } k\ \langle\ x, \vec{x}^{\sigma}\ \rangle = e_k \textbf{ in } e\big] = \\
&\qquad \textbf{letcont } k\ \langle\ \hat{x}, \vec{x}^{\sigma}\ \rangle = \\
&\qquad\quad \textbf{case } \hat{x} \textbf{ of } \{ \\
&\qquad\qquad \textbf{val}(x) \to e_k\ \textbf{;} \\
&\qquad\qquad \textbf{ELSE}(\hat{x}) \to eval\ \langle\ \hat{x}, \vec{ae}^{\sigma}\ \rangle\ \vec{ae}^{\rho}\ ae_k \\
&\qquad\quad \} \\
&\quad \text{if } \mathsf{findApp}\ k\ e = eval\ \langle\ _, \vec{ae}^{\sigma}\ \rangle\ \vec{ae}^{\rho}\ ae_k
\end{aligned}$$

where

- $\mathsf{findApp}\ k\ e$ is the unique use site of the continuation k in expression e, that is, the *CExpr* where *eval* is applied with k as its continuation; and
- $\hat{x}$ is a fresh variable associated with x – it stands for "a term corresponding to (the value) x".

Following the CPS conversion, each named continuation is applied exactly once in e, so $\mathsf{findApp}\ k\ e$ is total and returns the continuation's unique use site. Moreover, because the continuation was originally defined and let-bound at that use site, all free variables in $\mathsf{findApp}\ k\ e$ are also free in the definition of k.

When performing this generalization transformation, we also modify tail positions in *eval* that return a value so that they wrap their result in the **val** constructor. That is, if the continuation parameter of *eval* is k, then we rewrite all sites applying k to a configuration as follows:

$$k\ \langle\ ae, \vec{ae}^{\sigma}\ \rangle \Rightarrow k\ \langle\ \textbf{val}(ae), \vec{ae}^{\sigma}\ \rangle$$

4.3 Argument Lifting in Continuations

In the next phase, we partially lift free variables in continuations to make them explicit arguments. We perform a *selective* lifting in that we avoid lifting non-term arguments to the evaluation function. These arguments represent entities that parameterize the evaluation of a term. If an entity is modified during evaluation, the modified entity variable gets lifted. In the running example of Section 2, such a lifting occurred for $kclo_1$.

Function lift specifies the transformation at the continuation definition site:

$$\mathsf{lift}\ \Xi\ \Delta\ [\mathbf{letcont}\ k\ = \boldsymbol{\lambda} x.\, e_k\ \mathbf{in}\ e] = \\ \mathbf{letcont}\ k\ =\ \boldsymbol{\lambda}\ x_1\ \ldots\ x_n\ x.\, (\mathsf{lift}\ \Xi'\ \Delta'\ [e_k])\ \mathbf{in}\ (\mathsf{lift}\ \Xi'\ \Delta'\ [e])$$

where

- $\Xi' = \Xi \cup \{k\}$
- $\{x_1, \ldots, x_n\} = \mathsf{fv}\, e_k \cup (\bigcup_{g \in (\mathsf{dom}\, \Delta\ \cap\ \mathsf{fv}\, e_k)} \Delta(g)) - \Xi'$
- $\Delta' = \Delta[k \mapsto (x_1, \ldots, x_n)]$

and at the continuation application site – recall that continuations are always applied fully, but at this point they are only applied to one argument:

$$\mathsf{lift}\ \Xi\ \Delta\ \big[k\ ae\big] =\ k\ x_1\ \ldots\ x_n\ (\mathsf{lift}\ \Xi\ \Delta\ \big[ae'\big])$$

if $k \in \mathsf{dom}\, \Delta$ and $\Delta(k) = (x_1, \ldots, x_n)$.

Our lifting function is a restricted version of a standard argument-lifting algorithm [19]. The first restriction is that we do not lift all free variables, since we do not aim to float and lift the continuations to the top-level of the program, only to the top-level of the evaluation function. The other difference is that we can use a simpler way to compute the set of lifted parameters due to the absence of mutual recursion between continuations. The correctness of this can be proved using the approach of Fischbach [16].

4.4 Continuations Switch Control Directly

At this point, continuations handle the full evaluation of a term themselves. Instead of calling *eval* with the continuation as an argument, we can call the continuation directly to switch control between evaluation stages of a term. We will replace original *eval* call sites with direct applications of the corresponding continuations. The recursive call to *eval* in congruence cases of continuations will be left untouched, as this is where the continuation's argument will be evaluated to a value. Following from the continuation generalization transformation, this call to *eval* is with the same arguments as in the original site (which we are now replacing). In particular, the *eval* is invoked with the same $\vec{ae}^{\rho}$ arguments in the continuation body as in the original call site.

$$\mathsf{directcont}_E\ \big[\mathbf{letcont}\ k = ce\ \mathbf{in}\ e\big] K = \\ \mathbf{letcont}\ k =\ \mathsf{directcont}_{CE}\ \big[ce\big] K\ \mathbf{in}\ \mathsf{directcont}_E\ \big[e\big] (K \uplus \{k\})$$

$$\mathsf{directcont}_{CE}\big[eval\ \langle\ ae,\ \vec{ae}^{\sigma}\ \rangle\ \vec{ae}^{\rho}\ (\boldsymbol{\lambda} y.\ k\ \vec{x}\ y)\big]\ K = k\ \vec{x}\ \langle\ ae,\ \vec{ae}^{\sigma}\ \rangle \qquad \text{if } k \in K$$

4.5 Defunctionalization

Now we can move towards a first-order representation of continuations which can be further converted into term constructions. We defunctionalize continuations by first collecting all continuations in *eval*, then introducing corresponding constructors (the syntax), and finally generating an *apply* function (the semantics). The collection function accumulates continuation names and their definitions. At the same time it removes the definitions.

$$\begin{aligned}\mathsf{collect}_E\big[\mathbf{letcont}\ k = ce\ \mathbf{in}\ e\big] &= (\{(k, ce')\} \cup K_{ce} \cup K_e, e') \\ \text{where } (K_{ce}, ce') &= \mathsf{collect}_{CE}\big[ce\big] \\ (K_e, e') &= \mathsf{collect}_E\big[e\big]\end{aligned}$$

We reuse continuation names for constructors. The *apply* function is generated by simply generating a case analysis on the constructors and reusing the argument names from the continuation function arguments. In addition to the defunctionalized continuations, the generated *apply* function will take the same arguments as *eval*. Because of the absence of mutual recursion in our meta-language, *apply* takes *eval* as an argument.

$$\begin{aligned}&\mathsf{genApply}\ \ \vec{x}^{\rho}\ \vec{x}^{\sigma}\ \ k_{top}\ \big\{(k_1, \lambda p_{1,1}\ \ldots p_{1,i}.\ e_1), \ldots, (k_n, \lambda p_{n,1}\ \ldots\ p_{n,j}.\ e_n)\big\} = \\ &\qquad \boldsymbol{\lambda} eval\ \langle\ x_k, \vec{x}^{\sigma}\ \rangle\ \vec{x}^{\rho}\ \ k_{top}. \\ &\qquad\quad \mathbf{case}\ x_k\ \mathbf{of}\ \{ \\ &\qquad\qquad k_1(p_{1,1}, \ldots, p_{1,i}) \rightarrow\ e_1\ \mathbf{;} \\ &\qquad\qquad \ldots\ \mathbf{;} \\ &\qquad\qquad k_n(p_{n,1}, \ldots, p_{n,j}) \rightarrow\ e_n \\ &\qquad\quad \}\end{aligned}$$

Now we need a way to replace calls to continuations with corresponding calls to *apply*. For $\vec{ae}^{\rho}$ and k_{top} we use the arguments passed to *eval* or *apply* (depending on where we are replacing).

$$\mathsf{replace}_{CE}\big[k\ \vec{ae}_k\ \langle ae, \vec{ae}^{\sigma}\rangle\big](\vec{x}^{\rho}, k_{top}) = apply\ eval\ \langle\ k(\vec{ae}_k, ae),\ \vec{ae}^{\sigma}\ \rangle\ \vec{x}^{\rho}\ \ k_{top}$$

Finally, the complete defunctionalization is defined in terms of the above three functions.

4.6 Remove Self-recursive Tail-Calls

This is the transformation which converts a recursive evaluator into a stepping function. The transformation itself is very simple: we simply replace the self-recursive calls to *apply* in congruence cases.

$$\begin{aligned}&\mathsf{derec}_{CE}\big[eval\ \langle\ ae,\ \vec{ae}^{\sigma}\ \rangle\ \vec{ae}^{\rho}\ (\boldsymbol{\lambda}\langle\ x',\ \vec{x}^{\sigma\prime}\ \rangle.\ apply\ eval\ \langle\ c^{\mathrm{K}}(\vec{ae},\ x'),\ \vec{x}^{\sigma\prime}\ \rangle\ \vec{ae}^{\rho\prime}\ k)\big] = \\ &\qquad eval\ \langle\ ae,\ \vec{ae}^{\sigma}\ \rangle\ \vec{ae}^{\rho}\ (\boldsymbol{\lambda}\langle\ x',\ \vec{x}^{\sigma\prime}\ \rangle.\ k\ \langle\ c^{\mathrm{K}}(\vec{ae},\ x'),\ \vec{x}^{\sigma\prime}\ \rangle)\end{aligned}$$

Note, that we still leave those invocations of *apply* that serve to switch control through the stages of evaluation. Unless a continuation constructor will become a part of the output language, its application will be inlined in the final phase of our transformation.

4.7 Convert Continuations to Terms

After defunctionalization, we effectively have two sorts of terms: those constructed using the original constructors and those constructed using continuation constructors. Terms in these two sorts are given their semantics by the *eval* and *apply* functions, respectively. To get only one evaluator function at the end of our transformation process, we will join these two sorts, adding extra continuation constructors as new term constructors. We could simply merge *apply* to *eval*, however, this would give us many overlapping constructors. For example, in Section 2, we established that $\mathsf{kapp1}(e_2, e_1) \approx \mathsf{app}(e_1, e_2)$ and $\mathsf{kapp2}(v_1, e_2) \approx \mathsf{app}(\mathsf{val}(v_1), e_2)$. The inference of equivalent term constructors is guided by the following simple principle. For each continuation term $c^{\mathrm{K}}(ae_1, \ldots, ae_n)$ we are looking for a term $c'(ae'_1, \ldots, ae'_m)$, such that, for all $\vec{ae}^{\,\sigma}$, $\vec{ae}^{\,\rho}$ and ae_k

$$\begin{aligned} &\mathit{apply}\ \mathit{eval}\ \langle\ c^{\mathrm{K}}(ae_1, \ldots, ae_n), \vec{ae}^{\,\sigma}\ \rangle\ \vec{ae}^{\,\rho}\ ae_k \\ &\qquad = \mathit{eval}\ \langle\ c'(ae'_1, \ldots, ae'_m), \vec{ae}^{\,\sigma}\ \rangle\ \vec{ae}^{\,\rho}\ ae_k \end{aligned}$$

In our current implementation, we use a conservative approach where, starting from the cases in *eval*, we search for continuations reachable along a control flow path. Variables appearing in the original term are instantiated along the way. Moreover, we collect variables dependent on configuration entities (state). If control flow is split based on information derived from the state, we automatically include any continuation constructors reachable from that point as new constructors in the resulting language and interpreter. This, together with how information flows from the top-level term to subterms in congruence cases, preserves the coupling between state and corresponding subterms between steps.

If, starting from an input term $c(\vec{x})$, an invocation of *apply* on a continuation term $c^{\mathrm{K}}(\vec{ae}_k)$ is reached, and if, after instantiating the variables in the input term $c(\vec{ae})$, the sets of their free variables are equal, then we can introduce a translation from $c^{\mathrm{K}}(\vec{ae}_k)$ into $c(\vec{ae})$. If such a direct path is not found, the c^{K} will become a new term constructor in the language and a case in *eval* is introduced such that the above equation is satisfied.

4.8 Inlining, Simplification and Conversion to Direct Style

To finalize the generation of a small-step interpreter, we inline all invocations of *apply* and simplify the final program. After this, the interpreter will consists of only the *eval* function, still in continuation-passing style. To convert the interpreter to direct style, we simply substitute *eval*'s continuation variable for

$(\boldsymbol{\lambda} x.x)$ and reduce the new redexes. Then we remove the continuation argument performing rewrites following the scheme:

$$\mathit{eval}\ \vec{ae}\ (\boldsymbol{\lambda} bn.\ e)\ \Rightarrow\ \textbf{let}\ bn\ =\ \mathit{eval}\ \vec{ae}\ \textbf{in}\ e$$

Finally, we remove the reflexive case on values (i.e., $\textsf{val}(v) \longrightarrow \textsf{val}(v)$). At this point we have a small-step interpreter in direct form.

4.9 Removing Vacuous Continuations

After performing the above transformation steps, we may end up with some redundant term constructors, which we call "empty" or vacuous. These are constructors which only have one argument and their semantics is equivalent to the argument itself, save for an extra step which returns the computed value. In other words, they are unary constructs which only have two rules in the resulting small-step semantics matching the following pattern.

$$\frac{}{\vec{\rho} \vdash \langle c(\textsf{val}(v)), \vec{\sigma}\rangle \longrightarrow \langle \textsf{val}(v), \vec{\sigma}\rangle} \qquad \frac{\vec{\rho} \vdash \langle e, \vec{\sigma}\rangle \longrightarrow \langle e', \vec{\sigma}'\rangle}{\vec{\rho} \vdash \langle c(e), \vec{\sigma}\rangle \longrightarrow \langle c(e'), \vec{\sigma}'\rangle}$$

Such a construct will result from a continuation, which, even after generalization and argument lifting, merely evaluates its sole argument and returns the corresponding value:

$$\begin{array}{l}\textbf{letcont rec}\ k_i\ e = \textbf{case}\ e\ \textbf{of}\ \{\\ \quad \textsf{val}(v) \longrightarrow k\ v\ \mid\\ \quad \textsf{ELSE}(e) \longrightarrow \mathit{eval}\ e\ (\boldsymbol{\lambda} e'.\ k_i\ e')\\ \}\end{array}$$

These continuations can be easily identified and removed once argument lifting is performed, or at any point in the transformation pipeline, up until *apply* is absorbed into *eval*.

4.10 Detour: Generating Pretty-Big-Step Semantics

It is interesting to see what kind of semantics we get by rearranging or removing some steps of the above process. If, after CPS conversion, we do not generalize the continuations, but instead just lift their arguments and defunctionalize them,[1] we obtain a *pretty-big-step* [6] interpreter. The distinguishing feature of pretty-big-step semantics is that constructs which would normally have rules with multiple premises are factorized into intermediate constructs. As observed by Charguéraud, each intermediate construct corresponds to an intermediate state of the interpreter, which is why, in turn, they naturally correspond to continuations. Here are the pretty-big-step rules generated from the big-step semantics in Fig. 2 (Section 2).

[1] The complete transformation to pretty-big-step style involves these steps: 1. CPS conversion, 2. argument lifting, 3. removal of vacuous continuations, 4. defunctionalization, 5. merging of apply and eval, and 6. conversion to direct style.

$$\frac{}{\rho \vdash \textbf{val}(v) \Downarrow_B^P v} \qquad \frac{\rho \vdash e_1 \Downarrow_B^P v_1 \quad \rho \vdash \textbf{kapp1}(e_2, v_1) \Downarrow_B^P v}{\rho \vdash \textbf{app}(e_1, e_2) \Downarrow_B^P v}$$

$$\frac{v = \rho\ x}{\rho \vdash \textbf{var}(x) \Downarrow_B^P v} \qquad \frac{\rho \vdash e_2 \Downarrow_B^P v_2 \quad \rho \vdash \textbf{kapp2}(v_1, v_2) \Downarrow_B^P v}{\rho \vdash \textbf{kapp1}(e_2, v_1) \Downarrow_B^P v}$$

$$\frac{}{\rho \vdash \textbf{lam}(x, e') \Downarrow_B^P \textbf{clo}(x, e', \rho)} \qquad \frac{\rho'' = \mathsf{update}\ x\ v_2\ \rho' \quad \rho'' \vdash e' \Downarrow_B^P v}{\rho \vdash \textbf{kapp2}(\textbf{clo}(x, e', \rho'), v_2) \Downarrow_B^P v}$$

As we can see, the evaluation of **app** now proceeds through two intermediate constructs, **kapp1** and **kapp2**, which correspond to continuations introduced in the CPS conversion. The evaluation of $\textbf{app}(e_1, e_2)$ starts by evaluating e_1 to v_1. Then **kapp1** is responsible for evaluating e_2 to v_2. Finally, **kapp2** evaluates the closure body just as the third premise of the original rule for **app**. Save for different order of arguments, the resulting intermediate constructs and their rules are identical to Charguéraud's examples.

4.11 Pretty-Printing

For the purpose of presenting and studying the original and transformed semantics, we add a final pretty-printing phase. This amounts to generating inference rules corresponding to the control flow in the interpreter. This pretty-printing stage can be applied to both the big-step and small-step interpreters and was used to generate many of the rules in this paper, as well as for generating the appendix of the full version of this paper [1].

4.12 Correctness

A correctness proof for the full pipeline is not part of our current work. However, several of these steps (partial CPS conversion, partial argument lifting, defunctionalization, conversion to direct style) are instances of well-established techniques. In other cases, such as generalization of continuations (Section 4.2) and removal of self-recursive tail-calls (Section 4.6), we have informal proofs using equational reasoning [1]. The proof for tail-call removal is currently restricted to compositional interpreters.

5 Evaluation

We have evaluated our approach to deriving small-step interpreters on a range of example languages. Table 2 presents an overview of example big-step specifications and their properties, together with their derived small-step counterparts. A full listing of the input and output specifications for these case studies appears in the appendix to the full version of the paper, which is available online [1].

Table 2. Overview of transformed example languages. Input is a given big-step interpreter and our transformations produce a small-step counterpart as output automatically. "Prems" columns only list structural premises: those that check for a big or small step. Unless otherwise stated, environments are used to give meaning to variables and they are represented as functions.

	Big-step		Small-step			
Example	Rules	Prems	Rules	Prems	New	Features
Call-by-value	4	3	7	3	1	
Call-by-value, substitution	4	5	7	4	0	addition
Call-by-value, booleans	13	20	24	11	1	add., conditional, equality
Call-by-value, pairs	7	7	14	7	1	pairs, left/right projection
Call-by-value, dynamic scopes	5	5	10	5	1	add., defunctionalized environments (DEs)
Call-by-value, recursion & iteration	26	44	57	26	6	fixpoint operator, add., sub., let-expressions, applicative for and while loops, cond., strict and "lazy" conjunction, eq., pairs
Call-by-name	5	5	11	5	2	add., DEs
Call-by-name, substitution	4	4	6	3	0	add., DEs
Call-by-name, booleans	13	20	25	11	2	add., cond., eq., DEs
Call-by-name, pairs	7	7	15	7	2	pairs, left/right proj., DEs
Minimal imperative	4	4	6	3	0	add., store without indirection, combined assignment *with* sequencing
While	7	9	14	6	2	add., store w/o indir., assign., seq., while
While, environments	8	10	17	7	3	add., store w/ indir., scoped var. declaration, assign., seq., while
Extended While	17	26	33	15	2	add., subt., mult., seq., store w/o indir., while, cond., "ints as bools", equality, "lazy conj."
Exceptions as state	8	7	11	3	1	add.
Exceptions as values	8	7	10	3	0	add.
Call-by-value, exceptions	21	29	34	12	2	add., div., try block
CBV, exceptions as state	20	26	39	11	8	add., div., handle & try blocks
CBV, non-determinism	7	7	13	5	2	add., choice operator
Store rewinding	8	10	19	8	4	assign., rewinding of the store

For our case studies, we have used call-by-value and call-by-name λ-calculi, and a simple imperative language as base languages and extended them with some common features. Overall, the small-step specifications (as well as the corresponding interpreters) resulting from our transformation are very similar to ones we could find in the literature. The differences are either well justified—for example, by different handling of value terms—or they are due to new term constructors which could be potentially eliminated by a more powerful translation.

We evaluated the correctness of our transformation experimentally, by comparing runs of the original big-step and the transformed small-step interpreters, as well as by inspecting the interpreters themselves. In a few cases, we proved the transformation correct by transcribing the input and output interpreters in Coq (as an evaluation relation coupled with a proof of determinism) and proving them equivalent. From the examples in Table 2, we have done so for "Call-by-value", "Exceptions as state", and a simplified version of "CBV, exceptions as state".

We make a few observations about the resulting semantics here.

New Auxiliary Constructs. In languages that use an environment to look up values bound to variables, new constructs are introduced to keep the updated environment as context. These constructs are simple: they have two arguments – one for the environment (context) and one for the term to be evaluated in that environment. A congruence rule will ensure steps of the term argument in the given context and another rule will return the result. The construct **kclo1** from the λ-calculus based examples is a typical example.

$$\frac{}{\rho \vdash \mathbf{kclo1}(\rho', \mathbf{val}(v)) \rightarrow \mathbf{val}(v)} \qquad \frac{\rho' \vdash t \rightarrow t'}{\rho \vdash \mathbf{kclo1}(\rho', t) \rightarrow \mathbf{kclo1}(\rho', t')}$$

As observed in Section 2, if the environment ρ'' is a result of updating an environment ρ' with a binding of x to v, then the **app** rule

$$\frac{\rho'' = \mathsf{update}\ x\ v\ \rho'}{\rho \vdash \mathbf{app}(\mathbf{clo}(\rho', x, e), v) \rightarrow \mathbf{kclo1}(\rho'', e)}$$

and the above two rules can be replaced with the following rules for **app**:

$$\frac{}{\rho \vdash \mathbf{app}(\mathbf{clo}(x, v, \rho'), v_2) \rightarrow v} \qquad \frac{\rho'' = \mathsf{update}\ x\ v_2\ \rho' \qquad \rho'' \vdash e \rightarrow e'}{\rho \vdash \mathbf{app}(\mathbf{clo}(x, e, \rho'), v_2) \rightarrow \mathbf{app}(\mathbf{clo}(x, e', \rho'), v_2)}$$

Another common type of constructs resulting in a recurring pattern of extra auxiliary constructs are loops. For example, the "While" language listed in Table 2 contains a while-loop with the following big-step rules:

$$\frac{\langle e_b, \sigma\rangle \Downarrow \langle \mathbf{false}, \sigma'\rangle}{\langle \mathbf{while}(e_b, c), \sigma\rangle \Downarrow \langle \mathbf{skip}, \sigma'\rangle}$$

$$\frac{\langle e_b, \sigma\rangle \Downarrow \langle \mathbf{true}, \sigma'\rangle \qquad \langle c, \sigma'\rangle \Downarrow \langle \mathbf{skip}, \sigma''\rangle \qquad \langle \mathbf{while}(e_b, c), \sigma''\rangle \Downarrow \langle v, \sigma'''\rangle}{\langle \mathbf{while}(e_b, c), \sigma\rangle \Downarrow \langle v, \sigma'''\rangle}$$

The automatic transformation of these rules introduces two extra constructs, **kwhile1** and **ktrue1**. The former ensures the full evaluation of the condition expression, keeping a copy of it together with the while's body. The latter construct ensures the full evaluation of while's body, keeping a copy of the body together with the condition expression.

$$\frac{}{\langle \mathbf{while}(e_b, c),\ \sigma\rangle \to \langle \mathbf{kwhile1}(c, e_b, e_b),\ \sigma\rangle}$$

$$\frac{}{\langle \mathbf{kwhile1}(c, e_b, \mathbf{true}),\ \sigma\rangle \to \langle \mathbf{ktrue1}(e_b, c, c),\ \sigma\rangle}$$

$$\frac{}{\langle \mathbf{kwhile1}(c, e_b, \mathbf{false}),\ \sigma\rangle \to \langle \mathbf{skip},\ \sigma\rangle}$$

$$\frac{\langle t,\ \sigma\rangle \to \langle t',\ \sigma'\rangle}{\langle \mathbf{kwhile1}(c, e_b, t),\ \sigma\rangle \to \langle \mathbf{kwhile1}(c, e_b, t'),\ \sigma'\rangle}$$

$$\frac{}{\langle \mathbf{ktrue1}(e_b, c, \mathbf{skip}),\ \sigma\rangle \to \langle \mathbf{while}(e_b, c),\ \sigma\rangle}$$

$$\frac{\langle t,\ \sigma\rangle \to \langle t',\ \sigma'\rangle}{\langle \mathbf{ktrue1}(e_b, c, t),\ \sigma\rangle \to \langle \mathbf{ktrue1}(e_b, c, t'),\ \sigma'\rangle}$$

We observe that in a language with a conditional and a sequencing construct we can find terms corresponding to **kwhile1** and **ktrue1:**

$$\mathbf{kwhile1}(c, e_b, e_b') \approx \mathbf{if}(e_b', \mathbf{seq}(c, \mathbf{while}(e_b, c)), \mathbf{skip})$$
$$\mathbf{ktrue1}(e_b, c, c') \approx \mathbf{seq}(c', \mathbf{while}(e_b, c))$$

The small-step semantics of **while** could then be simplified to a single rule.

$$\frac{}{\langle \mathbf{while}(e_b, c), \sigma\rangle \to \langle \mathbf{if}(e_b, \mathbf{seq}(c, \mathbf{while}(e_b, c)), \mathbf{skip}), \sigma\rangle}$$

Our current, straightforward way of deriving term–continuation equivalents is not capable of finding these equivalences. In future work, we want to explore external tools, such as SMT solvers, to facilitate searching for translations from continuations to terms. This search could be possibly limited to a specific term depth.

Exceptions as Values. We tested our transformations with two ways of representing exceptions in big-step semantics currently supported by our input language: as values and as state. Representing exceptions as values appears to be more common and is used, for example, in the big-step specification of Standard ML [24], or in [6] in connection with *pretty big-step semantics*. Given a big-step specification (or interpreter) in this style, the generated small-step semantics handles exceptions correctly (based on our experiments). However, since exceptions are just values, propagation to top-level is spread out across multiple steps – depending on the depth of the term which raised the exception. The following example illustrates this behavior.

$$\begin{aligned}\mathbf{add}(1, \mathbf{add}(2, \mathbf{add}(\mathbf{raise}(3), \mathbf{raise}(4)))) &\to \mathbf{add}(1, \mathbf{add}(2, \mathbf{add}(\mathbf{exc}(3), \mathbf{raise}(4))))\\ &\to \mathbf{add}(1, \mathbf{add}(2, \mathbf{exc}(3))) \to \mathbf{add}(1, \mathbf{exc}(3)) \to \mathbf{exc}(3)\end{aligned}$$

Since we expect the input semantics to be deterministic and the propagation of exceptions in the resulting small-step follows the original big-step semantics, this "slow" propagation is not a problem, even if it does not take advantage of "fast" propagation via labels or state. A possible solution we are considering for future work is to let the user flag values in the big-step semantics and translate such values as labels on arrows or a state change to allow propagating them in a single step.

Exceptions as State. Another approach to specifying exceptions is to use a flag in the configuration. Rules may be specified so that they only apply if the incoming state has no exception indicated. As with the exceptions-as-values approach, propagation rules have to be written to terminate a computation early if a computation of a subterm indicates an exception. Observe the exception propagation rule for **add** and the exception handling rule for **try**.

$$\frac{\langle e_1, \sigma, \mathsf{ok}\rangle \Downarrow \langle v_1, \sigma', \mathsf{ex}\rangle}{\langle \mathbf{app}(e_1, e_2), \sigma, \mathsf{ok}\rangle \Downarrow \langle \mathbf{skip}, \sigma', \mathsf{ex}\rangle}$$

$$\frac{\langle e_1, \sigma, \mathsf{ok}\rangle \Downarrow \langle v_1, \sigma', \mathsf{ex}\rangle \qquad \langle e_2, \sigma', \mathsf{ok}\rangle \Downarrow \langle v_2, \sigma'', \mathsf{ok}\rangle}{\langle \mathbf{try}(e_1, e_2), \sigma, \mathsf{ok}\rangle \Downarrow \langle v_2, \sigma'', \mathsf{ok}\rangle}$$

Using state to propagate exceptions is mentioned in connection with small-step SOS in [4]. While this approach has the potential advantage of manifesting the currently raised exception immediately at the top-level, it also poses a problem of locality. If an exception is reinserted into the configuration, it might become decoupled from the original site. This can result, for example, in the wrong handler catching the exception in a following step. Our transformation deals with this style of exceptions naturally by preserving more continuations in the final interpreter. After being raised, an exception is inserted into the state and propagated to top-level by congruence rules. However, it will only be caught after the corresponding subterm has been evaluated, or rather, a value has been propagated upwards to signal a completed computation. This behavior corresponds to exception handling in big-step rules, only it is spread out over multiple steps. Continuations are kept in the final language to correspond to stages of computation and thus, to preserve the locality of a raised exception. A handler will only handle an exception once the raising subterm has become a value. Hence, the exception will be intercepted by the innermost handler – even if the exception is visible at the top-level of a step.

Based on our experiments, the exception-as-state handling in the generated small-step interpreters is a truthful unfolding of the big-step evaluation process. This is further supported by our ad-hoc proofs of equivalence between input and output interpreters. However, the generated semantics suffers from a blowup in the number of rules and moves away from the usual small-step propagation and exception handling in congruence rules. We see this as a shortcoming of the transformation. To overcome this, we briefly experimented with a case-floating stage,

which would result in catching exceptions in the congruence cases of continuations. Using such transformation, the resulting interpreter would more closely mirror the standard small-step treatment of exceptions as signals. However, the conditions when this transformations should be triggered need to be considered carefully and we leave this for future work.

Limited Non-determinism. In the present work, our aim was to only consider deterministic semantics implemented as an interpreter in a functional programming language. However, since cases of the interpreter are considered independently in the transformation, some forms of non-determinism in the input semantics get translated correctly. For example, the following internal choice construct (cf. CSP's $\sqcap$ operator [5,17]) gets transformed correctly. The straightforward big-step rules are transformed into small-step rules as expected. Of course, one has to keep in mind that these rules are interpreted as ordered, that is, the first rule in both styles will always apply.

$$\frac{e_1 \Downarrow v_1}{\mathsf{choose}(e_1, e_2) \Downarrow v_1} \qquad \frac{}{\mathsf{choose}(e_1, e_2) \rightarrow e_1}$$

$$\frac{e_2 \Downarrow v_2}{\mathsf{choose}(e_1, e_2) \Downarrow v_2} \qquad \frac{}{\mathsf{choose}(e_1, e_2) \rightarrow e_2}$$

6 Related Work

In their short paper [18], the authors propose a direct syntactic way of deriving small-step rules from big-step ones. Unlike our approach, based on manipulating control flow in an interpreter, their transformation applies to a set of inference rules. While axioms are copied over directly, for conditional rules a stack is added to the configuration to keep track of evaluation. For each conditional big-step rule, an auxiliary construct and 4 small-step rules are generated. Results of "premise computations" are accumulated and side-conditions are only discharged at the end of such a computation sequence. For this reason, we can view the resulting semantics more as a "leap" semantics, which makes it less suitable for a semantics-based interpreter or debugger. A further disadvantage is that the resulting semantics is far removed from a typical small-step specification with a higher potential for blow-up as 4 rules are introduced for each conditional rule. On the other hand, the delayed unification of meta-variables and discharging of side-conditions potentially makes the transformation applicable to a wider array of languages, including those where control flow is not as explicit.

In [2], the author explores an approach to constructing abstract machines from big-step (natural) specifications. It applies to a class of big-step specifications called *L-attributed big-step semantics*, which allows for sufficiently interesting languages. The extracted abstract machines use a stack of evaluation contexts to keep track of the stages of computations. In contrast, our transformed interpreters rebuild the context via congruence rules in each step. While this is less efficient as a computation strategy, the intermediate results of the

computation are visible in the context of the original program, in line with usual SOS specifications.

A significant body of work has been developed on transformations that take a form of small-step semantics (usually an interpreter) and produce a big-step-style interpreter. The relation between semantic specifications, interpreters and abstract machines has been thoroughly investigated, mainly in the context of reduction semantics [10–13,26]. In particular, our work was inspired by and is based on Danvy's work on refocusing in reduction semantics [13] and on use of CPS conversion and defunctionalization to convert between representations of control in interpreters [11].

A more direct approach to deriving big-step semantics from small-step is taken by authors of [4], where a small-step Modular SOS specification is transformed into a pretty-big-step one. This is done by introducing reflexivity and transitivity rules into a specification, along with a "refocus" rule which effectively compresses a transition sequence into a single step. The original small-step rules are then specialized with respect to these new rules, yielding refocused rules in the style of pretty-big-step semantics [6]. A related approach is by Ciobâcă [7], where big-step rules are generated for a small-step semantics. The big-step rules are, again, close to a pretty-big-step style.

7 Conclusion and Future Work

We have presented a stepwise functional derivation of a small-step interpreter from a big-step one. This derivation proceeds through a sequence of, mostly basic, transformation steps. First, the big-step evaluation function is converted into continuation-passing style to make control-flow explicit. Then, the continuations are generalized (or lifted) to handle non-value inputs. The non-value cases correspond to congruence rules in small-step semantics. After defunctionalization, we remove self-recursive calls, effectively converting the recursive interpreter into a stepping function. The final major step of the transformation is to decide which continuations will have to be introduced as new auxiliary terms into the language. We have evaluated our approach on several languages covering different features. For most of these, the transformation yields small-step semantics which are close to ones we would normally write by hand.

We see this work as an initial exploration of automatic transformations of big-step semantics into small-step counterparts. We identified a few areas where the current process could be significantly improved. These include applying better equational reasoning to identify terms equivalent to continuations, or transforming exceptions as state in a way that would avoid introducing many intermediate terms and would better correspond to usual signal handling in small-step SOS. Another research avenue is to fully verify the transformations in an interactive theorem prover, with the possibility of extracting a correct transformer from the proofs.

Acknowledgements. We would like to thank Jeanne-Marie Musca, Brian LaChance and the anonymous referees for their useful comments and suggestions. This work was supported in part by DARPA award FA8750-15-2-0033.

References

1. https://www.eecs.tufts.edu/~fvesely/esop2019
2. Ager, M.S.: From natural semantics to abstract machines. In: Etalle, S. (ed.) LOPSTR 2004. LNCS, vol. 3573, pp. 245–261. Springer, Heidelberg (2005). https://doi.org/10.1007/11506676_16
3. Amin, N., Rompf, T.: Collapsing towers of interpreters. Proc. ACM Program. Lang. **2**(POPL), 52:1–52:33 (2017). https://doi.org/10.1145/3158140
4. Bach Poulsen, C., Mosses, P.D.: Deriving pretty-big-step semantics from small-step semantics. In: Shao, Z. (ed.) ESOP 2014. LNCS, vol. 8410, pp. 270–289. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54833-8_15
5. Brookes, S.D., Roscoe, A.W., Walker, D.J.: An operational semantics for CSP. Technical report, Oxford University (1986)
6. Charguéraud, A.: Pretty-big-step semantics. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 41–60. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_3
7. Ciobâcă, Ş.: From small-step semantics to big-step semantics, automatically. In: Johnsen, E.B., Petre, L. (eds.) IFM 2013. LNCS, vol. 7940, pp. 347–361. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38613-8_24
8. Danvy, O., Filinski, A.: Representing control: a study of the CPS transformation. Math. Struct. Comput. Sci. **2**(4), 361–391 (1992). https://doi.org/10.1017/S0960129500001535
9. Danvy, O.: On evaluation contexts, continuations, and the rest of computation. In: Thielecke, H. (ed.) Workshop on Continuations, pp. 13–23, Technical report CSR-04-1, Department of Computer Science, Queen Mary's College, Venice, Italy, January 2004
10. Danvy, O.: From reduction-based to reduction-free normalization. Electr. Notes Theor. Comput. Sci. **124**(2), 79–100 (2005). https://doi.org/10.1016/j.entcs.2005.01.007
11. Danvy, O.: Defunctionalized interpreters for programming languages. In: ICFP 2008, pp. 131–142. ACM, New York (2008). https://doi.org/10.1145/1411204.1411206
12. Danvy, O., Johannsen, J., Zerny, I.: A walk in the semantic park. In: PEPM 2011, pp. 1–12. ACM, New York (2011). https://doi.org/10.1145/1929501.1929503
13. Danvy, O., Nielsen, L.R.: Refocusing in reduction semantics. Technical report, BRICS RS-04-26, DAIMI, Department of Computer Science, University of Aarhus, November 2004
14. Ellison, C., Roşu, G.: An executable formal semantics of C with applications. In: POPL 2012, pp. 533–544. ACM, New York (2012). https://doi.org/10.1145/2103656.2103719
15. Felleisen, M., Findler, R.B., Flatt, M.: Semantics Engineering with PLT Redex, 1st edn. The MIT Press, Cambridge (2009)
16. Fischbach, A., Hannan, J.: Specification and correctness of lambda lifting. J. Funct. Program. **13**(3), 509–543 (2003). https://doi.org/10.1017/S0956796802004604
17. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall Inc., Upper Saddle River (1985)

18. Huizing, C., Koymans, R., Kuiper, R.: A small step for mankind. In: Dams, D., Hannemann, U., Steffen, M. (eds.) Concurrency, Compositionality, and Correctness. LNCS, vol. 5930, pp. 66–73. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11512-7_5
19. Johnsson, T.: Lambda lifting: transforming programs to recursive equations. In: Jouannaud, J.-P. (ed.) FPCA 1985. LNCS, vol. 201, pp. 190–203. Springer, Heidelberg (1985). https://doi.org/10.1007/3-540-15975-4_37
20. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine, and compiler. ACM Trans. Program. Lang. Syst. **28**(4), 619–695 (2006). https://doi.org/10.1145/1146809.1146811
21. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: a verified implementation of ML. In: POPL 2014, pp. 179–191. ACM, New York (2014). https://doi.org/10.1145/2535838.2535841
22. Leroy, X., Grall, H.: Coinductive big-step operational semantics. Inf. Comput. **207**(2), 284–304 (2009). https://doi.org/10.1016/j.ic.2007.12.004
23. Midtgaard, J., Ramsey, N., Larsen, B.: Engineering definitional interpreters. In: PPDP 2013, pp. 121–132. ACM, New York (2013). https://doi.org/10.1145/2505879.2505894
24. Milner, R., Tofte, M., Macqueen, D.: The Definition of Standard ML. MIT Press, Cambridge (1997)
25. Nielsen, L.R.: A selective CPS transformation. Electr. Notes Theor. Comput. Sci. **45**, 311–331 (2001). https://doi.org/10.1016/S1571-0661(04)80969-1
26. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. High. Order Symbolic Comput. **11**(4), 363–397 (1998). https://doi.org/10.1023/A:1010027404223
27. Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. J. Logic Algebraic Program. **79**(6), 397–434 (2010). https://doi.org/10.1016/j.jlap.2010.03.012
28. Strachey, C., Wadsworth, C.P.: Continuations: a mathematical semantics for handling full jumps. High. Order Symbolic Comput. **13**(1), 135–152 (2000). https://doi.org/10.1023/A:1010026413531
29. Wright, A., Felleisen, M.: A syntactic approach to type soundness. Inf. Comput. **115**(1), 38–94 (1994). https://doi.org/10.1006/inco.1994.1093