Making the stack explicit:
the continuation-passing style transformation

MPRI 2.4

François Pottier

2017

Motivation

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?

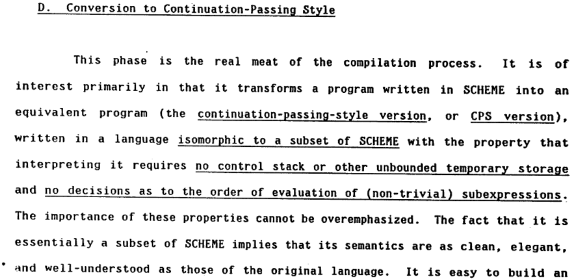
Motivation

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. 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

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

A direct-style interpreter

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

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

```
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

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

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

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

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

More formally,

$$
\begin{array}{rcll}
v & ::= & x \mid \lambda x.tt & \text{values} \\
tt & ::= & & \text{terms in tail position} \\
 & \mid & v & \\
 & \mid & nt\ nt & \text{– a tail call} \\
 & \mid & \text{let } nt \text{ in } tt & \\
 & \mid & \text{if } nt \text{ then } tt \text{ else } tt & \\
nt & ::= & & \text{terms not in tail position} \\
 & \mid & v & \\
 & \mid & nt\ nt & \text{– an ordinary call} \\
 & \mid & \text{let } nt \text{ in } nt & \\
 & \mid & \text{if } nt \text{ then } nt \text{ else } nt &
\end{array}
$$

Verified tail calls

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

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

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

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

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

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

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

`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

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

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

A type of binary trees

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.

CPS traversal

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

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

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

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.

Inded, 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).

More recycling

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

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

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

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

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

“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.)

Formulations of the CPS transformation

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 λ-calculus, 1975.

Definition of the CBV CPS transformation

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

$$[\![x]\!] =$$

Definition of the CBV CPS transformation

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

$$[\![x]\!] = \lambda k.$$

Definition of the CBV CPS transformation

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

$$
\begin{aligned}
[\![x]\!] &= \lambda k.\ k\ x \\
[\![\lambda x.t]\!] &=
\end{aligned}
$$

Definition of the CBV CPS transformation

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

$$\begin{aligned} [\![x]\!] &= \lambda k.\ k\ x \\ [\![\lambda x.t]\!] &= \lambda k. \end{aligned}$$

Definition of the CBV CPS transformation

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

$$\begin{aligned} [\![x]\!] &= \lambda k.\ k\ x \\ [\![\lambda x.t]\!] &= \lambda k.\ k\ (\lambda x.[\![t]\!]) \\ [\![t_1\ t_2]\!] &= \lambda k. \end{aligned}$$

Definition of the CBV CPS transformation

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

$$
\begin{aligned}
[\![x]\!] &= \lambda k.\ k\ x \\
[\![\lambda x.t]\!] &= \lambda k.\ k\ (\lambda x.[\![t]\!]) \\
[\![t_1\ t_2]\!] &= \lambda k.\ [\![t_1]\!]
\end{aligned}
$$

Definition of the CBV CPS transformation

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

$$\begin{aligned}
[\![x]\!] &= \lambda k.\ k\ x \\
[\![\lambda x.t]\!] &= \lambda k.\ k\ (\lambda x.[\![t]\!]) \\
[\![t_1\ t_2]\!] &= \lambda k.\ [\![t_1]\!]\ (\lambda x_1.
\end{aligned}$$

Definition of the CBV CPS transformation

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

$$\begin{aligned} [\![x]\!] &= \lambda k.\ k\ x \\ [\![\lambda x.t]\!] &= \lambda k.\ k\ (\lambda x.[\![t]\!]) \\ [\![t_1\ t_2]\!] &= \lambda k.\ [\![t_1]\!]\ (\lambda x_1.\ [\![t_2]\!]\ (\lambda x_2. \end{aligned}$$

Definition of the CBV CPS transformation

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

$$\begin{aligned}
[\![x]\!] &= \lambda k.\ k\ x \\
[\![\lambda x.t]\!] &= \lambda k.\ k\ (\lambda x.[\![t]\!]) \\
[\![t_1\ t_2]\!] &= \lambda k.\ [\![t_1]\!]\ (\lambda x_1.\ [\![t_2]\!]\ (\lambda x_2.\ x_1\ x_2\ k)) \\
[\![\text{let } x = t_1 \text{ in } t_2]\!] &= \lambda k.\ [\![t_1]\!]\ (\lambda x.\ [\![t_2]\!]\ k)
\end{aligned}$$

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 distinguishing the translation of terms $[\![t]\!]$ and the translation of values $(\!|v|\!)$.

$$
\begin{aligned}
(\!|x|\!) &= x \\
(\!|\lambda x.t|\!) &= \lambda x.[\![t]\!] \\
[\![v]\!] &= \lambda k.\; k\; (\!|v|\!) \\
[\![t_1\; t_2]\!] &= \lambda k.\; [\![t_1]\!]\; (\lambda x_1.\; [\![t_2]\!]\; (\lambda x_2.\; x_1\; x_2\; k)) \\
[\![\text{let } x = t_1 \text{ in } t_2]\!] &= \lambda k.\; [\![t_1]\!]\; (\lambda x.\; [\![t_2]\!]\; k)
\end{aligned}
$$

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.

CPS can serve as an encoding of call-by-value into call-by-name.

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

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 computation of type $[\![T]\!]$.

$$\begin{aligned} (\!|\alpha|\!) &= \alpha \\ (\!|T_1 \rightarrow T_2|\!) &= \end{aligned}$$

Effect of the transformation of types

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 computation of type $[\![T]\!]$.

$$(\!|\alpha|\!) = \alpha$$

$$(\!|T_1 \rightarrow T_2|\!) = (\!|T_1|\!) \rightarrow$$

Effect of the transformation of types

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 computation of type $[\![T]\!]$.

$$(\!|\alpha|\!) = \alpha$$

$$(\!|T_1 \rightarrow T_2|\!) = (\!|T_1|\!) \rightarrow [\![T_2]\!]$$

$$[\![T]\!] =$$

Effect of the transformation of types

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 computation of type $[\![T]\!]$.

$$(\!|\alpha|\!) = \alpha$$
$$(\!|T_1 \rightarrow T_2|\!) = (\!|T_1|\!) \rightarrow [\![T_2]\!]$$

$$[\![T]\!] = ((\!|T|\!) \rightarrow A) \rightarrow A$$

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

One may take A to be the empty type 0. Then, $[\![T]\!]$ is $\neg\neg(\!|T|\!)$. 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?

$$[\![T]\!] = (\llparenthesis T \rrparenthesis \rightarrow A) \rightarrow A$$

Effect of the transformation of types – refined

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

$$[\![T]\!] = ((\!|T|\!) \to A) \to A$$

Every transformed term is in fact answer-type polymorphic:

$$[\![T]\!] = \forall A.((\!|T|\!) \to A) \to A$$

Furthermore,

Effect of the transformation of types – refined

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

$$[\![T]\!] = ((\!|T|\!) \rightarrow A) \rightarrow A$$

Every transformed term is in fact answer-type polymorphic:

$$[\![T]\!] = \forall A.((\!|T|\!) \rightarrow A) \rightarrow A$$

Furthermore, every transformed term invokes its continuation once:

$$[\![T]\!] = \forall A.((\!|T|\!) \rightarrow A) \multimap A$$

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

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

Administrative redexes

The translation presented so far is naïve.

It produces many “administrative” β-redexes.

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

$$\begin{array}{lcl}
\llbracket f\ x \rrbracket & = & \lambda k.\ \llbracket f \rrbracket\ (\lambda x_1.\ \llbracket x \rrbracket\ (\lambda x_2.\ x_1\ x_2\ k)) \\
 & = & \lambda k.\ (\lambda k.\ k\ \llparenthesis f \rrparenthesis)\ (\lambda x_1.\ (\lambda k.\ k\ \llparenthesis x \rrparenthesis)\ (\lambda x_2.\ x_1\ x_2\ k)) \\
 & = & \lambda k.\ (\lambda k.\ k\ f)\ (\lambda x_1.\ (\lambda k.\ k\ x)\ (\lambda x_2.\ x_1\ x_2\ k)) \\
 & =_\beta & \lambda k.\ (\lambda x_1.\ (\lambda k.\ k\ x)\ (\lambda x_2.\ x_1\ x_2\ k))\ f \\
 & =_\beta & \lambda k.\ (\lambda k.\ k\ x)\ (\lambda x_2.\ f\ x_2\ k) \\
 & =_\beta & \lambda k.\ (\lambda x_2.\ f\ x_2\ k)\ x \\
 & =_\beta & \lambda k.\ f\ x\ k
\end{array}$$

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

Semantic preservation

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* $[\![t]\!]\ w \longrightarrow^{\star}_{cbv} w\ (\!|v|\!)$.

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 λ that corresponds to a source λ.
- After the translation, reduce every redex whose λ 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

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 $[\![t]\!] = \lambda k.\ \ldots$ as a function of a term to a term,
let us view $[\![t]\!]\,\{\,w\,\} = \ldots$ as a function of a term and a value to a term.

$$
\begin{aligned}
(\!|x|\!) &= x \\
(\!|\lambda x.t|\!) &= \lambda x.\lambda k.\ [\![t]\!]\,\{\,k\,\}
\end{aligned}
$$

$$
\begin{aligned}
[\![v]\!]\,\{\,w\,\} &= w\ (\!|v|\!) \\
[\![t_1\ t_2]\!]\,\{\,w\,\} &= [\![t_1]\!]\,\{\,\lambda x_1.\ [\![t_2]\!]\,\{\,\lambda x_2.\ x_1\ x_2\ w\,\}\,\} \\
[\![\text{let } x = t_1 \text{ in } t_2]\!]\,\{\,w\,\} &= [\![t_1]\!]\,\{\,\lambda x.\ [\![t_2]\!]\,\{\,w\,\}\,\}
\end{aligned}
$$

k denotes a variable; w denotes a value.

Towards a one-pass transformation

This transformation produces fewer administrative redexes:

$$\begin{array}{rcl} [\![f\ x]\!]\,\{\,k\,\} & = & [\![f]\!]\,\{\,\lambda x_1.\,[\![x]\!]\,\{\,\lambda x_2.\,x_1\ x_2\ k\,\}\,\} \\ & = & (\lambda x_1.\,(\lambda x_2.\,x_1\ x_2\ k)\ x)\ f \\ & =_\beta & (\lambda x_2.\,f\ x_2\ k)\ x \\ & =_\beta & f\ x\ k \end{array}$$

The remaining administrative redexes arise from the equation

$$[\![v]\!]\,\{\,w\,\} = w\ (\!|v|\!)$$

in the case where the continuation w is a λ-abstraction.

How could we alter this equation?

Towards a one-pass transformation

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

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

Note:

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

A one-pass transformation

Change the translation of values. Make every “transformation” λ linear.

$$
\begin{aligned}
(\!|x|\!) &= x \\
(\!|\lambda x.t|\!) &= \lambda x.\lambda k.\ [\![t]\!]\,\{\,k\,\} \\
[\![v]\!]\,\{\,w\,\} &= w \mathbin{@_\beta} (\!|v|\!) \\
[\![t_1\ t_2]\!]\,\{\,w\,\} &= [\![t_1]\!]\,\{\,\lambda x_1.\ [\![t_2]\!]\,\{\,\lambda x_2.\ x_1\ x_2\ w\,\}\,\} \\
[\![\text{let } x = t_1 \text{ in } t_2]\!]\,\{\,w\,\} &= [\![t_1]\!]\,\{\,\lambda x.\ \text{let } x = x \text{ in } [\![t_2]\!]\,\{\,w\,\}\,\}
\end{aligned}
$$

This transformation produces no administrative redexes.

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

A one-pass transformation

Look Ma, no administrative redexes!

$$\begin{array}{rcl}
[\![f\ x]\!]\,\{\,k\,\} & = & [\![f]\!]\,\{\,\lambda x_1.\,[\![x]\!]\,\{\,\lambda x_2.\,x_1\ x_2\ k\,\}\,\} \\
 & = & (\lambda x_1.\,(\lambda x_2.\,x_1\ x_2\ k)\ @_\beta\ x)\ @_\beta\ f \\
 & = & (\lambda x_2.\,f\ x_2\ k)\ @_\beta\ x \\
 & = & f\ x\ k
\end{array}$$

One drawback of Dargaye and Leroy's formulation is that $\cdot\ @_\beta\ \cdot$ does not commute with substitutions.

This is repaired in the formulations shown next...

A higher-order formulation

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

Their formulation was in a "higher-order" style.

Let a continuation c be either an arbitrary object or a "transformation" λ:

$$\begin{array}{l} \kappa ::= \langle \text{a meta-level function } v \Rightarrow t \text{ of values to terms} \rangle \\ c ::= \mathsf{o}\ w \mid \mathsf{m}\ \kappa \end{array}$$

Define smart application *apply c v* and reification *reify c* as follows:

$$\begin{array}{rll} \mathit{apply}\ (\mathsf{o}\ w)\ v = w\ v & & \text{– an object-level application} \\ \mathit{apply}\ (\mathsf{m}\ \kappa)\ v = \kappa(v) & & \text{– a meta-level application} \\ \mathit{reify}\ (\mathsf{o}\ w) = w & & \text{– a no-op} \\ \mathit{reify}\ (\mathsf{m}\ \kappa) = \lambda x.(\kappa(x)) & & \text{– a “two-level } \eta\text{-expansion”} \end{array}$$

A higher-order formulation

Danvy and Filinski's transformation is then formulated as follows:

$$\begin{aligned}
(\!|x|\!) &= x \\
(\!|\lambda x.t|\!) &= \lambda x.\lambda k.\,[\![t]\!]\,\{\,\mathrm{o}\ k\,\}
\end{aligned}$$

$$\begin{aligned}
[\![v]\!]\,\{\,c\,\} &= \mathit{apply}\ c\ (\!|v|\!) \\
[\![t_1\ t_2]\!]\,\{\,c\,\} &= [\![t_1]\!]\,\{\,\mathrm{m}\ v_1 \Rightarrow [\![t_2]\!]\,\{\,\mathrm{m}\ v_2 \Rightarrow v_1\ v_2\ (\mathit{reify}\ c)\,\}\,\} \\
[\![\mathrm{let}\ x = t_1\ \mathrm{in}\ t_2]\!]\,\{\,c\,\} &= [\![t_1]\!]\,\{\,\mathrm{m}\ v_1 \Rightarrow \mathrm{let}\ x = v_1\ \mathrm{in}\ [\![t_2]\!]\,\{\,c\,\}\,\}
\end{aligned}$$

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

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

A first-order reformulation

Danvy and Filinski's transformation
can just as well be presented in a "first-order" style.

No need for meta-level functions!

Let us just view m as a binder – roughly, a "transformation" λ:

$$c ::= \mathrm{o}\ w \mid \mathrm{m}x.t$$

Define smart application *apply c v* and reification *reify c* as follows:

$$\begin{aligned} \mathit{apply}\ (\mathrm{o}\ w)\ v &= w\ v && \text{– an object-level application} \\ \mathit{apply}\ (\mathrm{m}x.t)\ v &= t[v/x] && \text{– a meta-level substitution} \end{aligned}$$

$$\begin{aligned} \mathit{reify}\ (\mathrm{o}\ w) &= w && \text{– a no-op} \\ \mathit{reify}\ (\mathrm{m}x.t) &= \lambda x.t && \text{– a "two-level } \eta\text{-expansion"} \end{aligned}$$

A first-order reformulation

Danvy and Filinski's transformation is then reformulated as follows:

$$\begin{aligned} (\!| x |\!) &= x \\ (\!| \lambda x.t |\!) &= \lambda x.\lambda k.\,[\![t]\!]\,\{\, \mathrm{o}\ k \,\} \end{aligned}$$

$$\begin{aligned} [\![v]\!]\,\{\, c \,\} &= \mathit{apply}\ c\ (\!| v |\!) \\ [\![t_1\ t_2]\!]\,\{\, c \,\} &= [\![t_1]\!]\,\{\, \mathrm{m}x_1.[\![t_2]\!]\,\{\, \mathrm{m}x_2.x_1\ x_2\ (\mathit{reify}\ c) \,\}\,\} \\ [\![\mathrm{let}\ x = t_1\ \mathrm{in}\ t_2]\!]\,\{\, c \,\} &= [\![t_1]\!]\,\{\, \mathrm{m}x_1.\mathrm{let}\ x = x_1\ \mathrm{in}\ [\![t_2]\!]\,\{\, c \,\}\,\} \end{aligned}$$

This formulation is simpler than the higher-order formulation.

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

A likely reason why Danvy and Filinski did not adopt this formulation is that their higher-order formulation is closer to an efficient implementation.

The first-order formulation in de Bruijn style

We still view m as a binder:

$$c ::= \mathsf{o}\ w \mid \mathsf{m}\ t$$

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

$$\begin{array}{rcll}
\mathit{apply}\ (\mathsf{o}\ w)\ v & = & w\ v & \text{– an object-level application} \\
\mathit{apply}\ (\mathsf{m}\ t)\ v & = & t[v/] & \text{– a meta-level substitution operation} \\
\mathit{reify}\ (\mathsf{o}\ w) & = & w & \text{– a no-op} \\
\mathit{reify}\ (\mathsf{m}\ t) & = & \lambda t & \text{– a two-level } \eta\text{-expansion} \\
(\mathsf{o}\ w)[\sigma] & = & \mathsf{o}\ (w[\sigma]) & \text{– apply } \sigma \\
(\mathsf{m}\ t)[\sigma] & = & \mathsf{m}\ (t[\Uparrow \sigma]) & \text{– apply } \sigma \text{ under the binding construct m}
\end{array}$$

The first-order formulation in de Bruijn style

The transformation is formulated in de Bruijn style as follows:

$$\begin{aligned}
\llparenthesis x \rrparenthesis &= x \\
\llparenthesis \lambda t \rrparenthesis &= \lambda\lambda(\llbracket \uparrow^1 t \rrbracket \, \{\, \mathrm{o}\ 0 \,\})
\end{aligned}$$

$$\begin{aligned}
\llbracket v \rrbracket \, \{\, c \,\} &= \mathit{apply}\ c\ \llparenthesis v \rrparenthesis \\
\llbracket t_1\ t_2 \rrbracket \, \{\, c \,\} &= \llbracket t_1 \rrbracket \, \{\, \mathrm{m}\ \llbracket \uparrow^1 t_2 \rrbracket \, \{\, \mathrm{m}\ 1\ 0\ \uparrow^2 (\mathit{reify}\ c) \,\} \,\} \\
\llbracket \mathrm{let}\ t_1\ \mathrm{in}\ t_2 \rrbracket \, \{\, c \,\} &= \llbracket t_1 \rrbracket \, \{\, \mathrm{m}\ \mathrm{let}\ 0\ \mathrm{in}\ \llbracket \uparrow^1_1 t_2 \rrbracket \, \{\, \uparrow^2 c \,\} \,\}
\end{aligned}$$

$\uparrow^i t$ is short for $t[+i]$. $\uparrow^1_1 t$ is short for $t[\Uparrow(+1)]$.

$\uparrow^1$ can be read as an end-of-scope mark for variable 0.

$\uparrow^2$ can be read as an end-of-scope mark for variables 0 and 1.

$\uparrow^1_1$ can be read as an end-of-scope mark for variable 1.

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

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

Towards semantic preservation

Let us consider the pure λ-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:

$$\begin{tikzcd}
t_1 \arrow[r, "cbv"'] \arrow[d, dash, "\llbracket\cdot\rrbracket\,\{\,c\,\}"] & t_2 \arrow[d, dash, "\llbracket\cdot\rrbracket\,\{\,c\,\}"] \\
\llbracket t_1\rrbracket\,\{\,c\,\} \arrow[r, dotted, "+", "cbv"'] & \llbracket t_2\rrbracket\,\{\,c\,\}
\end{tikzcd}$$

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.

$$\begin{tikzcd}
t_1 \arrow[r, "cbv"'] \arrow[d, dash, "\llbracket\cdot\rrbracket\,\{\,c\,\}"] & t_2 \arrow[r, "cbv"'] \arrow[d, dash, "\llbracket\cdot\rrbracket\,\{\,c\,\}"] & t_3 \arrow[r, "cbv"'] \arrow[d, dash, "\llbracket\cdot\rrbracket\,\{\,c\,\}"] & \cdots \\
\llbracket t_1\rrbracket\,\{\,c\,\} \arrow[r, dotted, "+", "cbv"'] & \llbracket t_2\rrbracket\,\{\,c\,\} \arrow[r, dotted, "+", "cbv"'] & \llbracket t_3\rrbracket\,\{\,c\,\} \arrow[r, dotted, "+", "cbv"'] & \cdots
\end{tikzcd}$$

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:

$$\begin{tikzcd}
t_1 \arrow[r, "\star", "cbv"'] \arrow[d, dash, "\llbracket \cdot \rrbracket \{ c \}"] & t_2 \arrow[d, dash, "\llbracket \cdot \rrbracket \{ c \}"] \\
\llbracket t_1 \rrbracket \{ c \} \arrow[r, dotted, "\star", "cbv"'] & \llbracket t_2 \rrbracket \{ c \}
\end{tikzcd}$$

(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.

$$\begin{tikzcd}
t \arrow[rr, "\star", "cbv"'] \arrow[d, dash, "\llbracket \cdot \rrbracket \{ \mathit{done} \}"] & & v \arrow[d, dash, "\llbracket \cdot \rrbracket \{ \mathit{done} \}"] \\
\llbracket t \rrbracket \{ \mathit{done} \} \arrow[rr, dotted, "\star", "cbv"'] & & \llbracket v \rrbracket \{ \mathit{done} \}
\end{tikzcd}$$

By definition, $\llbracket v \rrbracket \{ \mathit{done} \}$ is $\mathit{apply}\ \mathit{done}\ (\!| v |\!)$, that is, $(\!| v |\!)$, therefore a value.

Thus, the CPS transformation is semantics-preserving.

The simulation lemma

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

Lemma (Simulation)

Assume reify c is a value. Then $t_1 \longrightarrow_{cbv} t_2$ *implies* $[\![t_1]\!]\,\{\,c\,\} \longrightarrow^{+}_{cbv} [\![t_2]\!]\,\{\,c\,\}$.

Let us now do the proof.

Onscreen or in Coq? Both, probably.

See `CPSSimulationWithoutLet`.

Proof of Simulation – case β_v

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

$$[\![(\lambda t)\ v]\!]\,\{\,c\,\} \longrightarrow^{+}_{\text{cbv}} [\![t[v/]]\!]\,\{\,c\,\}$$

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

$$(\!|\lambda t|\!)\ (\!|v|\!)\ (\mathit{reify}\ c)$$

By definition of $(\!|\lambda t|\!)$, this is:

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

The transformed function is passed an actual argument $(\!|v|\!)$ and a continuation *reify c*.

Proof of Simulation – case β_v

$$(\lambda\lambda(\llbracket\uparrow^1 t\rrbracket\,\{\,\mathrm{o}\ 0\,\}))\ (\!|v|\!)\ (\mathit{reify}\ c)$$

In two β-reduction steps, this term reduces to:

$$(\llbracket\uparrow^1 t\rrbracket\,\{\,\mathrm{o}\ 0\,\})\ \ [\Uparrow((\!|v|\!)/)]\ \ [\mathit{reify}\ c/]$$

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

$$(\llbracket\uparrow^1 t\rrbracket\,\{\,\mathrm{o}\ 0\,\})\ \ [\mathit{reify}\ c\cdot(\!|v|\!)\cdot \mathit{ids}]$$

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

We now wish to push the substitutions inside, one after the other.

Proof of Simulation – case β_v

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

By the Substitution lemma, the substitution $\Uparrow (\llparenthesis v \rrparenthesis /)$ acts on both the term $\uparrow^1 t$ and the continuation $\mathrm{o}\ 0$.

However, $\Uparrow (\llparenthesis v \rrparenthesis /)$ has no effect on variable 0.

Thus, the above term is:

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

that is,

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

Proof of Simulation – case β_v

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

By the Kubstitution lemma, the substitution $\mathit{reify}\ c/$ acts only on the continuation $\mathrm{o}\ 0$, not on the term $t[v/]$, because it cancels out with $\uparrow^1$.

Thus, this term is:

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

that is,

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

Proof of Simulation – case β_v

We have now reached the term:

$$[\![t[v/]]\!]\,\{\, \mathrm{o}\ (\mathit{reify}\ c)\,\}$$

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

$$[\![t[v/]]\!]\,\{\, \mathrm{o}\ c\,\}$$

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

Key Lemmas

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

Lemma (Value-Value Application)

$[\![v_1\ v_2]\!]\,\{\,c\,\} = (\!|v_1|\!)\ (\!|v_2|\!)\ (\mathit{reify}\ c)$.

Lemma (Substitution)

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

$$([\![t]\!]\,\{\,c\,\})[\sigma'] = [\![t[\sigma]]\!]\,\{\,c[\sigma']\,\}.$$

Lemma (Kubstitution)

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

$$[\![(t[\theta]]\!]\,\{\,c\,\})[\sigma] = [\![t]\!]\,\{\,c[\sigma]\,\}.$$

Lemma (Magic Step)

$[\![t]\!]\,\{\,\mathrm{o}\ (\mathit{reify}\ c)\,\} \longrightarrow^{?}_{cbv} [\![t]\!]\,\{\,c\,\}$.

Proof of Simulation – cases AppL and AppR

Case: $t_1\; u \longrightarrow_{\text{cbv}} t_2\; u$, where $t_1 \longrightarrow_{\text{cbv}} t_2$.

We must show $[\![t_1\; u]\!]\,\{\,c\,\} \longrightarrow^{+}_{\text{cbv}} [\![t_2\; u]\!]\,\{\,c\,\}$.

By definition of the CPS transformation, this is

$$\begin{array}{rl} & [\![t_1]\!]\,\{\,\mathrm{m}\;[\![\uparrow^1 u]\!]\,\{\,\mathrm{m}\;1\;0\;\uparrow^2(\mathit{reify}\;c)\,\}\,\} \\ \longrightarrow^{+}_{\text{cbv}} & [\![t_2]\!]\,\{\,\mathrm{m}\;[\![\uparrow^1 u]\!]\,\{\,\mathrm{m}\;1\;0\;\uparrow^2(\mathit{reify}\;c)\,\}\,\} \end{array}$$

Proof of Simulation – cases AppL and AppR

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

We must show $[\![t_1\ u]\!]\,\{\,c\,\} \longrightarrow^{+}_{\text{cbv}} [\![t_2\ u]\!]\,\{\,c\,\}$.

By definition of the CPS transformation, this is

$$\begin{array}{rl} & [\![t_1]\!]\,\{\,\mathsf{m}\ [\![\uparrow^1 u]\!]\,\{\,\mathsf{m}\ 1\ 0\uparrow^2(\mathit{reify}\ c)\,\}\,\} \\ \longrightarrow^{+}_{\text{cbv}} & [\![t_2]\!]\,\{\,\mathsf{m}\ [\![\uparrow^1 u]\!]\,\{\,\mathsf{m}\ 1\ 0\uparrow^2(\mathit{reify}\ c)\,\}\,\} \end{array}$$

Wow – the induction hypothesis applies directly to this goal!

Indeed, *reify* $(\mathsf{m}\ \ldots)$ is a λ-abstraction, therefore a value.

This proof case is complete!

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

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

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 λ-terms

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

Step 1: In OCaml, implement an exhaustive enumeration of the λ-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 $\cdot \longrightarrow^{\star}_{\text{cbv}} \cdot$:

```
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:

$$\begin{tikzcd}
t_1 \arrow[rr, "cbv"'] \arrow[d, dash, "\llbracket \cdot \rrbracket \{ c \}"] & & t_2 \arrow[d, dash, "\llbracket \cdot \rrbracket \{ c \}"] \\
\llbracket t_1 \rrbracket \{ c \} \arrow[r, dotted, "+", "cbv"'] & \cdot \arrow[r, Rightarrow, dotted, "cbv"'] & \llbracket t_2 \rrbracket \{ c \}
\end{tikzcd}$$

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

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

$$\textsc{Parallel}\ \beta_v\quad \frac{t_1 \Rightarrow_{\text{cbv}} t_2 \qquad v_1 \Rightarrow_{\text{cbv}} v_2}{(\lambda t_1)\ v_1 \Rightarrow_{\text{cbv}} t_2[v_2/]} \qquad \textsc{Parallel}\ \text{let}_v\quad \frac{t_1 \Rightarrow_{\text{cbv}} t_2 \qquad v_1 \Rightarrow_{\text{cbv}} v_2}{\text{let } v_1 \text{ in } t_1 \Rightarrow_{\text{cbv}} t_2[v_2/]} \qquad x \Rightarrow_{\text{cbv}} x$$

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

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

$$\begin{tikzcd}
\cdot \arrow[r, "+", "cbv"'] & \cdot \\
\cdot \arrow[u, Rightarrow, "\star", "cbv"'] \arrow[r, dotted, "+", "cbv"'] & \cdot \arrow[u, Rightarrow, dotted, "\star", "cbv"']
\end{tikzcd}$$

Lemma (Commutation)

$(\Rightarrow^{\star}_{cbv} \; ; \longrightarrow^{+}_{cbv}) \subseteq (\longrightarrow^{+}_{cbv} \; ; \Rightarrow^{\star}_{cbv})$.

See `LambdaCalculusStandardization/pcbv_cbv_commutation`.

Well-behavedness of parallel reduction

Lemma (Equiconvergence)

$$(\exists v,\ t \Rightarrow^{\star}_{cbv} v) \iff (\exists v',\ t \longrightarrow^{\star}_{cbv} v').$$

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

See `LambdaCalculusStandardization/equiconvergence`.

Consequences of Fixed Simulation

There follows that divergence is preserved.

Indeed, from:

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

we get:

$$[\![t]\!]\,\{\,c\,\} \longrightarrow^{+}_{\text{cbv}} \cdot \Rightarrow_{\text{cbv}} \cdot \longrightarrow^{+}_{\text{cbv}} \cdot \Rightarrow_{\text{cbv}} \cdots$$

which, by Commutation, yields:

$$[\![t]\!]\,\{\,c\,\} \longrightarrow^{+}_{\text{cbv}} \cdot \longrightarrow^{+}_{\text{cbv}} \cdot \Rightarrow^{\star}_{\text{cbv}} \cdot \Rightarrow_{\text{cbv}} \cdots$$

that is,

$$[\![t]\!]\,\{\,c\,\} \longrightarrow^{\geq 2}_{\text{cbv}} \cdot \Rightarrow^{\star}_{\text{cbv}} \cdots$$

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

$$[\![t]\!]\,\{\,c\,\} \longrightarrow^{\geq n}_{\text{cbv}} \cdot \Rightarrow^{\star}_{\text{cbv}} \cdots$$

Consequences of Fixed Simulation

Convergence to a value is preserved, too.

Indeed, from:

$$t \longrightarrow^{n}_{\text{cbv}} v$$

we get, as on the previous slide:

$$[\![t]\!]\,\{\,\mathit{done}\,\} \longrightarrow^{\geq n}_{\text{cbv}} \cdot \Rightarrow^{\star}_{\text{cbv}} (\!|v|\!)$$

and, by Equiconvergence:

$$\exists v' \quad [\![t]\!]\,\{\,\mathit{done}\,\} \longrightarrow^{\geq n}_{\text{cbv}} \cdot \longrightarrow^{\star}_{\text{cbv}} v'$$

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

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

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

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

$$t ::= \ldots \mid \langle t \rangle \mid \xi x.t$$

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,

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

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

CPS-transforming shift / reset

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

$$[\![\langle t \rangle]\!] = \lambda k.$$

CPS-transforming shift / reset

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

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

CPS-transforming shift / reset

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

$$\begin{array}{rl}[\![\langle t\rangle]\!] = & \lambda k.\ k\ ([\![t]\!]\ (\lambda y.y)) \\ [\![\xi x.t]\!] = & \lambda k.\ \mathsf{let}\ x = \lambda y.\lambda k'.\ k'\ (k\ y)\ \mathsf{in} \\ & [\![t]\!]\ (\lambda y.y)\end{array}$$

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

The target of the transformation is λ-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

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

This transformation, too, achieves indifference:

$$
\begin{aligned}
[\![x]\!] &= x \\
[\![\lambda x.t]\!] &= \lambda x.[\![t]\!] \\
[\![t_1\ t_2]\!] &= \text{let } x_1 = [\![t_1]\!] \text{ in} \\
&\ \text{let } x_2 = [\![t_2]\!] \text{ in} \\
&\ x_1\ x_2 \\
[\![\text{let } x = t_1 \text{ in } t_2]\!] &= \text{let } x = [\![t_1]\!] \text{ in } [\![t_2]\!]
\end{aligned}
$$

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.

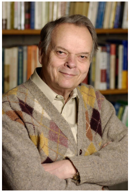
Flanagan, Sabry, Felleisen, The essence
of compiling with continuations, 1993 (2003).

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.

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.

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!