

MPRI 2.4  
CPS

François  
Pottier

Examples

Interpreter  
Traversal

Formulations

Soundness

Remarks

Appendix

# Making the stack explicit: the continuation-passing style transformation

MPRI 2.4

François Pottier



2018

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  $\lambda$ -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  $\lambda$ -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 Examples

From a direct-style interpreter down to an abstract machine

From recursive traversal down to iterative traversal with link inversion

## 2 Formulations

## 3 Soundness

## 4 Remarks

## 5 Madness in small steps

## 1 Examples

From a direct-style interpreter down to an abstract machine

From recursive traversal down to iterative traversal with link inversion

## 2 Formulations

## 3 Soundness

## 4 Remarks

## 5 Madness in small steps

## A direct-style interpreter

Recall our environment-based interpreter for call-by-value  $\lambda$ -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

[Examples](#)[Interpreter](#)[Traversal](#)[Formulations](#)[Soundness](#)[Remarks](#)[Appendix](#)

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

[Examples](#)[Interpreter](#)[Traversal](#)[Formulations](#)[Soundness](#)[Remarks](#)[Appendix](#)

```
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

[Examples](#)[Interpreter](#)[Traversal](#)[Formulations](#)[Soundness](#)[Remarks](#)[Appendix](#)

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

Examples

Interpreter

Traversal

Formulations

Soundness

Remarks

Appendix

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

Examples

Interpreter

Traversal

Formulations

Soundness

Remarks

Appendix

What is special about this interpreter?

## Properties of the interpreter

Examples

Interpreter

Traversal

Formulations

Soundness

Remarks

Appendix

What is special about this interpreter?

- Every call of evalk to itself is a tail call.
- Every call of evalk to a continuation is a tail call.

## Tail calls

[Examples](#)[Interpreter](#)[Traversal](#)[Formulations](#)[Soundness](#)[Remarks](#)[Appendix](#)

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

 $| \quad v$  $| \quad nt\ nt$ 

– a tail call

 $| \quad \text{let } nt \text{ in } tt$  $| \quad \text{if } nt \text{ then } tt \text{ else } tt$  $nt ::=$ 

terms not in tail position

 $| \quad v$  $| \quad nt\ nt$ 

– an ordinary call

 $| \quad \text{let } nt \text{ in } nt$  $| \quad \text{if } nt \text{ then } nt \text{ else } nt$

## Verified tail calls

[Examples](#)[Interpreter](#)[Traversal](#)[Formulations](#)[Soundness](#)[Remarks](#)[Appendix](#)

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

Examples

Interpreter

Traversal

Formulations

Soundness

Remarks

Appendix

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

Examples

Interpreter

Traversal

Formulations

Soundness

Remarks

Appendix

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

Examples

Interpreter

Traversal

Formulations

Soundness

Remarks

Appendix

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

Examples

Interpreter

Traversal

Formulations

Soundness

Remarks

Appendix

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

[Examples](#)[Interpreter](#)[Traversal](#)[Formulations](#)[Soundness](#)[Remarks](#)[Appendix](#)

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

[Examples](#)[Interpreter](#)[Traversal](#)[Formulations](#)[Soundness](#)[Remarks](#)[Appendix](#)

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](#):

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

# A defunctionalized CPS interpreter

[Examples](#)[Interpreter](#)[Traversal](#)[Formulations](#)[Soundness](#)[Remarks](#)[Appendix](#)

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

[Examples](#)[Interpreter](#)[Traversal](#)[Formulations](#)[Soundness](#)[Remarks](#)[Appendix](#)

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

[Examples](#)[Interpreter](#)[Traversal](#)[Formulations](#)[Soundness](#)[Remarks](#)[Appendix](#)

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  $\lambda$ -calculus, 1987.

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

# Re-discovering other abstract machines

[Examples](#)[Interpreter](#)[Traversal](#)[Formulations](#)[Soundness](#)[Remarks](#)[Appendix](#)

**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 Examples

From a direct-style interpreter down to an abstract machine

From recursive traversal down to iterative traversal with link inversion

## 2 Formulations

## 3 Soundness

## 4 Remarks

## 5 Madness in small steps

# A type of binary trees

[Examples](#)[Interpreter](#)[Traversal](#)[Formulations](#)[Soundness](#)[Remarks](#)[Appendix](#)

Consider a simple type of binary trees:

```
type tree =
| Leaf
| Node of { data: int; left: tree; right: tree }
```

## Direct-style traversal

Suppose we wish to perform a postfix tree traversal:

```
let rec walk (t : tree) : unit =
  match t with
  | Leaf ->
    ()
  | Node { data; left; right } ->
    walk left;
    walk right;
    printf "%d\n" data
```

This is **recursive code in direct style**.

Neither of the recursive calls is a tail call.

[Examples](#)[Interpreter](#)[Traversal](#)[Formulations](#)[Soundness](#)[Remarks](#)[Appendix](#)

Now suppose we wish to make the code [iterative](#). Swoop, CPS!

```
let rec walkk (t : tree) (k : unit -> 'a) : 'a =
  match t with
  | Leaf ->
    k()
  | Node { data; left; right } ->
    walkk left (fun () ->
      walkk right (fun () ->
        printf "%d\n" data;
        k())))

```

The traversal is initiated with an identity continuation:

```
let walk t =
  walkk t (fun t -> t)
```

# CPS traversal, defunctionalized

[Examples](#)[Interpreter](#)[Traversal](#)[Formulations](#)[Soundness](#)[Remarks](#)[Appendix](#)

Next, we might wish to make the stack an explicit **data structure**.

Swoop, defunctionalization!

The type of defunctionalized continuations:

```
type kont =
| Init
| GoneL of { data: int; tail: kont; right: tree }
| GoneR of { data: int; tail: kont }
```

# CPS traversal, defunctionalized

[Examples](#)[Interpreter](#)[Traversal](#)[Formulations](#)[Soundness](#)[Remarks](#)[Appendix](#)

The main function is a loop that walks down the leftmost branch while **pushing** information onto the stack:

```
let rec walkkd (t : tree) (k : kont) : unit =
  match t with
  | Leaf ->
    apply k ()
  | Node { data; left; right } ->
    walkkd left (GoneL { data; tail = k; right })
```

Think of the stack as **Ariadne's thread**.

## CPS traversal, defunctionalized

[Examples](#)[Interpreter](#)[Traversals](#)[Formulations](#)[Soundness](#)[Remarks](#)[Appendix](#)

The apply function comes back up out of a child.

```
and apply k () =
  match k with
  | Init ->
    ()
  | GoneL { data; tail; right } ->
    walkkd right (GoneR { data; tail })
  | GoneR { data; tail } ->
    printf "%d\n" data;
    apply tail ()
```

It **pops** information off the stack so as to decide what to do.

When coming out of a left child, go down into its right sibling.

When coming out of a right child, go further up.

And now, for something a little  
**UNEXPECTED and WILD.**

And now, for something a little  
**UNEXPECTED and WILD.**  
**A CRAZY HACK.**

# Recycling

When we allocate a `GoneR` continuation,  
we drop a `GoneL` continuation at the same time.

Indeed, here, continuations are linear. They are used exactly once.

```
| GoneL { data; tail; right } ->
  walkkd right (GoneR { data; tail })
```

This suggests that the memory block could be recycled (re-used).

[Examples](#)[Interpreter](#)[Traversal](#)[Formulations](#)[Soundness](#)[Remarks](#)[Appendix](#)

When we allocate a `GoneL` continuation,  
a `Node` goes temporarily unused at the same time.

This node won't be accessed until this `GoneL` frame  
first is changed to `GoneR` then is popped off the stack.

```
| Node { data; left; right } ->
    walkkd left (GoneL { data; tail = k; right })
```

This suggests that the memory block could be `recycled`, too,  
provided we `restore` it when we are done with it.

# A tree is a continuation is a tree

Examples

Interpreter

Traversal

Formulations

Soundness

Remarks

Appendix

In OCaml, the type of a memory block **cannot** be changed over time.

Thus, recycling tree nodes as stack frames, and vice-versa,  
requires **trees** and **continuations** to have **the same type**.

Uh?

# A tree is a continuation is a tree

[Examples](#)[Interpreter](#)[Traversal](#)[Formulations](#)[Soundness](#)[Remarks](#)[Appendix](#)

Could we **disguise** a continuation as a tree?

In other words, could a stack frame **fit** in a tree node?

```
type kont =
| Init
| GoneL of { data: int; tail: kont; right: tree }
| GoneR of { data: int; tail: kont }
```

```
type tree =
| Leaf
| Node of { data: int; left: tree; right: tree }
```

# A tree is a continuation is a tree

Could we **disguise** a continuation as a tree?

In other words, could a stack frame **fit** in a tree node?

```
type kont =
| Init
| Gonel of { data: int; tail: kont; right: tree }
| Goner of { data: int; tail: kont }
```

```
type tree =
| Leaf
| Node of { data: int; left: tree; right: tree }
```

Yes, kind of.

We just need **one extra bit** of storage per tree node,  
so as to distinguish **Gonel** and **Goner**.

# A tree is a continuation is a tree

Examples

Interpreter

Traversal

Formulations

Soundness

Remarks

Appendix

Add one “status” bit per tree node. Make nodes mutable.

```
type status = GoneL | GoneR
type mtree = Leaf | Node of {
    data: int;           mutable status: status;
    mutable left: mtree; mutable right: mtree
}
type mkont = mtree
```

Tree records and continuation records occupy **the same space** in memory.

Thus, a tree record can be turned into a continuation record, and back!

By convention, in a “tree” record, the `status` field is `GoneL`.

In a “continuation” record,

- either `status` is `GoneL` and the `left` field stores tail;
- or `status` is `GoneR` and the `right` field stores tail.

# CPS traversal with link inversion

[Examples](#)[Interpreter](#)[Traversals](#)[Formulations](#)[Soundness](#)[Remarks](#)[Appendix](#)

Instead of allocating a `GoneL` continuation,  
we now **change** the tree record to a continuation record:

```
let rec walkkdi (t : mtree) (k : mkont) : unit =
  match t with
  | Leaf ->
    apply k t
  | Node ({ left; _ } as n) ->
    (* Change this tree to a [GoneL] continuation. *)
    assert (n.status = GoneL);
    n.left (* n.tail *) <- k;
    walkkdi left (t : mkont)
```

The `left` field is **overwritten**, which is scary! We must restore it later.

We find that, in every call to `walkkdi t k` and `apply k t`,  
`k` is the **parent** of `t` in the tree.

# CPS traversal with link inversion

The rest of the code, in its horrific glory:

```
and apply (k : mkont) (child : mtree) : unit =
  match k with
  | Leaf -> ()
  | Node ({ status = GoneL; left = tail; right; _ } as n) ->
    n.status <- GoneR;          (* update continuation! *)
    n.left <- child;          (* restore orig. left child! *)
    n.right (* n.tail *) <- tail;
    walkkdi right k
  | Node ({ data; status = GoneR; right = tail; _ } as n) ->
    printf "%d\n" data;
    n.status <- GoneL;          (* change back to a tree! *)
    n.right <- child;          (* restore orig. right child! *)
    apply tail (k : mtree)
```

This code runs in **constant space**. Look Ma, no stack! (Uh?)

## CPS traversal with link inversion

More accurately, the stack is stored [in the tree itself](#), by [reversing pointers](#).

This [hack](#) technique is known as [link inversion](#).

It was invented for use in garbage collectors, which must [traverse the heap](#) without requiring a huge stack.

We have re-discovered it via the idea of allocating continuations [in place](#).

Schorr and Waite, [An efficient machine-independent procedure for garbage collection in various list structures](#), 1967.

Hubert and Marché, [A case study of C source code verification: the Schorr-Waite algorithm](#), 2005.

Sobel and Friedman, [Recycling continuations](#), 1998.

# CPS traversal with link inversion

[Examples](#)[Interpreter](#)[Traversal](#)[Formulations](#)[Soundness](#)[Remarks](#)[Appendix](#)

“Kids, do not try this at home”: this idea is [complicated](#) and [expensive](#).

(The OCaml GC imposes a [write barrier](#): write operations are slow.)

**Exercise:** Extend the code to deal with [graphs](#), where there can be [sharing](#) and [cycles](#). (Use a [mark](#) bit in every node.)

## 1 Examples

From a direct-style interpreter down to an abstract machine

From recursive traversal down to iterative traversal with link inversion

## 2 Formulations

### 3 Soundness

### 4 Remarks

### 5 Madness in small steps

# Formulations of the CPS transformation

Examples

Interpreter

Traversal

Formulations

Soundness

Remarks

Appendix

There are **many** variants of the CPS transformation,  
and sometimes **many** formulations of a single variant.

Let us begin with 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

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

$$\llbracket x \rrbracket =$$

# Definition of the CBV CPS transformation

Examples

Interpreter

Traversal

Formulations

Soundness

Remarks

Appendix

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

Examples

Interpreter

Traversal

Formulations

Soundness

Remarks

Appendix

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

Examples

Interpreter

Traversal

Formulations

Soundness

Remarks

Appendix

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

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

Examples

Interpreter

Traversal

Formulations

Soundness

Remarks

Appendix

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

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

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

Examples

Interpreter

Traversal

Formulations

Soundness

Remarks

Appendix

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 function  $\lambda x. t$  is translated to a function of two arguments  $\lambda x. \lambda k..$

## Definition of the CBV CPS transformation

One avoids some redundancy by defining two mutually recursive functions, namely the translation of values  $(v)$ :

$$(x) = x$$

$$(\lambda x. t) = \lambda x. [t]$$

and the translation of terms  $[t]$ :

$$[v] = \lambda k. k (v)$$

$$[t_1 \ t_2] = \lambda k. [t_1] (\lambda x_1. [t_2] (\lambda x_2. x_1 \ x_2 \ k))$$

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

## 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



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

Examples

Interpreter

Traversal

Formulations

Soundness

Remarks

Appendix

How are **types** transformed?

A **value** of type  $T$  is translated to a value of type  $(T)$ .

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

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

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

# Effect of the transformation of types

Examples

Interpreter

Traversal

Formulations

Soundness

Remarks

Appendix

How are **types** transformed?

A **value** of type  $T$  is translated to a value of type  $(T)$ .

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

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

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

# Effect of the transformation of types

Examples

Interpreter

Traversal

Formulations

Soundness

Remarks

Appendix

How are **types** transformed?

A **value** of type  $T$  is translated to a value of type  $(T)$ .

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

$$(\alpha) = \alpha$$

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

$$\llbracket T \rrbracket =$$

# Effect of the transformation of types

Examples

Interpreter

Traversal

Formulations

Soundness

Remarks

Appendix

How are **types** transformed?

A **value** of type  $T$  is translated to a value of type  $(T)$ .

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

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

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**:

$$\llbracket T \rrbracket = \textcolor{red}{\forall A.} ((\llbracket T \rrbracket) \rightarrow A) \rightarrow A$$

Furthermore,

## Effect of the transformation of types – refined

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.

## Administrative redexes

[Examples](#)[Interpreter](#)[Traversal](#)[Formulations](#)[Soundness](#)[Remarks](#)[Appendix](#)

The translation presented so far is naïve.

It produces many “administrative”  $\beta$ -redexes.

E.g., in an application of a variable to a variable:

$$\begin{aligned} [[f\ x]] &= \lambda k. [[f]] (\lambda x_1. [[x]] (\lambda x_2. x_1\ x_2\ k)) \\ &= \lambda k. (\lambda k. k\ (f)) (\lambda x_1. (\lambda k. k\ (x))) (\lambda x_2. x_1\ x_2\ k) \\ &= \lambda k. (\lambda k. k\ f) (\lambda x_1. (\lambda k. k\ x) (\lambda x_2. x_1\ x_2\ k)) \\ &\stackrel{\beta}{=} \lambda k. (\lambda x_1. (\lambda k. k\ x) (\lambda x_2. x_1\ x_2\ k))\ f \\ &\stackrel{\beta}{=} \lambda k. (\lambda k. k\ x) (\lambda x_2. f\ x_2\ k) \\ &\stackrel{\beta}{=} \lambda k. (\lambda x_2. f\ x_2\ k)\ x \\ &\stackrel{\beta}{=} \lambda k. f\ x\ k \end{aligned}$$

This is inefficient: one function call is translated to five function calls!

# Semantic preservation

Examples

Interpreter

Traversal

Formulations

Soundness

Remarks

Appendix

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  $\llbracket t \rrbracket w \xrightarrow{^*_{cbv}} w(v)$ .

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

Exercise (recommended): Prove this lemma.

## Ways of eliminating administrative redexes

Administrative redexes can be reduced [after](#) the CPS transformation.

- During the translation, mark each  $\lambda$  that corresponds to a source  $\lambda$ .
- After the translation, reduce every redex whose  $\lambda$  is unmarked.

Another idea is to reduce all “[no-brainer](#)” redexes. They include the admin. redexes and are size-decreasing. This can be done on the fly.

Davis, Meehan, Shivers, [No-brainer CPS conversion](#), 2017.

Yet another approach is to define a “[one-pass](#)” CPS transformation that does not produce any administrative redexes in the first place...

## Towards a one-pass transformation

[Examples](#)[Interpreter](#)[Traversal](#)[Formulations](#)[Soundness](#)[Remarks](#)[Appendix](#)

The first step is to make some of the abstractions and applications **static**.

They should take place at **transformation time**, not at **runtime**.

Instead of viewing  $\llbracket t \rrbracket = \lambda k. \dots$  as a function of a term to a term,  
let us view  $\llbracket t \rrbracket \{ w \} = \dots$  as a function of a term and a value to a term.

$$\llbracket x \rrbracket = x$$

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

$$\llbracket v \rrbracket \{ w \} = w \llbracket v \rrbracket$$

$$\llbracket t_1 \ t_2 \rrbracket \{ w \} = \llbracket t_1 \rrbracket \{ \lambda x_1. \llbracket t_2 \rrbracket \{ \lambda x_2. \ x_1 \ x_2 \ w \} \}$$

$$\llbracket \text{let } x = t_1 \text{ in } t_2 \rrbracket \{ w \} = \llbracket t_1 \rrbracket \{ \lambda x. \llbracket t_2 \rrbracket \{ w \} \}$$

$k$  denotes a **variable**;  $w$  denotes a **value**.

## Towards a one-pass transformation

[Examples](#)[Interpreter](#)[Traversal](#)[Formulations](#)[Soundness](#)[Remarks](#)[Appendix](#)

This transformation produces [fewer administrative redexes](#):

$$\begin{aligned} \llbracket f x \rrbracket \{ k \} &= \llbracket f \rrbracket \{ \lambda x_1. \llbracket x \rrbracket \{ \lambda x_2. x_1 x_2 k \} \} \\ &= (\lambda x_1. (\lambda x_2. x_1 x_2 k) x) f \\ &\stackrel{=\beta}{=} (\lambda x_2. f x_2 k) x \\ &\stackrel{=\beta}{=} f x k \end{aligned}$$

The remaining administrative redexes arise from the equation

$$\llbracket v \rrbracket \{ w \} = w \langle v \rangle$$

in the case where the continuation  $w$  is a  $\lambda$ -abstraction.

How could we alter this equation?

## Towards a one-pass transformation

Examples

Interpreter

Traversal

Formulations

Soundness

Remarks

Appendix

Define the **smart application** of a (continuation) value  $w$  to a value  $v$ :

$$\begin{aligned} x @_{\beta} v &= x v \\ (\lambda x.t) @_{\beta} v &= t[v/x] \end{aligned}$$

Note:

- A continuation  $w$  is always either a variable or a “transformation”  $\lambda$ , never a “source”  $\lambda$ , so the redex reduced by  $w @_{\beta} v$  is **administrative**.
- Provided every “transformation”  $\lambda$  uses its argument **linearly**,  $w @_{\beta} (v)$  does not duplicate ( $v$ ), so transformed terms remain **linear** in size.

## A one-pass transformation

Change the translation of values. Make every “transformation”  $\lambda$  linear.

$$\langle\!\langle x\rangle\!\rangle = x$$

$$\langle\!\langle \lambda x. t \rangle\!\rangle = \lambda x. \lambda k. \langle\!\langle t \rangle\!\rangle \{ k \}$$

$$\langle\!\langle v \rangle\!\rangle \{ w \} = w @_{\beta} \langle\!\langle v \rangle\!\rangle$$

$$\langle\!\langle t_1 \; t_2 \rangle\!\rangle \{ w \} = \langle\!\langle t_1 \rangle\!\rangle \{ \lambda x_1. \langle\!\langle t_2 \rangle\!\rangle \{ \lambda x_2. x_1 \; x_2 \; w \} \}$$

$$\langle\!\langle \text{let } x = t_1 \text{ in } t_2 \rangle\!\rangle \{ w \} = \langle\!\langle t_1 \rangle\!\rangle \{ \lambda x. \text{let } x = x \text{ in } \langle\!\langle t_2 \rangle\!\rangle \{ w \} \}$$

This transformation produces no administrative redexes.

Dargaye and Leroy, Mechanized Verification  
of CPS Transformations, 2007.

# A one-pass transformation

Look Ma, no administrative redexes!

$$\begin{aligned} \llbracket f x \rrbracket \{ k \} &= \llbracket f \rrbracket \{ \lambda x_1. \llbracket x \rrbracket \{ \lambda x_2. x_1 x_2 k \} \} \\ &= (\lambda x_1. (\lambda x_2. x_1 x_2 k) @_\beta x) @_\beta f \\ &= (\lambda x_2. f x_2 k) @_\beta x \\ &= f x k \end{aligned}$$

## A one-pass transformation

Look Ma, no administrative redexes!

$$\begin{aligned} \llbracket f x \rrbracket \{ k \} &= \llbracket f \rrbracket \{ \lambda x_1. \llbracket x \rrbracket \{ \lambda x_2. x_1 x_2 k \} \} \\ &= (\lambda x_1. (\lambda x_2. x_1 x_2 k) @_\beta x) @_\beta f \\ &= (\lambda x_2. f x_2 k) @_\beta x \\ &= f x k \end{aligned}$$

A drawback of Dargaye and Leroy's approach is that  $\cdot @_\beta \cdot$  does not commute with substitutions, which causes a difficulty in the proof of semantic preservation.

This is repaired in the formulation shown next...



# Higher-order versus first-order formulations

Danvy and Filinski (1992) first defined this one-pass transformation.

Their formulation was in a “higher-order” style.

Let me give a simpler, “first-order” presentation of their transformation.

Danvy and Filinski, *Representing control: a study of the CPS transformation*, 1992.

Pottier, *Revisiting the CPS transformation and its implementation*, 2017.

## A first-order one-pass CPS transformation

Let a continuation  $c$  be either a value  $w$  or a “transformation”  $\lambda$ :

$$c ::= w \mid mx.t$$

In  $mx.t$ , the term  $t$  must have exactly one occurrence of  $x$ .

Define **continuation application**  $apply\ c\ v$  and **reification**  $reify\ c$ :

$$apply\ w\ v = w\ v \quad \text{-- an object-level application}$$

$$apply\ (mx.t)\ v = t[v/x] \quad \text{-- a meta-level substitution}$$

$$reify\ w = w \quad \text{-- a no-op}$$

$$reify\ (mx.t) = \lambda x.t$$

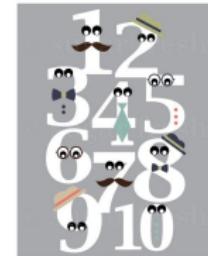
# A first-order one-pass CPS transformation

Danvy and Filinski's transformation can then be presented as follows:

$$\begin{aligned}\langle\!\langle x\rangle\!\rangle &= x \\ \langle\!\langle \lambda x.t \rangle\!\rangle &= \lambda x. \lambda k. \langle\!\langle t \rangle\!\rangle \{ k \} \\ \langle\!\langle v \rangle\!\rangle \{ c \} &= \text{apply } c \langle\!\langle v \rangle\!\rangle \\ \langle\!\langle t_1 \ t_2 \rangle\!\rangle \{ c \} &= \langle\!\langle t_1 \rangle\!\rangle \{ \text{mx}_1. \langle\!\langle t_2 \rangle\!\rangle \{ \text{mx}_2. x_1 \ x_2 \ (\text{reify } c) \} \} \\ \langle\!\langle \text{let } x = t_1 \text{ in } t_2 \rangle\!\rangle \{ c \} &= \langle\!\langle t_1 \rangle\!\rangle \{ \text{mx}_1. \text{let } x = x_1 \text{ in } \langle\!\langle t_2 \rangle\!\rangle \{ c \} \}\end{aligned}$$

It is close to Dargaye and Leroy's formulation, yet is better behaved: as we will see, it commutes with substitution.

## Now, in de Bruijn style



Let us use **o** and **m** as explicit injections:

$$c ::= o\ w \mid m\ t$$

**m**, like  $\lambda$ , is considered a binder.

Continuation application, reification, and substitution  $c[\sigma]$  are as follows:

$$\begin{array}{lll} \text{apply } (o\ w)\ v = w\ v & \text{reify } (o\ w) = w & (o\ w)[\sigma] = o\ (w[\sigma]) \\ \text{apply } (m\ t)\ v = t[v/] & \text{reify } (m\ t) = \lambda t & (m\ t)[\sigma] = m\ (t[\uparrow\sigma]) \end{array}$$

See [CPSDefinition](#).

# The CPS transformation in de Bruijn style

The transformation is formulated in de Bruijn style as follows:

$$\begin{aligned} (\!(x)\!) &= x \\ (\!(\lambda t)\!) &= \lambda \lambda (\![\uparrow^1 t]\!) \{ \textcolor{red}{0} \ 0 \} \end{aligned}$$

$$\begin{aligned} \llbracket v \rrbracket \{ c \} &= \text{apply } c \ (\!(v)\!) \\ \llbracket t_1 \ t_2 \rrbracket \{ c \} &= \llbracket t_1 \rrbracket \{ \textcolor{red}{m} \ \llbracket \uparrow^1 t_2 \rrbracket \{ \textcolor{red}{m} \ 1 \ 0 \ \uparrow^2 (reify \ c) \} \} \\ \llbracket \text{let } t_1 \text{ in } t_2 \rrbracket \{ c \} &= \llbracket t_1 \rrbracket \{ \textcolor{red}{m} \ \text{let } 0 \text{ in } \llbracket \uparrow^1 t_2 \rrbracket \{ \uparrow^2 c \} \} \end{aligned}$$

$\uparrow^i t$  is short for  $t[+i]$ .  $\uparrow_1^1 t$  is short for  $t[\uparrow (+1)]$ .

$\uparrow^1$  means end-of-scope for variable 0.

# The CPS transformation in de Bruijn style

[Examples](#)[Interpreter](#)[Traversal](#)[Formulations](#)[Soundness](#)[Remarks](#)[Appendix](#)

The transformation is formulated in de Bruijn style as follows:

$$\begin{aligned} (\!(x)\!) &= x \\ (\!(\lambda t)\!) &= \lambda \lambda (\![\uparrow^1 t]\!) \{ \textcolor{red}{0} \ 0 \} \end{aligned}$$

$$\begin{aligned} \llbracket v \rrbracket \{ c \} &= \text{apply } c \ (v) \\ \llbracket t_1 \ t_2 \rrbracket \{ c \} &= \llbracket t_1 \rrbracket \{ \textcolor{red}{m} \ \llbracket \uparrow^1 t_2 \rrbracket \{ m \ 1 \ 0 \ \uparrow^2 (reify \ c) \} \} \\ \llbracket \text{let } t_1 \text{ in } t_2 \rrbracket \{ c \} &= \llbracket t_1 \rrbracket \{ \textcolor{red}{m} \ \text{let } 0 \text{ in } \llbracket \uparrow^1 t_2 \rrbracket \{ \uparrow^2 c \} \} \end{aligned}$$

$\uparrow^i t$  is short for  $t[+i]$ .  $\uparrow_1^1 t$  is short for  $t[\uparrow (+1)]$ .

$\uparrow^1$  means [end-of-scope](#) for variable 0.

$\uparrow^2$  means end-of-scope for variables 0 and 1.

$\uparrow_1^1$  means end-of-scope for variable 1.



Worse, Coq does not like this definition because the recursive calls concern [renamed](#) subterms! Well-founded recursion on [size](#) is required.

See [CPSDefinition](#).

## 1 Examples

From a direct-style interpreter down to an abstract machine

From recursive traversal down to iterative traversal with link inversion

## 2 Formulations

## 3 Soundness

## 4 Remarks

## 5 Madness in small steps

# Semantic Preservation

[Examples](#)[Interpreter](#)[Traversal](#)[Formulations](#)[Soundness](#)[Remarks](#)[Appendix](#)

We would like to prove this:

## Lemma (Semantic Preservation)

If  $t \downarrow_{cbv} v$ , then  $\llbracket t \rrbracket \{ m\ 0 \} \downarrow_{cbv} (v)$ .

$m\ 0$  is the **identity continuation**: in nominal style,  $m\ x.x$ .

For an inductive proof, the statement must be generalized, as follows...

# Semantic Preservation

Examples

Interpreter

Traversal

Formulations

Soundness

Remarks

Appendix

Define  $t \lesssim u$  as follows: for every value  $v$ ,  $u \downarrow_{\text{cbv}} v$  implies  $t \downarrow_{\text{cbv}} v$ .

## Lemma (Big-step Simulation)

Suppose  $\text{reify } c$  is a value. If  $t \downarrow_{\text{cbv}} v$ , then  $\llbracket t \rrbracket \{ c \} \lesssim \text{apply } c \ (v)$ .

Compare with our earlier claim concerning Plotkin's CPS transformation.

The proof is in [CPSCorrectnessBigStep](#).

Exercise: Replay the proof in Coq. Then erase it and redo it from scratch.

Exercise: Write a clear paper or L<sup>A</sup>T<sub>E</sub>X proof and send it to me!

The proof requires two key lemmas, shown next...

# Key Lemma 1: Substitution

[Examples](#)[Interpreter](#)[Traversal](#)[Formulations](#)[Soundness](#)[Remarks](#)[Appendix](#)

## Lemma (Substitution)

Let  $\sigma$  and  $\sigma'$  be value substitutions such that  $\sigma'$  is equal to  $\sigma ; \{ \cdot \}$ . Then,

$$(\llbracket t \rrbracket \{ c \})[\sigma'] = \llbracket t[\sigma] \rrbracket \{ c[\sigma'] \}.$$

## Lemma (Substitution—a special case)

Let  $v$  and  $w$  be values. Then,

$$(\llbracket t \rrbracket \{ \uparrow^2 c \ })[(\llbracket v \rrbracket \cdot (\llbracket w \rrbracket \cdot id)] = \llbracket t[v \cdot w \cdot id] \rrbracket \{ c \}.$$

In nominal style: if  $x, y \notin fv(c)$ , then

$$(\llbracket t \rrbracket \{ c \ })[(\llbracket v \rrbracket / x, (\llbracket w \rrbracket / y)] = \llbracket t[v/x, w/y] \rrbracket \{ c \}.$$

We push a substitution into the term, leaving the continuation untouched.  
A target language substitution becomes a source language substitution.

See [CPSSubstitution](#).

## Key Lemma 2: Kubstitution

Examples

Interpreter

Traversal

Formulations

Soundness

Remarks

Appendix

### Lemma (Kubstitution)

Let  $\theta$  and  $\sigma$  be substitutions such that  $\theta ; \sigma$  is id. Then,

$$\llbracket (t[\theta]) \{ c \} \rrbracket [\sigma] = \llbracket t \rrbracket \{ c[\sigma] \}.$$

### Lemma (Kubstitution—a special case)

For every value  $v$ ,  $(\llbracket \uparrow^1 t \rrbracket \{ c \})[v/] = \llbracket t \rrbracket \{ c[v/] \}$ .

In nominal style: if  $x \notin fv(t)$ , then  $(\llbracket t \rrbracket \{ c \})[v/x] = \llbracket t \rrbracket \{ c[v/x] \}$ .

We push a substitution into the continuation, leaving the term untouched.  
This is and remains a target language substitution.

See [CPSKubstitution](#).

Interlude: Enumerating  $\lambda$ -terms[Examples](#)[Interpreter](#)[Traversal](#)[Formulations](#)[Soundness](#)[Remarks](#)[Appendix](#)

Define the **size** of a term as follows: variables have size 0;  $\lambda$ -abstractions and applications contribute 1.

**Step 1:** In OCaml, implement an exhaustive **enumeration** of the  $\lambda$ -terms of size  $s$  and with at most  $n$  free variables. (Given as an exercise in week 1.)

```
(* Enumerate all variables between 0 and n excluded. *)
let var (n : int) (k : term -> unit) : unit = ...
(* Enumerate all manners of splitting an integer s. *)
let split (s : int) (k : int -> int -> unit) : unit = ...
(* Enumerate all terms of size s with at most n variables. *)
let term (s : int) (n : int) (k : term -> unit) : unit = ...
```

An enumerator is naturally written in CPS style!

## Interlude: Testing Semantic Preservation

**Step 2:** In OCaml, implement the CPS transformation.

```
type continuation =
| O of term
| M of term
let rec cps (t : term) (c : continuation) : term = ...
```

**Step 3:** In OCaml, implement a test for the relation  $\cdot \lesssim \cdot$ :

```
let sim (t1 : term) (t2 : term) : bool = ...
```

Hint: Re-use the big-step interpreter of week 2. See [Lambda](#).

**Step 4:** Up to a certain size, search for a term that violates Semantic Preservation. There should be none!

## 1 Examples

From a direct-style interpreter down to an abstract machine

From recursive traversal down to iterative traversal with link inversion

## 2 Formulations

## 3 Soundness

## 4 Remarks

## 5 Madness in small steps

## Control operators

In a CPS-transformed program, the continuation is a first-class object.

Why not give programmers [access](#) to it?

That is, extend the source language with [control operators](#) that allow [\(delimiting and\) capturing](#) the current continuation.

## Shift / reset

[Examples](#)[Interpreter](#)[Traversal](#)[Formulations](#)[Soundness](#)[Remarks](#)[Appendix](#)

An example is Danvy and Filinski's shift / reset (1990).

$$t ::= \dots | \langle t \rangle | \xi x. t$$

A "reset"  $\langle t \rangle$  does nothing by itself: e.g.,  $\langle 42 \rangle$  reduces to 42.

A "shift"  $\xi x. t$  captures the current evaluation context (up to and excluding the nearest reset), reifies it as a function, and binds the variable  $x$  to it.

Then it discards the evaluation context (up to and including the nearest reset) and executes  $t$  instead.

E.g., roughly,

$$\begin{aligned} & 1 + \langle 10 + \xi c. c (c 100) \rangle \\ \longrightarrow & 1 + (\text{let } c = \lambda x. (10 + x) \text{ in } c (c 100)) \\ \longrightarrow & 1 + (10 + (10 + 100)) \\ \longrightarrow & 121 \end{aligned}$$

Exercise: Give a small-step semantics to shift / reset.

## CPS-transforming shift / reset

The naïve call-by-value CPS transformation is extended as follows:

$$\llbracket \langle t \rangle \rrbracket = \lambda k.$$

## CPS-transforming shift / reset

The naïve call-by-value CPS transformation is extended as follows:

$$\begin{aligned} \llbracket \langle t \rangle \rrbracket &= \lambda k. k (\llbracket t \rrbracket (\lambda y. y)) \\ \llbracket \xi x. t \rrbracket &= \lambda k. \end{aligned}$$

## CPS-transforming shift / reset

[Examples](#)[Interpreter](#)[Traversal](#)[Formulations](#)[Soundness](#)[Remarks](#)[Appendix](#)

The naïve call-by-value CPS transformation is extended as follows:

$$\begin{aligned}\llbracket \langle t \rangle \rrbracket &= \lambda k. k (\llbracket t \rrbracket (\lambda y. y)) \\ \llbracket \xi x. t \rrbracket &= \lambda k. \text{let } x = \lambda y. \lambda k'. k' (k y) \text{ in} \\ &\quad \llbracket t \rrbracket (\lambda y. y)\end{aligned}$$

**Exercise (experimental!):** Extend the proof of Semantic Preservation.

The target of the transformation is  $\lambda$ -calculus **without** shift / reset.

It is **no longer the case** that every call is a tail call, that the right-hand side of every application is a value, or that continuations are linearly used.

Thus, shift / reset allow reaching terms which previously lied **outside** the image of the CPS transformation. CPS lets us **think outside the box!**

## Other control operators

Many other control operators or control constructs can be [explained](#) and [compiled away](#) via CPS.

[Exceptions](#) can be compiled away by “double-barrelled CPS”, that is, by using [two](#) continuations.

[Effect handlers](#) can be compiled away via (type-directed, selective) CPS.

Rompf, Maier, Odersky, [Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform](#), 2009.

Leijen, [Type-directed compilation of row-typed algebraic effects](#), 2017.

See Régis-Gianas’ lectures!

## Monadic intermediate form

Examples  
Interpreter  
Traversal  
Formulations  
Soundness  
Remarks  
Appendix

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

This transformation, too, achieves indifference:

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

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

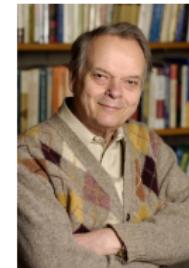
By further hoisting “let” out of the left-hand side of “let”,  
one gets administrative normal form.

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.

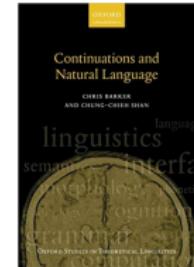
Today, heap-allocating the stack is considered [too costly](#):

- 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  $\lambda$ -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

Examples

Interpreter

Traversal

Formulations

Soundness

Remarks

Appendix

### 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.

We have illustrated a few proof techniques:

- A small-step simulation diagram, in a proof of semantic preservation.
- Testing, to refute a conjecture and find a counter-example!

## 1 Examples

From a direct-style interpreter down to an abstract machine

From recursive traversal down to iterative traversal with link inversion

## 2 Formulations

## 3 Soundness

## 4 Remarks

## 5 Madness in small steps

# Madness Soundness in small steps

Examples

Interpreter

Traversal

Formulations

Soundness

Remarks

Appendix

(Presented at MPRI 2.4 in 2017.)

Could we use a **small-step operational semantics**  
in the proof that CPS is semantics-preserving?

# Towards semantic preservation

Examples

Interpreter

Traversal

Formulations

Soundness

Remarks

Appendix

Let us consider the pure  $\lambda$ -calculus, without “let”.

Let us use de Bruijn notation.

The transformation is defined in [CPSDefinition](#).

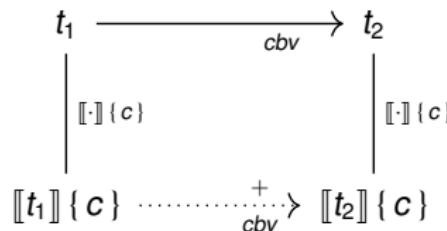
The proof of Simulation is in [CPSSimulationWithoutLet](#).

The key lemmas are in [CPSSpecialCases](#), [CPSSubstitution](#), [CPSKubstitution](#).

## A small-step simulation diagram

We propose to use the **small-step substitution** semantics and to establish a **simulation** diagram.

**One** step by the source program is simulated in **one or more** steps by the transformed program:

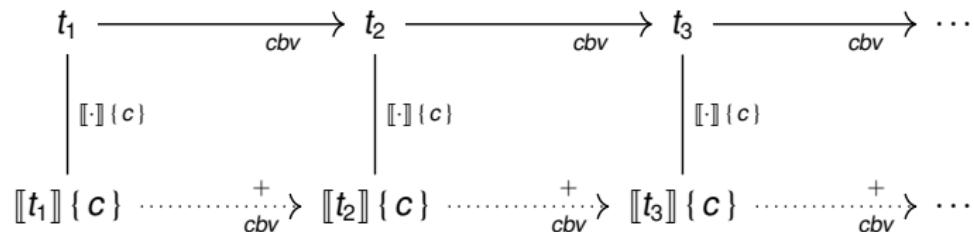


A solid arrow represents a **universal** quantification (a hypothesis).

A dashed arrow represents an **existential** quantification (a conclusion).

## Consequences of the simulation diagram

There immediately follows that divergence is preserved.



The fact that each step is simulated by **one or more** steps is crucial.

(A proof by co-induction. See [Relations/infseq\\_simulation](#).)

## Consequences of the simulation diagram

Obviously, **several** steps by the source program  
are simulated in **several** steps by the transformed program:

$$t_1 \xrightarrow[\text{cbv}]{\star} t_2$$

$\boxed{\llbracket t_1 \rrbracket \{c\}} \cdots \xrightarrow[\text{cbv}]{\star} \boxed{\llbracket t_2 \rrbracket \{c\}}$

(A proof by induction. See [Relations/star\\_diamond\\_left](#).)

## Consequences of the simulation diagram

There follows that convergence to a value is preserved.

We use the identity continuation *done*, defined as  $m \ 0$ .

$$\begin{array}{ccc} t & \xrightarrow[\text{cbv}]{\star} & v \\ \left\| \cdot \right\| \{ done \} & & \left\| \cdot \right\| \{ done \} \\ \left[ t \right] \{ done \} & \xrightarrow[\text{cbv}]{\star} & \left[ v \right] \{ done \} \end{array}$$

By definition,  $\left[ v \right] \{ done \}$  is *apply done* ( $v$ ), that is,  $(v)$ , therefore a value.

Thus, the CPS transformation is semantics-preserving.

# The simulation lemma

Examples

Interpreter

Traversal

Formulations

Soundness

Remarks

Appendix

Here is the simulation statement again, this time in textual form:

## Lemma (Simulation)

*Assume  $\text{reify } c$  is a value. Then  $t_1 \rightarrow_{\text{cbv}} t_2$  implies  $\llbracket t_1 \rrbracket \{ c \} \xrightarrow{+}_{\text{cbv}} \llbracket t_2 \rrbracket \{ c \}$ .*

Let us now do the proof.

Onscreen or in Coq? Both, probably.

See [CPSSimulationWithoutLet](#).

## Proof of Simulation – case $\beta_v$

**Case:**  $(\lambda t) v \rightarrow_{\text{cbv}} t[v/]$ . We must show:

$$\llbracket (\lambda t) v \rrbracket \{ c \} \xrightarrow{+}_{\text{cbv}} \llbracket t[v/] \rrbracket \{ c \}$$

By the Value-Value Application lemma, the left-hand term is:

$$\langle \lambda t \rangle \langle v \rangle (\text{reify } c)$$

By definition of  $\langle \lambda t \rangle$ , this is:

$$(\lambda \lambda (\llbracket \uparrow^1 t \rrbracket \{ o \ 0 \})) \langle v \rangle (\text{reify } c)$$

The transformed function is passed **an actual argument**  $\langle v \rangle$  and **a continuation**  $\text{reify } c$ .

Proof of Simulation – case  $\beta_v$ [Examples](#)[Interpreter](#)[Traversal](#)[Formulations](#)[Soundness](#)[Remarks](#)[Appendix](#)

$$(\lambda \lambda ([\![\uparrow^1 t]\!] \{ o \ 0 \})) \ (\!(v)\!) \ (reify \ c)$$

In two  $\beta$ -reduction steps, this term reduces to:

$$([\![\uparrow^1 t]\!] \{ o \ 0 \}) \ [\uparrow ((\!(v)\!)/)] \ [reify \ c /]$$

We have [two successive substitutions](#). This term could also be written using a single substitution that acts on variables 0 and 1:

$$([\![\uparrow^1 t]\!] \{ o \ 0 \}) \ [reify \ c \cdot (\!(v)\!) \cdot id]$$

(We won't use this fact, though.)

We now wish to [push](#) the substitutions inside, one after the other.

## Proof of Simulation – case $\beta_v$

$$(\llbracket \uparrow^1 t \rrbracket \{ o \ 0 \}) \ [ \uparrow (\langle v \rangle /) ] \ [ reify \ c / ]$$

By the Substitution lemma, the substitution  $\uparrow (\langle v \rangle /)$  acts on both **the term**  $\uparrow^1 t$  and **the continuation**  $o \ 0$ .

However,  $\uparrow (\langle v \rangle /)$  has no effect on variable 0.

Thus, the above term is:

$$(\llbracket (\uparrow^1 t)[\uparrow (v /)] \rrbracket \{ o \ 0 \}) \ [ reify \ c / ]$$

that is,

$$(\llbracket \uparrow^1 t[v /] \rrbracket \{ o \ 0 \}) \ [ reify \ c / ]$$

Proof of Simulation – case  $\beta_v$ [Examples](#)[Interpreter](#)[Traversal](#)[Formulations](#)[Soundness](#)[Remarks](#)[Appendix](#)

$$(\llbracket \uparrow^1 t[v/] \rrbracket \{ \circ 0 \}) \ [reify c/]$$

By the Kuststitution lemma, the substitution *reify c/* acts **only on the continuation  $\circ 0$ , not on the term  $t[v/]$** , because it cancels out with  $\uparrow^1$ .

Thus, this term is:

$$\llbracket t[v/] \rrbracket \{ (\circ 0) [reify c/] \}$$

that is,

$$\llbracket t[v/] \rrbracket \{ \circ (reify c) \}$$

## Proof of Simulation – case $\beta_v$

We have now reached the term:

$$\llbracket t[v/] \rrbracket \{ o (reify c) \}$$

and the goal is to prove that it reduces (in zero or more steps) to:

$$\llbracket t[v/] \rrbracket \{ o c \}$$

This is the Magic Step lemma. This proof case is finished!

Here are the four key lemmas that we have used so far.

### Lemma (Value-Value Application)

$$\llbracket v_1 \ v_2 \rrbracket \{ c \} = (\llbracket v_1 \rrbracket) (\llbracket v_2 \rrbracket) \text{ (reify } c\text{)}.$$

### Lemma (Substitution)

Let  $\sigma$  and  $\sigma'$  be value substitutions such that  $\sigma'$  is equal to  $\sigma ; (\cdot)$ . Then,

$$(\llbracket t \rrbracket \{ c \})[\sigma'] = \llbracket t[\sigma] \rrbracket \{ c[\sigma'] \}.$$

### Lemma (Kubstitution)

Let  $\theta$  and  $\sigma$  be substitutions such that  $\theta ; \sigma$  is id. Then,

$$\llbracket (t[\theta]) \{ c \} \rrbracket [\sigma] = \llbracket t \rrbracket \{ c[\sigma] \}.$$

### Lemma (Magic Step)

$$\llbracket t \rrbracket \{ o \text{ (reify } c \text{)} \} \xrightarrow{?_{cbv}} \llbracket t \rrbracket \{ c \}.$$

# Proof of Simulation – cases AppL and AppR

Examples

Interpreter

Traversal

Formulations

Soundness

Remarks

Appendix

**Case:**  $t_1 \ u \xrightarrow{\text{cbv}} t_2 \ u$ , where  $t_1 \xrightarrow{\text{cbv}} t_2$ .

We must show  $\llbracket t_1 \ u \rrbracket \{ c \} \xrightarrow{+_{\text{cbv}}} \llbracket t_2 \ u \rrbracket \{ c \}$ .

By definition of the CPS transformation, this is

$$\xrightarrow{+_{\text{cbv}}} \begin{array}{l} \llbracket t_1 \rrbracket \{ m \llbracket \uparrow^1 u \rrbracket \{ m \ 1 \ 0 \ \uparrow^2 (\text{reify } c) \} \} \\ \llbracket t_2 \rrbracket \{ m \llbracket \uparrow^1 u \rrbracket \{ m \ 1 \ 0 \ \uparrow^2 (\text{reify } c) \} \} \end{array}$$

# Proof of Simulation – cases AppL and AppR

Examples

Interpreter

Traversal

Formulations

Soundness

Remarks

Appendix

**Case:**  $t_1 \ u \xrightarrow{\text{cbv}} t_2 \ u$ , where  $t_1 \xrightarrow{\text{cbv}} t_2$ .

We must show  $\llbracket t_1 \ u \rrbracket \{ c \} \xrightarrow{+_{\text{cbv}}} \llbracket t_2 \ u \rrbracket \{ c \}$ .

By definition of the CPS transformation, this is

$$\xrightarrow{+_{\text{cbv}}} \begin{array}{c} \llbracket t_1 \rrbracket \{ m \llbracket \uparrow^1 u \rrbracket \{ m 1 0 \uparrow^2 (\text{reify } c) \} \} \\ \llbracket t_2 \rrbracket \{ m \llbracket \uparrow^1 u \rrbracket \{ m 1 0 \uparrow^2 (\text{reify } c) \} \} \end{array}$$

Wow – the induction hypothesis applies directly to this goal!

Indeed, *reify* ( $m \dots$ ) is a  $\lambda$ -abstraction, therefore a value.

This proof case is complete!

**Case:**  $v \ u_1 \xrightarrow{\text{cbv}} v \ u_2$ , where  $u_1 \xrightarrow{\text{cbv}} u_2$ .

Analogous to the previous case, using a Value-Term Application lemma.

We see in these proof cases that reduction under a context in the source program is translated to reduction at the root in the transformed program.

# Simulation in the presence of let constructs

Examples

Interpreter

Traversal

Formulations

Soundness

Remarks

Appendix

In the presence of “let” constructs, Simulation breaks down.

**Challenge:** can you find a (minimal) counter-example?

Hint: Enlist a machine’s help. (See next two slides.)

Enumerating  $\lambda$ -terms[Examples](#)[Interpreter](#)[Traversal](#)[Formulations](#)[Soundness](#)[Remarks](#)[Appendix](#)

Define the **size** of a term as follows: variables have size 0;  $\lambda$ -abstractions and applications contribute 1.

**Step 1:** In OCaml, implement an exhaustive **enumeration** of the  $\lambda$ -terms of size  $s$  and with at most  $n$  free variables. (Given as an exercise in week 1.)

```
(* Enumerate all variables between 0 and n excluded. *)
let var (n : int) (k : term -> unit) : unit = ...
(* Enumerate all manners of splitting an integer s. *)
let split (s : int) (k : int -> int -> unit) : unit = ...
(* Enumerate all terms of size s with at most n variables. *)
let term (s : int) (n : int) (k : term -> unit) : unit = ...
```

An enumerator is naturally written in CPS style!

# Testing Simulation

**Step 2:** In OCaml, implement the CPS transformation.

```
type continuation =
| O of term
| M of term
let cps (t : term) (c : continuation) : term = ...
```

**Step 3:** In OCaml, implement a test for the relation  $\cdot \longrightarrow_{\text{cbv}}^* \cdot$ :

```
let reduces (t1 : term) (t2 : term) : bool = ...
```

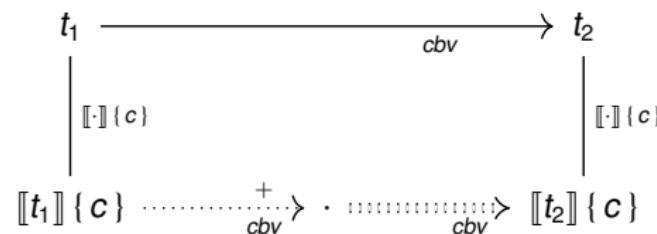
Hint: Re-use the auxiliary functions of week 2. See [Lambda](#).

**Step 4:** Find a term  $t_1$  of minimal size that violates Simulation.

Solution: see [CPSCounterExample](#).

# Fixing Simulation

In the presence of “let”, Simulation can be fixed as follows:



We allow one step of parallel call-by-value reduction  $\Rightarrow_{\text{cbv}}$ .

The proof of Simulation is more complex; see [CPSSimulation](#).

## Parallel (call-by-value) reduction

Examples

Interpreter

Traversal

Formulations

Soundness

Remarks

Appendix

Parallel reduction allows reducing **all** (currently visible) redexes at once, including under “ $\lambda$ ” and in the right-hand side of “let”.

PARALLEL  $\beta_v$ 

$$\frac{t_1 \Rightarrow_{\text{cbv}} t_2 \quad v_1 \Rightarrow_{\text{cbv}} v_2}{(\lambda t_1) v_1 \Rightarrow_{\text{cbv}} t_2[v_2/]}$$

PARALLEL  $\text{let}_v$ 

$$\frac{t_1 \Rightarrow_{\text{cbv}} t_2 \quad v_1 \Rightarrow_{\text{cbv}} v_2}{\text{let } v_1 \text{ in } t_1 \Rightarrow_{\text{cbv}} t_2[v_2/]} \quad x \Rightarrow_{\text{cbv}} x$$

$$\frac{}{t_1 \Rightarrow_{\text{cbv}} t_2}$$

$$\frac{}{\lambda t_1 \Rightarrow_{\text{cbv}} \lambda t_2}$$

$$\frac{}{t_1 \Rightarrow_{\text{cbv}} t_2}$$

$$\frac{}{t_1 u_1 \Rightarrow_{\text{cbv}} t_2 u_2}$$

$$\frac{}{u_1 \Rightarrow_{\text{cbv}} u_2}$$

$$\frac{}{t_1 \Rightarrow_{\text{cbv}} t_2}$$

$$\frac{}{\text{let } t_1 \text{ in } u_1 \Rightarrow_{\text{cbv}} \text{let } t_2 \text{ in } u_2}$$

The ability to **reduce under a binder** is needed to fix Simulation.

Call-by-name parallel reduction is studied by **Takahashi (1995)**.

**Crary (2009)** adapts these results to a call-by-value setting.

# Well-behavedness of parallel reduction

Examples

Interpreter

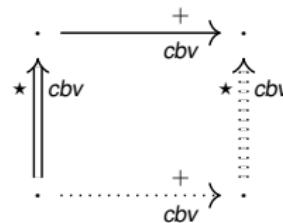
Traversal

Formulations

Soundness

Remarks

Appendix



## Lemma (Commutation)

$$(\Rightarrow_{cbv}^* ; \longrightarrow_{cbv}^+) \subseteq (\longrightarrow_{cbv}^+ ; \Rightarrow_{cbv}^*).$$

See [LambdaCalculusStandardization/pcbv\\_cbv\\_commutation](#).

# Well-behavedness of parallel reduction

Examples

Interpreter

Traversal

Formulations

Soundness

Remarks

Appendix

## Lemma (Equiconvergence)

$$(\exists v, t \Rightarrow_{cbv}^* v) \iff (\exists v', t \longrightarrow_{cbv}^* v').$$

(The idea is,  $v'$  reduces to  $v$  via **internal** parallel reduction steps.)

See [LambdaCalculusStandardization/equiconvergence](#).

# Consequences of Fixed Simulation

[Examples](#)[Interpreter](#)[Traversal](#)[Formulations](#)[Soundness](#)[Remarks](#)[Appendix](#)

There follows that **divergence** is preserved.

Indeed, from:

$$t \xrightarrow{\text{cbv}} \cdot \xrightarrow{\text{cbv}} \cdots$$

we get:

$$\llbracket t \rrbracket \{ c \} \xrightarrow[\text{cbv}]{+} \cdot \Rightarrow_{\text{cbv}} \cdot \xrightarrow[\text{cbv}]{+} \cdot \Rightarrow_{\text{cbv}} \cdots$$

which, by Commutation, yields:

$$\llbracket t \rrbracket \{ c \} \xrightarrow[\text{cbv}]{+} \cdot \xrightarrow[\text{cbv}]{+} \cdot \Rightarrow^*_{\text{cbv}} \cdot \Rightarrow_{\text{cbv}} \cdots$$

that is,

$$\llbracket t \rrbracket \{ c \} \xrightarrow[\text{cbv}]{{\geq} 2} \cdot \Rightarrow^*_{\text{cbv}} \cdots$$

And so on. For an arbitrary  $n \geq 0$ , we have:

$$\llbracket t \rrbracket \{ c \} \xrightarrow[\text{cbv}]{{\geq} n} \cdot \Rightarrow^*_{\text{cbv}} \cdots$$

# Consequences of Fixed Simulation

[Examples](#)[Interpreter](#)[Traversal](#)[Formulations](#)[Soundness](#)[Remarks](#)[Appendix](#)

Convergence to a value is preserved, too.

Indeed, from:

$$t \xrightarrow[n]{\text{cbv}} v$$

we get, as on the previous slide:

$$\llbracket t \rrbracket \{ \text{done} \} \xrightarrow[\text{cbv}]{\geq n} \cdot \Rightarrow_{\text{cbv}}^{\star} (v)$$

and, by Equiconvergence:

$$\exists v' \quad \llbracket t \rrbracket \{ \text{done} \} \xrightarrow[\text{cbv}]{\geq n} \cdot \xrightarrow[\text{cbv}]{\star} v'$$

The CPS transformation remains semantics-preserving in the presence of “let” constructs (phew!).