

Making the stack explicit: the continuation-passing style transformation

MPRI 2.4

François Pottier



2024

What if a program transformation could:

- ensure that every function call is a **tail call** and the **stack** is **explicit**, so the code is no longer really recursive, but **iterative**;
- make the evaluation order **explicit** in the code, so that it does not depend on the ambient strategy (CBN / CBV);
- eliminate the apparent **redundancy** between calls and returns, by exploiting solely function calls – **functions never return!**
- suggest extending the λ -calculus with **control operators**?

What if a program transformation could:

- ensure that every function call is a **tail call** and the **stack** is **explicit**, so the code is no longer really recursive, but **iterative**;
- make the evaluation order **explicit** in the code, so that it does not depend on the ambient strategy (CBN / CBV);
- eliminate the apparent **redundancy** between calls and returns, by exploiting solely function calls – **functions never return!**
- suggest extending the λ -calculus with **control operators**?

The **continuation-passing style** transformation does all this.

Motivation



D. Conversion to Continuation-Passing Style

This phase is the real meat of the compilation process. It is of interest primarily in that it transforms a program written in SCHEME into an equivalent program (the continuation-passing-style version, or CPS version), written in a language isomorphic to a subset of SCHEME with the property that interpreting it requires no control stack or other unbounded temporary storage and no decisions as to the order of evaluation of (non-trivial) subexpressions. The importance of these properties cannot be overemphasized. The fact that it is essentially a subset of SCHEME implies that its semantics are as clean, elegant, and well-understood as those of the original language. It is easy to build an

Steele, **RABBIT: a compiler for SCHEME**, 1978.

1 Example: from a direct-style interpreter down to an abstract machine

2 Formalization

3 Remarks

A direct-style interpreter

Example

Formalization

Remarks

Recall our environment-based interpreter for call-by-value λ -calculus:

```
let rec eval (e : cenv) (t : term) : cvalue =  
  match t with  
  | Var x ->  
    lookup e x  
  | Lam t ->  
    Clo (t, e)  
  | App (t1, t2) ->  
    let cv1 = eval e t1 in  
    let cv2 = eval e t2 in  
    let Clo (u1, e') = cv1 in  
    eval (cv2 :: e') u1
```

This is an OCaml transcription, without a fuel parameter.

A continuation-passing style interpreter

[Example](#)[Formalization](#)[Remarks](#)

Instead of **returning** a value,

```
let rec eval (e : cenv) (t : term) : cvalue =  
  ...
```

let's **pass** this value to a **continuation** that we get as an argument:

```
let rec evalk (e : cenv) (t : term) (k : cvalue -> 'a) : 'a =  
  ...
```

Exercise (in class): write evalk. (See [EvalCBVExercise](#).)

A continuation-passing style interpreter

Example

Formalization

Remarks

```
let rec evalk (e : cenv) (t : term) (k : cvalue -> 'a) : 'a =  
  match t with  
  | Var x ->  
    k (lookup e x)  
  | Lam t ->  
    k (Clo (t, e))  
  | App (t1, t2) ->  
    evalk e t1 (fun cv1 ->  
      evalk e t2 (fun cv2 ->  
        let Clo (u1, e') = cv1 in  
        evalk (cv2 :: e') u1 k))
```

Instead of **returning** a value, **pass** it to k.

Instead of **sequencing** computations via **let**, **nest** continuations.

A continuation-passing style interpreter

[Example](#)[Formalization](#)[Remarks](#)

To run the interpreter, start it with the [identity](#) continuation:

```
let eval (e : cenv) (t : term) : cvalue =  
  evalk e t (fun cv -> cv)
```

Correctness of the CPS interpreter

Example

Formalization

Remarks

The continuation-passing style interpreter is “obviously” correct.

Exercise: define `evalk` in Coq (with `fuel`) and prove it equivalent to the direct-style interpreter: `evalk n e t k = k (eval n e t)`.

Properties of the interpreter

What is special about this interpreter?

Properties of the interpreter

Example

Formalization

Remarks

What is special about this interpreter?

- Every call to `evalk` is a **tail call**.
- Every call to a continuation `k` is a **tail call**.

A call $g\ x$ is a tail call if it is the “last thing” that the calling function does...

More formally,

$v ::= x \mid \lambda x. tt$	values
$tt ::=$	terms in tail position
$ \ v$	
$ \ nt\ nt$	– a tail call
$ \ let\ nt\ in\ tt$	
$ \ if\ nt\ then\ tt\ else\ tt$	
$nt ::=$	terms not in tail position
$ \ v$	
$ \ nt\ nt$	– not a tail call
$ \ let\ nt\ in\ nt$	
$ \ if\ nt\ then\ nt\ else\ nt$	

This can be understood as the description of a top-down computation that assigns a Boolean flag (“tail” or “non-tail”) to every subterm.

OCaml allows us to **verify** that these are indeed tail calls:

```
let rec evalk (e : cenv) (t : term) (k : cvalue -> 'a) : 'a =  
  match t with  
  | Var x ->  
    (k[@tailcall]) (lookup e x)  
  | Lam t ->  
    (k[@tailcall]) (Clo (t, e))  
  | App (t1, t2) ->  
    (evalk[@tailcall]) e t1 (fun cv1 ->  
      (evalk[@tailcall]) e t2 (fun cv2 ->  
        let Clo (u1, e') = cv1 in  
        (evalk[@tailcall]) (cv2 :: e') u1 k))
```

A nice feature (though with somewhat ugly syntax).

Properties of the interpreter

Example

Formalization

Remarks

Tail calls are compiled by OCaml to **jumps**.

Thus, tail-recursive functions are compiled by OCaml to **loops**.

Steele, **Lambda: the ultimate GOTO**, 1977.

Thus, the CPS interpreter is not truly **recursive**: it is **iterative**.

It uses **constant space** on OCaml's implicit stack.

Wait! Does the interpreter really **not need a stack** any more?

Properties of the interpreter

Example

Formalization

Remarks

Tail calls are compiled by OCaml to **jumps**.

Thus, tail-recursive functions are compiled by OCaml to **loops**.

Steele, **Lambda: the ultimate GOTO**, 1977.

Thus, the CPS interpreter is not truly **recursive**: it is **iterative**.

It uses **constant space** on OCaml's implicit stack.

Wait! Does the interpreter really **not need a stack** any more?

- Of course it **does** need a stack.

Properties of the interpreter

Example

Formalization

Remarks

Tail calls are compiled by OCaml to **jumps**.

Thus, tail-recursive functions are compiled by OCaml to **loops**.

Steele, **Lambda: the ultimate GOTO**, 1977.

Thus, the CPS interpreter is not truly **recursive**: it is **iterative**.

It uses **constant space** on OCaml's implicit stack.

Wait! Does the interpreter really **not need a stack** any more?

- Of course it **does** need a stack.
- The **continuation**, allocated in the OCaml heap, serves as a stack.

A defunctionalized CPS interpreter

To better see the structure of the continuation,
let us **defunctionalize** the CPS interpreter.

Reynolds, **Definitional interpreters**
for programming languages, 1972 (1998).

Reynolds, **Definitional interpreters revisited**, 1998.

Defunctionalization (reminder)

Steps:

- Identify the sites where closures are allocated, that is, where anonymous functions are built.
- Compute, at each site, the free variables of the anonymous function.
- Introduce an algebraic data type of closures.
- Transform the code:
 - replace anonymous functions with constructor applications,
 - replace function applications with calls to `apply`,
 - and define `apply`.

Exercise (in class): defunctionalize the CPS interpreter. ([EvalCBVExercise.](#))

A defunctionalized CPS interpreter

[Example](#)[Formalization](#)[Remarks](#)

There are three sites where an anonymous continuation is built.

We name them and compute their free variables.

This leads to the following algebraic data type of continuations:

```
type kont =  
  | AppL of { e: cenv; t2: term; k: kont }  
  | AppR of {          cv1: cvalue; k: kont }  
  | Init
```

What data structure is this?

A defunctionalized CPS interpreter

[Example](#)[Formalization](#)[Remarks](#)

There are three sites where an anonymous continuation is built.

We name them and compute their free variables.

This leads to the following algebraic data type of continuations:

```
type kont =  
  | AppL of { e: cenv; t2: term; k: kont }  
  | AppR of {          cv1: cvalue; k: kont }  
  | Init
```

What data structure is this? A [linked list](#). A heap-allocated stack.

In fact, it is a (call-by-value) [evaluation context](#):

$$E ::= E[\] \ t_2[e] \mid E[v_1 \ \] \mid \ []$$

It is a [zipper](#), a path from the context's hole up to the root of a term.

Huet, [The Zipper](#), 1997.

A defunctionalized CPS interpreter

[Example](#)[Formalization](#)[Remarks](#)

We transform the interpreter's main function:

```
let rec evalkd (e : cenv) (t : term) (k : kont) : cvalue =  
  match t with  
  | Var x ->  
    apply k (lookup e x)  
  | Lam t ->  
    apply k (Clo (t, e))  
  | App (t1, t2) ->  
    evalkd e t1 (AppL { e; t2; k })
```

To evaluate t_1 t_2 , the interpreter **pushes** information on the stack, then **jumps** straight to evaluating t_1 .

A defunctionalized CPS interpreter

Example

Formalization

Remarks

apply interprets continuations as functions of values to values:

```
and apply (k : kont) (cv : cvalue) : cvalue =  
  match k with  
  | AppL { e; t2; k } ->  
    let cv1 = cv in  
    evalkd e t2 (AppR { cv1; k })  
  | AppR { cv1; k } ->  
    let cv2 = cv in  
    let Clo (u1, e') = cv1 in  
    evalkd (cv2 :: e') u1 k  
  | Init ->  
    cv
```

It pops the top stack frame and decides what to do, based on it.

A defunctionalized CPS interpreter

[Example](#)[Formalization](#)[Remarks](#)

To run the interpreter, start it with the [identity](#) continuation:

```
let eval e t =  
  evalkd e t Init
```


An abstract machine

We have reached an **abstract machine**, a simple **iterative** interpreter which maintains a few data structures:

- a **code** pointer: the term t ,
- an **environment** e ,
- a stack, or **continuation** k .

In fact, we have mechanically rediscovered the **CEK** machine.

Felleisen and Friedman,
Control operators, the SECD machine, and the λ -calculus, 1987.

Sig Ager, Biernacki, Danvy and Midtgaard,
**A Functional Correspondence between Evaluators
and Abstract Machines**, 2003.

Re-discovering other abstract machines

Example

Formalization

Remarks

Exercise: start with a **call-by-name** interpreter and follow an analogous process to rediscover Krivine's machine.

The solution is in **EvalCBNCPS**.

There once was a man named Krivine

Who invented a wond'rous machine.

It pushed and it popped

On abstractions it stopped;

That lean mean machine from Krivine.

— **Mitchell Wand**

Krivine, **A call-by-name lambda-calculus machine**, (1985) 2007.

1 Example: from a direct-style interpreter down to an abstract machine

2 Formalization

3 Remarks

Formulations of the CPS transformation

[Example](#)[Formalization](#)[Remarks](#)

There are [many](#) variants of the CPS transformation,
and sometimes [many](#) formulations of a single variant.

Let us look at the simplest formulation: Fischer and Plotkin's.

Fischer, [Lambda-Calculus Schemata](#), (1972) 1993.

Plotkin, [Call-by-name, call-by-value and the \$\lambda\$ -calculus](#), 1975.

Definition of the CBV CPS transformation

[Example](#)[Formalization](#)[Remarks](#)

A term is translated to a **function** of a continuation k to an answer.

$$\llbracket x \rrbracket =$$

Definition of the CBV CPS transformation

[Example](#)[Formalization](#)[Remarks](#)

A term is translated to a **function** of a continuation k to an answer.

$$\llbracket x \rrbracket = \lambda k.$$

Definition of the CBV CPS transformation

Example

Formalization

Remarks

A term is translated to a **function** of a continuation k to an answer.

$$\llbracket x \rrbracket = \lambda k. k \ x$$

$$\llbracket \lambda x. t \rrbracket =$$

Definition of the CBV CPS transformation

Example

Formalization

Remarks

A term is translated to a **function** of a continuation k to an answer.

$$\llbracket x \rrbracket = \lambda k. k \ x$$

$$\llbracket \lambda x. t \rrbracket = \lambda k.$$

Definition of the CBV CPS transformation

Example

Formalization

Remarks

A term is translated to a **function** of a continuation k to an answer.

$$\llbracket x \rrbracket = \lambda k. k \ x$$

$$\llbracket \lambda x. t \rrbracket = \lambda k. k \ (\lambda x. \llbracket t \rrbracket)$$

$$\llbracket t_1 \ t_2 \rrbracket = \lambda k.$$

Definition of the CBV CPS transformation

Example

Formalization

Remarks

A term is translated to a **function** of a continuation k to an answer.

$$\llbracket x \rrbracket = \lambda k. k \ x$$

$$\llbracket \lambda x. t \rrbracket = \lambda k. k \ (\lambda x. \llbracket t \rrbracket)$$

$$\llbracket t_1 \ t_2 \rrbracket = \lambda k. \llbracket t_1 \rrbracket$$

Definition of the CBV CPS transformation

Example

Formalization

Remarks

A term is translated to a **function** of a continuation k to an answer.

$$\llbracket x \rrbracket = \lambda k. k \ x$$

$$\llbracket \lambda x. t \rrbracket = \lambda k. k \ (\lambda x. \llbracket t \rrbracket)$$

$$\llbracket t_1 \ t_2 \rrbracket = \lambda k. \llbracket t_1 \rrbracket \ (\lambda x_1.$$

Definition of the CBV CPS transformation

Example

Formalization

Remarks

A term is translated to a **function** of a continuation k to an answer.

$$\llbracket x \rrbracket = \lambda k. k \ x$$

$$\llbracket \lambda x. t \rrbracket = \lambda k. k \ (\lambda x. \llbracket t \rrbracket)$$

$$\llbracket t_1 \ t_2 \rrbracket = \lambda k. \llbracket t_1 \rrbracket \ (\lambda x_1. \llbracket t_2 \rrbracket \ (\lambda x_2.$$

Definition of the CBV CPS transformation

Example

Formalization

Remarks

A term is translated to a **function** of a continuation k to an answer.

$$\llbracket x \rrbracket = \lambda k. k \ x$$

$$\llbracket \lambda x. t \rrbracket = \lambda k. k \ (\lambda x. \llbracket t \rrbracket)$$

$$\llbracket t_1 \ t_2 \rrbracket = \lambda k. \llbracket t_1 \rrbracket \ (\lambda x_1. \llbracket t_2 \rrbracket \ (\lambda x_2. x_1 \ x_2 \ k))$$

$$\llbracket \text{let } x = t_1 \text{ in } t_2 \rrbracket = \lambda k. \llbracket t_1 \rrbracket \ (\lambda x. \llbracket t_2 \rrbracket \ k)$$

A **value** $\lambda x. t$ is translated to a function of **two** arguments $\lambda x. \lambda k. \dots$

Definition of the CBV CPS transformation

Example

Formalization

Remarks

One avoids some redundancy by defining **two mutually recursive functions**, namely the translation of values $\llbracket v \rrbracket$:

$$\llbracket x \rrbracket = x$$

$$\llbracket \lambda x. t \rrbracket = \lambda x. \llbracket t \rrbracket$$

and the translation of terms $\llbracket t \rrbracket$:

$$\llbracket v \rrbracket = \lambda k. k \llbracket v \rrbracket$$

$$\llbracket t_1 t_2 \rrbracket = \lambda k. \llbracket t_1 \rrbracket (\lambda x_1. \llbracket t_2 \rrbracket (\lambda x_2. x_1 x_2 k))$$

$$\llbracket \text{let } x = t_1 \text{ in } t_2 \rrbracket = \lambda k. \llbracket t_1 \rrbracket (\lambda x. \llbracket t_2 \rrbracket k)$$

Indifference



In a transformed term, **the right-hand side of every application** is a **value**.

Therefore, its execution is **indifferent** to the choice of a call-by-name or call-by-value evaluation strategy.

In other words, **evaluation order** is fully **explicit** in a transformed term.

The transformation on the previous slide fixes a call-by-value strategy: it is the **CBV CPS transformation**.

It can serve as an **encoding** of call-by-value into call-by-name, thus answering a question raised in week 1.

Exercise (recommended): Define the CBN CPS transformation.

Stacklessness



Example

Formalization

Remarks

In a transformed term, **every call is a tail call**.

Therefore, reduction under a context is not required.

That is, execution **does not require a stack**.

We could (but won't) give a (small-step, substitution-based) semantics that takes **indifference** and **stacklessness** into account.

Exercise: Propose such a semantics. Prove that, when executing a CPS-transformed term, it is equivalent to the standard semantics.

Effect of the transformation of types

Example

Formalization

Remarks

How are **types** transformed?

A **value** of type T is translated to a value of type $\llbracket T \rrbracket$.

A **computation** of type T is translated to a value of type $\llbracket T \rrbracket$.

$$\llbracket \alpha \rrbracket = \alpha$$

$$\llbracket T_1 \rightarrow T_2 \rrbracket =$$

Effect of the transformation of types

Example

Formalization

Remarks

How are **types** transformed?

A **value** of type T is translated to a value of type $\langle T \rangle$.

A **computation** of type T is translated to a value of type $\llbracket T \rrbracket$.

$$\langle \alpha \rangle = \alpha$$

$$\langle T_1 \rightarrow T_2 \rangle = \langle T_1 \rangle \rightarrow$$

Effect of the transformation of types

Example

Formalization

Remarks

How are **types** transformed?

A **value** of type T is translated to a value of type $\langle T \rangle$.

A **computation** of type T is translated to a value of type $\llbracket T \rrbracket$.

$$\langle \alpha \rangle = \alpha$$

$$\langle T_1 \rightarrow T_2 \rangle = \langle T_1 \rangle \rightarrow \langle T_2 \rangle$$

$$\llbracket T \rrbracket =$$

Effect of the transformation of types

Example

Formalization

Remarks

How are **types** transformed?

A **value** of type T is translated to a value of type $\langle T \rangle$.

A **computation** of type T is translated to a value of type $\llbracket T \rrbracket$.

$$\langle \alpha \rangle = \alpha$$

$$\langle T_1 \rightarrow T_2 \rangle = \langle T_1 \rangle \rightarrow \llbracket T_2 \rrbracket$$

$$\llbracket T \rrbracket = (\langle T \rangle \rightarrow A) \rightarrow A$$

The type A , known as the **answer** type, is arbitrary and fixed.

One may take A to be the **empty type** 0 . Then, $\llbracket T \rrbracket$ is $\neg\neg\langle T \rangle$. The CPS transformation is known in logic as the **double-negation translation**.

Exercise (recommended): state and prove Type Preservation.

Effect of the transformation of types –
refined

Example

Formalization

Remarks

Could the transformation of types be made **more precise** in some sense?

$$\llbracket T \rrbracket = (\llbracket T \rrbracket \rightarrow A) \rightarrow A$$

Effect of the transformation of types –
refined

Example

Formalization

Remarks

Could the transformation of types be made **more precise** in some sense?

$$\llbracket T \rrbracket = (\llbracket T \rrbracket \rightarrow A) \rightarrow A$$

Every transformed term is in fact **answer-type polymorphic**:

$$\llbracket T \rrbracket = \forall A. (\llbracket T \rrbracket \rightarrow A) \rightarrow A$$

Furthermore,

Effect of the transformation of types –
refined

Example

Formalization

Remarks

Could the transformation of types be made **more precise** in some sense?

$$\llbracket T \rrbracket = (\llbracket T \rrbracket \rightarrow A) \rightarrow A$$

Every transformed term is in fact **answer-type polymorphic**:

$$\llbracket T \rrbracket = \forall A. (\llbracket T \rrbracket \rightarrow A) \rightarrow A$$

Furthermore, every transformed term invokes its continuation **once**:

$$\llbracket T \rrbracket = \forall A. (\llbracket T \rrbracket \rightarrow A) \multimap A$$

However, these properties are violated in the presence of **control effects**.

Thielecke, **From control effects to typed continuation passing**, 2003.

Semantic preservation

Example

Formalization

Remarks

Plotkin (1975) proved semantic preservation,
based on a [small-step simulation diagram](#).

This proof is complicated by the presence of administrative reductions.

A simpler approach is to use big-step semantics in the hypothesis:

Lemma (Semantic Preservation)

If $t \Downarrow_{cbv} v$ and if w is a value, then $\llbracket t \rrbracket w \longrightarrow_{cbv}^ w \langle v \rangle$.*

One should prove, in addition, that divergence is preserved.

[Exercise](#) (recommended): Prove this lemma.

- 1 Example: from a direct-style interpreter down to an abstract machine
- 2 Formalization
- 3 Remarks

Monadic intermediate form

Example

Formalization

Remarks

If one just aims to make evaluation order explicit, CPS is **overkill**.

This transformation, too, achieves **indifference**:

$$\begin{aligned}
 \llbracket x \rrbracket &= x \\
 \llbracket \lambda x. t \rrbracket &= \lambda x. \llbracket t \rrbracket \\
 \llbracket t_1 \ t_2 \rrbracket &= \text{let } x_1 = \llbracket t_1 \rrbracket \text{ in} \\
 &\quad \text{let } x_2 = \llbracket t_2 \rrbracket \text{ in} \\
 &\quad x_1 \ x_2 \\
 \llbracket \text{let } x = t_1 \text{ in } t_2 \rrbracket &= \text{let } x = \llbracket t_1 \rrbracket \text{ in } \llbracket t_2 \rrbracket
 \end{aligned}$$

In a transformed term, **the components of every application are values**.

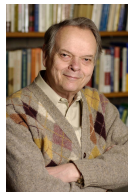
By further hoisting “*let*” out of the left-hand side of “*let*”, one gets **administrative normal form**.

Flanagan, Sabry, Felleisen, **The essence of compiling with continuations**, 1993 (2003).

The CPS monad

The CPS transformation is a special case of the [monadic transformation](#).
See Dagand's lectures!

Some history



Continuations, and the CPS transformation, were independently discovered by many researchers during the 1960s.

John C. Reynolds, *The discoveries of continuations*, 1993.

Some history

The CPS transformation has been used in compilers.

Rabbit (Steele). SML/NJ.

Appel, *Compiling with Continuations*, 1992.

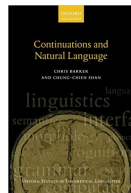
Today, heap-allocating the stack is considered *too costly*:

- bad locality;
- increased GC load;
- confuses the processor's built-in prediction of return addresses.

Yet, *selective* CPS transformations are used to compile effect handlers, and some compilers use CPS as an *intermediate form* before coming back to direct style.

Kennedy, *Compiling with continuations, continued*, 2007.

Some history



Can λ -calculus and continuations explain the structure of speech?

Chris Barker,
Continuations and the nature of quantification, 2002.

Chris Barker and Chung-Chieh Shan,
Continuations and Natural Language, 2014.

A few things to remember

Continuations rule!

- The CPS transformation achieves several remarkable effects:
 - making **the stack** explicit;
 - making **evaluation order** explicit;
 - suggesting/explaining **control operators**.
- It plays a **fundamental role** in prog. language theory and in logic.
- Continuation-passing is also a useful **programming technique**.