Compiling away first-class functions: closure conversion and defunctionalization

MPRI 2.4

François Pottier

2017

Program transformations

Last week, we studied how to interpret functional programs.

This week and in the next weeks, we compile them to lower-level code.

- This leads to more efficient execution. (No interpretive overhead.)
- This helps understand advanced language features, such as
  - first-class functions,
  - recursion,
  - exceptions, effect handlers, etc.
- This helps understand the organization of memory:
  - code versus data,
  - the stack versus the heap.
- This is an occasion to learn some programming techniques.
- This is an occasion to use operational semantics and do proofs.

Closure conversion apparatus for existing closure applicating machines

U.S. Patent Oct. 20, 1981 Sheet 3 of 10 4,295,320

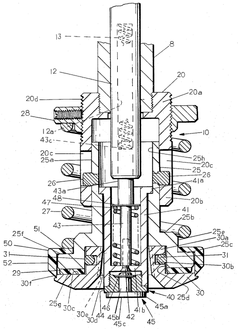


FIG. 3

Closure conversion apparatus for existing closure applicating machines

U.S. Patent Oct. 20, 1981 Sheet 3 of 10 4,295,320

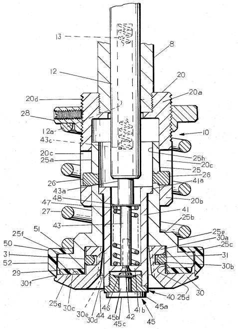


FIG. 3

Innovation in the carbonated beverage bottling industry is very much dependent on the ready availability of machinery for processing new types of containers and/or **closures**.

This invention provides substitute capping heads to apply a threaded closure on a bottle neck through the utilization of existing machines which were designed to apply an aluminum closure on a bottle neck by in situ formation of the threads in an aluminum shell.

Motivation

We wish to compile (translate):

- a language with arbitrary first-class functions (i.e., λ-abstractions)

down to:

- a language with closed first-class functions (i.e., “code pointers”)

Motivation

Why might we wish to understand this compilation technique?

- a key step in the compilation of functional programming languages;
- a way of explaining the magic of first-class functions;
- a way of understanding their space and time cost;
- a programming technique in languages without first-class functions.

Example 1 / downward funargs

A functional programming language, such as OCaml, naturally allows:

- defining local (nested) functions,
- passing a function as an argument to a function.

This combination of features is sometimes known as “downward funargs”.

```
let iter f t =
  for i = 0 to Array.length t - 1 do f t.(i) done
let sum t =
  let s = ref 0 in
  let add x = (s := !s + x) in
  iter add t;
  !s
```

`add` refers to the variable `s`, which is neither global nor local to `add`.

A nested function can refer to a local variable of an enclosing function.

Example 2 / upward funargs

A functional programming language also allows:

- returning a function out of a function;
- storing a function in a reference, for future use.

A nested function can thus outlive a local variable to which it refers.

This is sometimes called "upward funargs", although this does not really mean anything.

First-class functions have unbounded lifetimes;
therefore, their data cannot be stack-allocated.

Example 2 / upward funargs

A memory cell that is accessible only via a pair of `get` and `set` functions:

```
let make x =
  let cell = ref x in
  let get () = !cell
  and set  x = (cell := x) in
  get, set

let () =
  let get, set = make 3 in
  set (get() + 1)
```

A typical example of procedural abstraction (Reynolds, 1975), which is widely popular in object-oriented programming languages.

`cell` is a local variable in `make`.
It no longer exists when `get` and `set` are called!

How can we transform this code so as to use only closed functions? ...

Principles of closure conversion

`get` and `set` need access to the value of the local variable `cell`.
(This value is the address of a heap-allocated reference cell.)

- Therefore, they need one more parameter, an environment, which somehow gives access to this value;

At a call site, we must be able to supply an environment.

- Therefore, a λ-abstraction must evaluate to a closure, which gives access to both the code and the environment.

A closure must be heap-allocated, as its lifetime is unbounded.

Example 2 / manual closure conversion

The result of closure conversion could be as follows:

```
let make x =
  let cell = ref x in
  let get (env, ()) = !(env.cell)
  and set (env, x) = (env.cell := x) in
  { code = get; cell = cell }, { code = set; cell = cell }

let () =
  let get, set = make 3 in
  set.code (set, get.code (get, ()) + 1)
```

`get` and `set` are now closed functions: they have no free variables.

They can be hoisted out of `make`, if desired...

Example 2 / manual closure conversion

Here is the result of hoisting the closed functions up to the top level:

```
let get (env, ()) = !(env.cell)
let set (env, x) = (env.cell := x)
let make x =
  let cell = ref x in
  { code = get; cell = cell }, { code = set; cell = cell }

let () =
  let get, set = make 3 in
  set.code (set, get.code (get, ()) + 1)
```

Example 2 / manual closure conversion

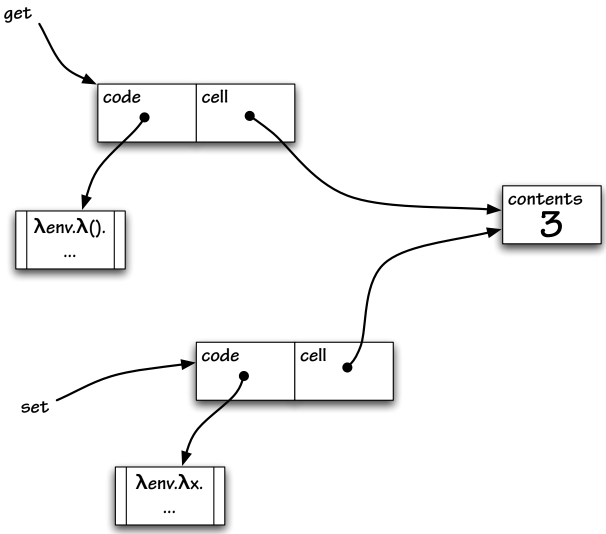
`{ code = get; cell = cell }` allocates a closure.

A record (i.e., a memory block),

- whose `code` field contains a closed function (i.e., a code pointer),
- whose other fields (`cell`) store the values which this function needs.

Here, code pointer and environment form a single memory block:
this is a flat closure.

The heap after `make 3`



Example 3

What could be the result of closure-converting this code?

```
let rec map f xs =
  match xs with
  | [] ->
      []
  | x :: xs ->
      f x :: map f xs

let scale k xs =
  map (fun x -> k * x) xs
```

Perform selective conversion – do not convert `map` and `scale`, which are closed functions.

Example 3

The anonymous function `fun x -> k * x` becomes a closure allocation.

```
let rec map f xs =
  match xs with
  | [] ->
      []
  | x :: xs ->
      f.code (f, x) :: map f xs

let scale k xs =
  map { code = (fun (env, x) -> env.k * x); k = k } xs
```

The unknown-function call `f x` is compiled to a closure invocation.

Definition of closure conversion

$$
\begin{aligned}
[\![x]\!] &= x \\
[\![\lambda x.t]\!] &= \text{let } \mathit{code} = \\
&\qquad \lambda(\mathit{clo}, x). \\
&\qquad\quad \text{let } x_i = \pi_i \ \mathit{clo} \text{ in} \\
&\qquad\quad [\![t]\!] \\
&\quad \text{in} \\
&\quad (\mathit{code}, x_1, \ldots, x_n) \\
[\![t_1 \ t_2]\!] &= \text{let } \mathit{clo} = [\![t_1]\!] \text{ in} \\
&\quad \text{let } (\mathit{code}, \ldots) = \mathit{clo} \text{ in} \\
&\quad \mathit{code} \ (\mathit{clo}, [\![t_2]\!])
\end{aligned}
$$

where $\{x_1, \ldots, x_n\} = \mathit{fv}(\lambda x.t)$
– note: this λ-abstraction is closed!
for each $i \in \{1, \ldots, n\}$

The target calculus must have tuples. We use the following sugar:

$$
\begin{aligned}
\text{let } (\mathit{code}, \ldots) = \mathit{clo} \text{ in } t &\equiv \text{let } \mathit{code} = \pi_0 \ \mathit{clo} \text{ in } t \\
\lambda(x, y).t &\equiv \lambda p.\text{let } x = \pi_0 \ p \text{ in let } y = \pi_1 \ p \text{ in } t
\end{aligned}
$$

Soundness of closure conversion

We would like to state that this program transformation is sound.

Soundness of closure conversion

We would like to state that this program transformation is sound.

That is, closure conversion preserves the meaning of programs.

We need a semantic preservation statement,

Soundness of closure conversion

We would like to state that this program transformation is sound.

That is, closure conversion preserves the meaning of programs.

We need a semantic preservation statement,

roughly:

> *If t exhibits a certain behavior,*
> *then $[\![t]\!]$ exhibits the same behavior.*

How can / should this be stated, more precisely?

Towards a semantic preservation statement

To write down such a statement, we must choose:

- a semantics for the source calculus;
- a semantics for the target calculus.

Thoughts?

Which semantics for the source calculus?

For the λ-calculus, we have encountered several semantics:

- small-step, substitution-based;
- big-step, substitution-based;
- big-step, environment-based;
- interpreter, with fuel, environment-based.

They are equivalent to one another,
but perhaps one of them is more convenient for this proof?

Which semantics for the source calculus?

For the λ-calculus, we have encountered several semantics:

- small-step, substitution-based;
- big-step, substitution-based;
- big-step, environment-based;
- interpreter, with fuel, environment-based.

They are equivalent to one another,
but perhaps one of them is more convenient for this proof?

Let us choose the big-step, environment-based semantics.

It already has explicit notions of environments and closures,
so this choice should make the proof easier.

Which semantics for the target calculus?

The target of closure conversion is also a λ-calculus.

We could therefore use the same semantics for it...?

Which semantics for the target calculus?

The target of closure conversion is also a λ-calculus.

We could therefore use the same semantics for it...?

The statement of semantic preservation would then be:

If $e \vdash t \downarrow_{cbv} c$, *then* $[\![e]\!] \vdash [\![t]\!] \downarrow_{cbv} [\![c]\!]$.

where the same semantics appears in the hypothesis and conclusion.

We could prove such a theorem, but...

Which semantics for the target calculus?

The target of closure conversion is also a λ-calculus.

We could therefore use the same semantics for it...?

The statement of semantic preservation would then be:

> *If* $e \vdash t \downarrow_{cbv} c$, *then* $[\![e]\!] \vdash [\![t]\!] \downarrow_{cbv} [\![c]\!]$.

where the same semantics appears in the hypothesis and conclusion.

We could prove such a theorem, but...

That would be philosophically and technically unsatisfactory:

- the identity transformation would satisfy this statement, too!
- this statement does not explain in what way closure conversion is a useful step in a compiler pipeline. It should lead to a simpler calculus!

Which semantics for the target calculus?

Something is special about the target calculus,

or about the subset of the target calculus that we exploit.

Which semantics for the target calculus?

Something is special about the target calculus,

or about the subset of the target calculus that we exploit.

Every λ-abstraction is closed – that is the point of closure conversion!

Which semantics for the target calculus?

Something is special about the target calculus,

or about the subset of the target calculus that we exploit.

Every λ-abstraction is closed – that is the point of closure conversion!

We should therefore equip this calculus with a simplified semantics which does not involve any closures.

Which semantics for the target calculus?

Recall the standard big-step, environment-based semantics:

$$\text{EBigCbvVar}\quad \frac{e(x) = c}{e \vdash x \downarrow_{\text{cbv}} c} \qquad \text{EBigCbvLam}\quad \frac{fv(\lambda x.t) \subseteq dom(e)}{e \vdash \lambda x.t \downarrow_{\text{cbv}} \langle \lambda x.t \mid e \rangle} \qquad \text{EBigCbvApp}\quad \frac{\begin{array}{c} e \vdash t_1 \downarrow_{\text{cbv}} \langle \lambda x.u_1 \mid e' \rangle \\ e \vdash t_2 \downarrow_{\text{cbv}} c_2 \\ e'[x \mapsto c_2] \vdash u_1 \downarrow_{\text{cbv}} c \end{array}}{e \vdash t_1\ t_2 \downarrow_{\text{cbv}} c}$$

where $c ::= \langle \lambda x.t \mid e \rangle$ and $e ::= [] \mid e[x \mapsto c]$.

How can this semantics be simplified?

A semantics for the target calculus

No closures! Raw λ-abstractions instead. Think of them as code pointers.

MBigCbvVar
$$\frac{e(x) = c}{e \vdash x \Downarrow_{\text{cbv}} c}$$

MBigcbvLam
$$\frac{fv(\lambda x.t) \subseteq \emptyset}{e \vdash \lambda x.t \Downarrow_{\text{cbv}} \lambda x.t}$$

MBigcbvApp
$$\frac{e \vdash t_1 \Downarrow_{\text{cbv}} \lambda x.u_1 \qquad e \vdash t_2 \Downarrow_{\text{cbv}} c_2 \qquad [][x \mapsto c_2] \vdash u_1 \Downarrow_{\text{cbv}} c}{e \vdash t_1\ t_2 \Downarrow_{\text{cbv}} c}$$

where $c ::= \lambda x.t$ and $e ::= [] \mid e[x \mapsto c]$.

Such a semantics explains in what way the target calculus is simpler.

Exercise: extend this semantics with "let" constructs, pairs and projections. The solution is in `MetalBigStep`.

Towards a semantic preservation statement

The semantic preservation statement should be, roughly:

If $e \vdash t \downarrow_{cbv} c$, *then* $[\![e]\!] \vdash [\![t]\!] \Downarrow_{cbv} [\![c]\!]$.

What is missing still for this statement to make sense?

Towards a semantic preservation statement

The semantic preservation statement should be, roughly:

If $e \vdash t \downarrow_{cbv} c$, *then* $[\![e]\!] \vdash [\![t]\!] \Downarrow_{cbv} [\![c]\!]$.

What is missing still for this statement to make sense?

We must define the translation of environments $[\![e]\!]$ and closures $[\![c]\!]$.

Towards a semantic preservation statement

The semantic preservation statement should be, roughly:

If $e \vdash t \downarrow_{cbv} c$, *then* $[\![e]\!] \vdash [\![t]\!] \downdownarrows_{cbv} [\![c]\!]$.

What is missing still for this statement to make sense?

We must define the translation of environments $[\![e]\!]$ and closures $[\![c]\!]$.

An environment is naturally transformed pointwise:

$$\begin{array}{rcl} [\![[]]\!] & = & [] \\ [\![e[x \mapsto c]]\!] & = & e[x \mapsto [\![c]\!]] \end{array}$$

A closure is represented as...

Towards a semantic preservation statement

The semantic preservation statement should be, roughly:

If $e \vdash t \downarrow_{cbv} c$, *then* $[\![e]\!] \vdash [\![t]\!] \Downarrow_{cbv} [\![c]\!]$.

What is missing still for this statement to make sense?

We must define the translation of environments $[\![e]\!]$ and closures $[\![c]\!]$.

An environment is naturally transformed pointwise:

$$\begin{array}{rcl} [\![[]]\!] & = & [] \\ [\![e[x \mapsto c]]\!] & = & e[x \mapsto [\![c]\!]] \end{array}$$

A closure is represented as... a tuple:

$$\begin{array}{rcl} [\![\langle \lambda x.t \mid e \rangle]\!] & = & (\lambda(clo, x). \ldots,\ e(x_1), \ldots, e(x_n)) \\ & & \text{where } \{x_1, \ldots, x_n\} = fv(\lambda x.t) \end{array}$$

Towards a semantic preservation statement

The semantic preservation statement should be, roughly:

If $e \vdash t \downarrow_{cbv} c$, *then* $[\![e]\!] \vdash [\![t]\!] \Downarrow_{cbv} [\![c]\!]$.

What is missing still for this statement to make sense?

We must define the translation of environments $[\![e]\!]$ and closures $[\![c]\!]$.

An environment is naturally transformed pointwise:

$$
\begin{array}{rcl}
[\![[]]\!] & = & [] \\
[\![e[x \mapsto c]]\!] & = & e[x \mapsto [\![c]\!]]
\end{array}
$$

A closure is represented as... a tuple:

$$
\begin{array}{rcl}
[\![\langle \lambda x.t \mid e \rangle]\!] & = & (\lambda(clo, x)\ldots,\ e(x_1), \ldots, e(x_n)) \\
& & \text{where } \{x_1, \ldots, x_n\} = fv(\lambda x.t) \\
& & \text{and } x_1 < \ldots < x_n \text{ – for this to define a function}
\end{array}
$$

A semantic preservation statement

Semantic preservation is stated as follows:

Lemma (Semantic preservation)

Assume $e \vdash t \downarrow_{cbv} c$.
Then, $[\![e]\!] \vdash [\![t]\!] \downdownarrows_{cbv} [\![c]\!]$ *holds.*

Exercise (recommended): write a careful proof of this lemma, on paper.

A semantic preservation statement

In Coq, we use de Bruijn style, and prove:

Lemma (Semantic preservation)

Assume $e \vdash t \downarrow_{cbv} c$.
Then, $[\![e]\!] \vdash [\![t]\!]_n \Downarrow_{cbv} [\![c]\!]$ holds,
where n is the length of the environment e.

Proof.

By induction on the hypothesis.
See `ClosureConversion/semantic_preservation`. □

Things are easier if the transformation $[\![\cdot]\!]_{\cdot}$ takes two parameters t and n, where the free variables of t are supposed to be below n.

Exercise (tricky): define $[\![t]\!]_n$ and prove that if t has free variables below n then $[\![t]\!]_n$ has free variables below n. See `ClosureConversion/fv_cc`.

Caveats

Are we happy with this semantic preservation statement?

$$e \vdash t \downarrow_{cbv} c \textit{ implies } [\![e]\!] \vdash [\![t]\!] \Downarrow_{cbv} [\![c]\!].$$

Does it actually mean what we think it means?

What do we think it means? Perhaps this?

> *The transformed program "computes the same thing" as the source program.*

Or this?

> *The transformed program "behaves in the same way" as the source program.*

Caveat 1: is the translation nontrivial?

The trivial transformation defined by $[\![t]\!] = ()$
also satisfies this semantic preservation statement.

We should somehow check that our transformation is nontrivial.

Some concrete observation of program behavior must be preserved:

Caveat 1: is the translation nontrivial?

The trivial transformation defined by $[\![t]\!] = ()$
also satisfies this semantic preservation statement.

We should somehow check that our transformation is nontrivial.

Some concrete observation of program behavior must be preserved:

- If our calculi had primitive integers, we could check that $[\![i]\!]$ is i, so a program that computes an integer value is transformed to a program that computes the same value.

Caveat 1: is the translation nontrivial?

The trivial transformation defined by $[\![t]\!] = ()$
also satisfies this semantic preservation statement.

We should somehow check that our transformation is nontrivial.

Some concrete observation of program behavior must be preserved:

- If our calculi had primitive integers, we could check that $[\![i]\!]$ is i,
so a program that computes an integer value
is transformed to a program that computes the same value.
- We could also check that $[\![t]\!]$ terminates if and only if t terminates.
This brings us to the next slide...

Caveat 2: is nontermination preserved?

If t diverges (i.e., does not terminate), then "obviously" $[\![t]\!]$ diverges too, since "$[\![t]\!]$ computes the same thing as t".

Caveat 2: is nontermination preserved?

If t diverges (i.e., does not terminate), then "obviously" $[\![t]\!]$ diverges too, since "$[\![t]\!]$ computes the same thing as t".

However, we have not proved this fact:
it does not follow from our semantic preservation statement.

A transformation that maps certain divergent programs to ()
would still satisfy this statement!

Caveat 2: is nontermination preserved?

If t diverges (i.e., does not terminate), then "obviously" $[\![t]\!]$ diverges too, since "$[\![t]\!]$ computes the same thing as t".

However, we have not proved this fact:
it does not follow from our semantic preservation statement.

A transformation that maps certain divergent programs to ()
would still satisfy this statement!

To prove that nontermination is preserved, we could use:

- a small-step semantics, or
- a co-inductive nontermination judgement $e \vdash t \uparrow_{\text{cbv}}$.
  See Leroy and Grall (2007).

Caveat 3: forward versus backward preservation

We have proved:

> *Every behavior of the source program*
> *is also a behavior of the transformed program.*

Caveat 3: forward versus backward preservation

We have proved:

> *Every behavior of the source program*
> *is also a behavior of the transformed program.*

That is,

> *The behaviors of the source program*
> *form a subset of the behaviors of the transformed program.*

Caveat 3: forward versus backward preservation

We have proved:

> *Every behavior of the source program*
> *is also a behavior of the transformed program.*

That is,

> *The behaviors of the source program*
> *form a subset of the behaviors of the transformed program.*

So,

> *A program that prints "hello"*
> *can be transformed to*
> *a program that prints "hello" OR launches a missile.*

Caveat 3: forward versus backward preservation

We really need backward preservation:

> *The behaviors of the source program*
> *form a superset*
> *of the behaviors of the transformed program.*

If the target calculus has deterministic semantics,
then the transformed program has exactly one behavior,
so forward and backward preservation coincide.

If the target calculus has nondeterministic semantics,
we must prove backward preservation.

- This can be tricky.
- A semantics can be made artificially deterministic via an oracle.

Hobor, Appel, Zappa Nardelli,
Oracle Semantics for Concurrent Separation Logic, 2008.

Caveats – summary

In summary, always question what you have proved.

Even if your proofs are machine-checked,
your definitions and statements can be incorrect or deceiving.

Minimal environments

My pencil-and-paper definition of closure conversion allocates closures with minimal environments:

$$
\begin{array}{ll}
[\![\lambda x.t]\!] = \text{let } \mathit{code} = & \text{where } \{x_1,\ldots,x_n\} = \mathit{fv}(\lambda x.t) \\
\quad \lambda(\mathit{clo}, x). & \\
\quad\quad \text{let } x_i = \pi_i\ \mathit{clo} \text{ in} & \text{for each } i \in \{1,\ldots,n\} \\
\quad\quad [\![t]\!] & \\
\quad \text{in} & \\
\quad (\mathit{code}, x_1, \ldots, x_n) &
\end{array}
$$

In contrast, for greater simplicity, my Coq definition in `ClosureConversion` allocates closures with full environments.

Full environments lead to larger closures and space leaks, so in practice are not viable.

Recursive functions

How should recursive function definitions be transformed?

$$
\begin{array}{ll}
[\![\lambda x.t]\!] = \text{let } code = & \text{where } \{x_1, \ldots, x_n\} = fv(\lambda x.t) \\
\quad \lambda(clo, x). & \\
\quad\quad \text{let } x_i = \pi_i \; clo \text{ in} & \text{for each } i \in \{1, \ldots, n\} \\
\quad\quad [\![t]\!] & \\
\quad \text{in} & \\
\quad (code, x_1, \ldots, x_n) &
\end{array}
$$

$$[\![\mu f.\lambda x.t]\!] = ?$$

$$
\begin{array}{l}
[\![t_1 \; t_2]\!] = \text{let } clo = [\![t_1]\!] \text{ in} \\
\quad \text{let } (code, \ldots) = clo \text{ in} \\
\quad code \; (clo, [\![t_2]\!])
\end{array}
$$

The transformation of function calls cannot be modified: the caller does not (want to) know whether the callee is a recursive function.

In other words, we must maintain a uniform calling convention.

Recursive functions

We must arrange for the variable f to be bound to an appropriate value:

$$
\begin{array}{ll}
[\![\mu f.\lambda x.t]\!] = \text{let } code = & \text{where } \{x_1, \ldots, x_n\} = fv(\mu f.\lambda x.t) \\
\qquad \lambda(clo, x). & \\
\qquad\quad \text{let } f = clo \text{ in} & \\
\qquad\quad \text{let } x_i = \pi_i\ clo \text{ in} & \text{for each } i \in \{1, \ldots, n\} \\
\qquad\quad [\![t]\!] & \\
\quad \text{in} & \\
\quad (code, x_1, \ldots, x_n) &
\end{array}
$$

The closure *clo* happens to be such a value.

Mutually recursive functions

What about mutually recursive function definitions?

Suppose the source calculus has mutually recursive function definitions:

$$t ::= \ldots \mid \text{let rec } f_1 = \lambda x_1.t_1 \text{ and } f_2 = \lambda x_2.t_2 \text{ in } t$$

Suppose the target calculus has mutually recursive value definitions:

$$t ::= \ldots \mid \text{let rec } f_1 = v_1 \text{ and } f_2 = v_2 \text{ in } t$$

Exercise: propose a (deterministic, call-by-value) operational semantics with support for these constructs. A syntactic contractiveness condition must be imposed to rule out nonsensical definitions such as `let rec x = x`.

Mutually recursive functions

It is then easy to define a transformation of recursive functions:

$$
\begin{array}{l}
[\![\text{let rec } f_1 = \lambda x_1.t_1 \\
\quad\text{and } f_2 = \lambda x_2.t_2 \\
\qquad\qquad\text{in } u]\!] = \begin{array}[t]{l}\text{let rec } f_1 = [\![\lambda x_1.t_1]\!] \\ \text{and } f_2 = [\![\lambda x_2.t_2]\!] \\ \text{in } [\![u]\!]\end{array}
\end{array}
$$

This transformation is correct, albeit slightly inefficient.

We get two closures, each of which contains
a pointer to itself and
a pointer to the other closure.

Indeed, f_1 and f_2 can be free in $\lambda x_1.t_1$ and $\lambda x_2.t_2$.
If so, the tuples $[\![\lambda x_1.t_1]\!]$ and $[\![\lambda x_2.t_2]\!]$ have slots for f_1 and f_2.

Mutually recursive functions

A slightly more efficient transformation is as follows:

$$
\begin{array}{l}
[\![\text{let rec } f_1 = \lambda x_1.t_1 \\
\quad \text{and } f_2 = \lambda x_2.t_2 \\
\qquad\qquad \text{in } u]\!] = \text{let rec } f_1 = [\![\mu f_1.\lambda x_1.t_1]\!] \\
\qquad\qquad\qquad\quad\;\; \text{and } f_2 = [\![\mu f_2.\lambda x_2.t_2]\!] \\
\qquad\qquad\qquad\quad\;\; \text{in } [\![u]\!]
\end{array}
$$

We get two closures, each of which points to the other.

The redundant pointer from each closure to itself has been removed.

Complexity and cost model

Closure conversion leads to the following cost model:

- Evaluating a variable costs $O(1)$.
- Evaluating a λ-abstraction costs $O(n)$, where n is the number of its free variables.
- Evaluating an application costs $O(1)$.

n can be considered $O(1)$, as it depends only on the program's text, not on the input data.

Understanding programs through closure conversion

People sometimes use first-class function in somewhat mysterious ways.

Closure conversion can help understand what is going on.

Let us look at a little example: difference lists.

This example also illustrates type-preserving closure conversion in OCaml.

A fringe computation

Suppose we have a type of trees with integer-labeled leaves:

```
type tree =
  | Leaf of int
  | Node of tree * tree
```

Suppose we wish to construct a tree's fringe, a sequence of integers:

```
let rec fringe (t : tree) : int list =
  match t with
  | Leaf i        -> [ i ]
  | Node (t1, t2) -> fringe t1 @ fringe t2
```

What do you think?

A fringe computation

Suppose we have a type of trees with integer-labeled leaves:

```
type tree =
  | Leaf of int
  | Node of tree * tree
```

Suppose we wish to construct a tree's fringe, a sequence of integers:

```
let rec fringe (t : tree) : int list =
  match t with
  | Leaf i         -> [ i ]
  | Node (t1, t2) -> fringe t1 @ fringe t2
```

What do you think?

This code is inefficient. It worst-case time complexity is quadratic.

Difference lists

To remedy this, people sometimes use difference lists (Hughes, 1984).

```
type 'a diff =
  'a list -> 'a list

let singleton (x : 'a) : 'a diff =
  fun xs -> x :: xs
let concat (xs : 'a diff) (ys : 'a diff) : 'a diff =
  fun zs -> xs (ys zs)
```

The idea is to represent a list `xs` as a function that maps `ys` to `xs @ ys`.

`singleton` is "cons", and `concat` is function composition!

They both have time complexity $O(1)$.

Difference lists

With difference lists, the fringe computation is as follows:

```
let rec fringe_ (t : tree) : int diff =
  match t with
  | Leaf i         -> singleton i
  | Node (t1, t2) -> concat (fringe_ t1) (fringe_ t2)

let fringe t =
  fringe_ t []
```

Is this code efficient? What is its time complexity?

Difference lists

With difference lists, the fringe computation is as follows:

```
let rec fringe_ (t : tree) : int diff =
  match t with
  | Leaf i         -> singleton i
  | Node (t1, t2) -> concat (fringe_ t1) (fringe_ t2)

let fringe t =
  fringe_ t []
```

Is this code efficient? What is its time complexity?

The application `fringe_t t` costs $O(n)$,
and the application of its result to the empty list `[]` costs $O(n)$ as well.

That is somewhat nonobvious, though. Closure conversion to the rescue!

A type of all closures

A type of closures can be defined in OCaml as follows:

```
type ('a, 'b) closure =
  | Clo:
      ('a * 'e -> 'b)    (* A (closed) function...       *)
          * 'e           (* ...and its environment...    *)
    -> ('a, 'b) closure  (* ...together form a closure. *)
```

This is an existential type: (α, β) *closure* $\simeq \exists\epsilon.((\alpha \times \epsilon) \rightarrow \beta) \times \epsilon$.

Here, the closure and its environment form distinct memory blocks; this is slightly different from what was shown earlier today.

To find out about type-preserving closure conversion, see Minamide *et al.* (1996) and my slides (2009).

Well-typed closure invocation

Because all closures have the same type, the code that invokes a closure can be written once and for all, and is polymorphic.

```
let apply (f : ('a, 'b) closure) (x : 'a) : 'b =
  let Clo (code, env) = f in
  code (x, env)
```

`env` has unknown type, yet the call `code (x, env)` is definitely safe.

Difference lists, closure-converted

Let us manually apply closure conversion to difference lists.

A difference list is a closure:

```
type 'a diff =
  ('a list, 'a list) closure
```

`singleton` and `concat` allocate and return a closure:

```
let singleton_code =
  fun (xs, x) -> x :: xs
let singleton (x : 'a) : 'a diff =
  Clo (singleton_code, x)

let concat_code =
  fun (zs, (xs, ys)) -> apply xs (apply ys zs)
let concat (xs : 'a diff) (ys : 'a diff) : 'a diff =
  Clo (concat_code, (xs, ys))
```

The closed functions `singleton_code` and `concat_code` are hoisted out.

Difference lists, closure-converted

The difference-list-based fringe computation, closure-converted:

```
let rec fringe_ (t : tree) : int diff =
  match t with
  | Leaf i         -> singleton i
  | Node (t1, t2) -> concat (fringe_ t1) (fringe_ t2)

let fringe t =
  apply (fringe_ t) []
```

It should be clear that `fringe_ t` builds a tree of closures,
whose shape is identical to the shape of the tree `t`.

Then, the call `apply _ []` interprets this tree of closures.

Is this efficient?

Difference lists, closure-converted

The difference-list-based fringe computation, closure-converted:

```
let rec fringe_ (t : tree) : int diff =
  match t with
  | Leaf i         -> singleton i
  | Node (t1, t2) -> concat (fringe_ t1) (fringe_ t2)

let fringe t =
  apply (fringe_ t) []
```

It should be clear that `fringe_ t` builds a tree of closures, whose shape is identical to the shape of the tree `t`.

Then, the call `apply _ []` interprets this tree of closures.

Is this efficient?

Each of the two phases costs $O(n)$. Asymptotic complexity is good.

Yet, the first phase, a tree copy, is useless. The constant factor is not great.

Exercise: write `fringe` so that cost is $O(n)$ and no tree copy is carried out.

Some history

Closures were used in interpreters in the 1960s and 1970s.

Landin, The Mechanical Evaluation of Expressions, 1964.

Sussman and Steele, SCHEME: an interpreter for extended lambda-calculus, 1975.

Closure conversion in a compiler (first?) appeared in Rabbit.

Steele, RABBIT: a compiler for SCHEME, 1978.

Further improvements

Naïve closure conversion can produce inefficient code.

Selective closure conversion applies when the environment would have zero slots, and avoids building a closure in that case.

Lightweight closure conversion applies when a value that should be stored in the environment happens to be available at every call site. Then, instead of being stored, this value becomes an extra argument.

Both ideas involve a modified closure invocation protocol,
therefore require an agreement between callers and callees,
therefore require a control flow analysis:
one must know which closures may be invoked where.

Steckler and Wand, Selective and lightweight closure conversion, 1994.

Cejtin *et al.*, Flow-directed closure conversion for typed languages, 2000.
— actually, defunctionalization

Further improvements

Applications of (curried) multi-parameter functions
to multiple actual arguments should be identified and optimized.

- e.g., `map f xs` should not be compiled as `(map f) xs`,
  where `(map f)` allocates and returns a closure!

This is done in OCaml and in CakeML,
where this transformation is verified.

Owens *et al.*, Verifying Efficient Function Calls in CakeML, 2017.

Trivia: closure and mutable variables

In ML, F#, Java, ..., closures capture immutable variables only.

In certain “interesting” programming languages, closures can refer to mutable variables, too. E.g., in JavaScript:

```
var messages = ["Wow!", "Hi!", "Closures are fun!"];
for (var i = 0; i < messages.length; i++) {
  setTimeout(function () {
    say(messages[i]);
  }, i * 1500);
}
```

This program, of course, ...

Trivia: closure and mutable variables

In ML, F#, Java, ..., closures capture immutable variables only.

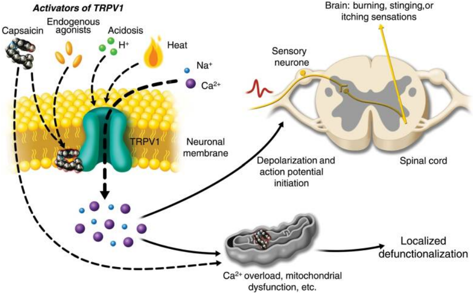
In certain "interesting" programming languages, closures can refer to mutable variables, too. E.g., in JavaScript:

```
var messages = ["Wow!", "Hi!", "Closures are fun!"];
for (var i = 0; i < messages.length; i++) {
  setTimeout(function () {
    say(messages[i]);
  }, i * 1500);
}
```

This program, of course, ... prints `undefined` three times.

See Orendorff's blog post for details.

Activation of TRPV1 by capsaicin



Activation of TRPV1 by capsaicin results in sensory neuronal depolarization, and can induce local sensitization to activation by heat, acidosis, and endogenous agonists. Topical exposure to capsaicin leads to the sensations of heat, burning, stinging, or itching. High concentrations of capsaicin or repeated applications can produce a persistent local effect on cutaneous nociceptors, which is best described as **defunctionalization** and constituted by reduced spontaneous activity and a loss of responsiveness to a wide range of sensory stimuli.

Defunctionalization

Defun°, like closure conversion, aims to eliminate first-class functions.

Defun° differs in the representation of closures:

- instead of a code pointer and an environment,
- use a tag and an environment.

Thus, instead of (closed) first-class functions,
the target language must have algebraic data types.

Functions become data!

Definition of defunctionalization

Assume that every λ-abstraction is decorated with a distinct label C.

$$
\begin{aligned}
[\![x]\!] &= x \\
[\![\lambda^C x.t]\!] &= C(x_1,\ldots,x_n) \qquad\qquad \text{where } \{x_1,\ldots,x_n\} = fv(\lambda x.t) \\
[\![t_1\ t_2]\!] &= apply\ ([\![t_1]\!],[\![t_2]\!])
\end{aligned}
$$

Simple, eh?

There remains to define *apply*.

Definition of defunctionalization

The toplevel function *apply* interprets a closure as a function.

The whole transformed program is placed in the scope of this definition:

let rec *apply* $(\mathit{this}, \mathit{that}) =$
  match *this* with
  ...
  $\mid C(x_1, \ldots, x_n) \rightarrow$ for each function $\lambda^C x.t$ in the source
    let $x = \mathit{that}$ in
    $[\![t]\!]$
  ...

apply must be recursively defined, as $[\![t]\!]$ can refer to *apply*.

Soundness of defunctionalization

Is defunctionalization sound, that is, semantics-preserving?

Of course it is!

For a (paper) proof in an untyped setting, see Pottier and Gauthier (2006).

Difference lists

Recall difference lists:

```
type 'a diff =
  'a list -> 'a list

let singleton (x : 'a) : 'a diff =
  fun xs -> x :: xs
let concat (xs : 'a diff) (ys : 'a diff) : 'a diff =
  fun zs -> xs (ys zs)
```

As an illustration, let us manually defunctionalize this code.

There are two first-class functions of interest in this code.

Difference lists, defunctionalized

Thus, we introduce an algebraic data type with two data constructors:

```
type 'a diff =
  | Singleton of 'a
  | Concat of 'a diff * 'a diff
let singleton (x : 'a) : 'a diff =
  Singleton x
let concat (xs : 'a diff) (ys : 'a diff) : 'a diff =
  Concat (xs, ys)
```

The functions `singleton` and `concat` just build a closure.

There remains to define `apply`.

Difference lists, defunctionalized

`apply` applies a closure `this` to an argument `that`.

```
let rec apply (this : 'a diff) (that : 'a list) : 'a list =
  match this with
  | Singleton x ->
      let xs = that in
      x :: xs
  | Concat (xs, ys) ->
      let zs = that in
      apply xs (apply ys zs)
```

Difference lists, defunctionalized

The fringe computation is the same as in the closure-converted version:

```
let rec fringe_ (t : tree) : int diff =
  match t with
  | Leaf i          -> singleton i
  | Node (t1, t2) -> concat (fringe_ t1) (fringe_ t2)

let fringe t =
  apply (fringe_ t) []
```

The types `tree` and `int diff` are isomorphic!

It is clear (again) that `fringe_` is just a tree copy.

Sets as characteristic functions

Let us use defunctionalization to investigate another slightly mysterious piece of code, namely sets implemented as functions.

Sets as characteristic functions

A set (of integers) can be represented by its characteristic function:

```
type set =
  int -> bool
let empty : set =
  fun y -> false
let singleton (x : int) : set =
  fun y -> y = x
let union (s1 : set) (s2 : set) : set =
  fun y -> s1 y || s2 y
let mem (x : int) (s : set) : bool =
  s x
```

This works. But is it smart... or is it not?

Sets as characteristic functions

A set (of integers) can be represented by its characteristic function:

```
type set =
  int -> bool
let empty : set =
  fun y -> false
let singleton (x : int) : set =
  fun y -> y = x
let union (s1 : set) (s2 : set) : set =
  fun y -> s1 y || s2 y
let mem (x : int) (s : set) : bool =
  s x
```

This works. But is it smart... or is it not?

`empty`, `singleton`, `union` have time complexity $O(1)$. What about `mem`?

Sets as characteristic functions

A set (of integers) can be represented by its characteristic function:

```
type set =
  int -> bool
let empty : set =
  fun y -> false
let singleton (x : int) : set =
  fun y -> y = x
let union (s1 : set) (s2 : set) : set =
  fun y -> s1 y || s2 y
let mem (x : int) (s : set) : bool =
  s x
```

This works. But is it smart... or is it not?

`empty`, `singleton`, `union` have time complexity $O(1)$. What about `mem`?

Answering requires understanding the structure of the closures that we build.

Sets as characteristic functions

There are three places where a closure of type `set` is built:

```
let empty : set =
  fun y -> false
let singleton (x : int) : set =
  fun y -> y = x
let union (s1 : set) (s2 : set) : set =
  fun y -> s1 y || s2 y
```

What fields do these closures carry?

- in `empty`, no fields;
- in `singleton`, one field of type `int` corresponding to `x`;
- in `union`, two fields of type `set` corresponding to `s1` and `s2`.

Let us give these three kinds of closures three distinct labels, say `E`, `S`, `U`.

Sets as characteristic functions, defunctionalized

A “set” is really a closure of one of these three kinds.

Through defunctionalization, `set` becomes an algebraic data type:

```
type set =
  | E
  | S of int
  | U of set * set
```

The three constructor functions become:

```
let empty : set =                          E
let singleton (x : int) : set =            S (x)
let union (s1 : set) (s2 : set) : set = U (s1, s2)
```

What is this data type?

Sets as characteristic functions, defunctionalized

A “set” is really a closure of one of these three kinds.

Through defunctionalization, `set` becomes an algebraic data type:

```
type set =
  | E
  | S of int
  | U of set * set
```

The three constructor functions become:

```
let empty : set =                            E
let singleton (x : int) : set =              S (x)
let union (s1 : set) (s2 : set) : set = U (s1, s2)
```

What is this data type?

A data type of trees with leaves `E` et `S` and binary nodes `U`.

Sets as characteristic functions, defunctionalized

`apply` interprets a `set` as a characteristic function of type `int -> bool`.

```
let rec apply (s : set) (y : int) : bool =
  match s with
  | E          -> false
  | S (x)      -> y = x
  | U (s1, s2) -> apply s1 y || apply s2 y
```

The membership test becomes:

```
let mem (x : int) (s : set) : bool =
  apply s x
```

Smart?...

Defun° helps see that a set is represented as an unbalanced tree.

`mem s x` traverses all of the tree `s` in search of `x`.

`mem` has time complexity $O(n)$. Inefficient!

Understanding closure conversion and/or defun° helps analyze this code.

Type-preserving defunctionalization

We have seen earlier that closure conversion can be type-preserving. This requires existential types in the target calculus.

Can defunctionalization be made type-preserving?

Yes, it can. If the source calculus has polymorphism, this requires generalized algebraic data types (GADTs) in the target calculus.

To see why, try translating this:

```
map (fun x -> not x) (map (fun x -> x + 1) xs)
```

Pottier and Gauthier,
Polymorphic typed defunctionalization and concretization, 2006.

Some history

Defunctionalization is applied by Reynolds to an interpreter.

More about this next week!

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

Reynolds, Definitional interpreters revisited, 1998.

Defunctionalization is used in some compilers, e.g., MLton.

They use data flow analysis to create multiple specialized `apply` functions, which dispatch on fewer cases.

Cejtin, Jagannathan, Weeks, Flow-directed Closure Conversion for Typed Languages, 2000.

Compiling functions as objects

On the surface, Scala has functions and objects, but functions are sugar.

The function type `T => R` is sugar for `Function1[T, R]`.

```
trait Function1[-T, +R] { // Defined in the base library.
  def apply(v: T): R       // Any object with an apply method
}                          // is a "function"!
```

An anonymous function:

```
(x: Int) => x + y
```

is sugar for an object creation expression:

```
new Function1[Int, Int] {
  def apply(x: Int): Int = x + y
}
```

Functions (w/ free variables) are translated to objects (w/ free variables).

Compiling objects with free variables

This object creation expression has a free variable `y`:

```
new Function1[Int, Int] {
  def apply(x: Int): Int = x + y
}
```

It can be viewed as sugar for

```
new C (y)
```

where `C` is a unique name. The class `C` is defined at the top level:

```
class C (y: Int) {
  def apply(x: Int): Int = x + y
}
```

Objects (w/ free variables) are translated to parameterized classes.

This is a modular form of defunctionalization.

Compiling parameterized classes

A parameterized class:

```
class C (y: Int) {
  def apply(x: Int): Int = x + y
}
```

is sugar for a class with an explicit field and a constructor:

```
class C {
  var y: Int = _
  def this (y: Int) = {
    this()
    this.y = y
  }
  def apply(x: Int): Int = x + y
}
```

Parameterized classes are translated down to ordinary classes.

Functions in Scala

Thus, functions in Scala are represented by heap-allocated closures, with one field per free variable, just as in OCaml.

λ-lifting

First-class functions (with free variables) can also be be compiled down to a combination of closed (toplevel) functions and partial applications.

This is known as λ-lifting.

Johnsson, Lambda lifting: transforming programs to recursive equations, 1985.

λ-lifting

`eval` evaluates a polynomial `cs` at a point `x`.

```
let eval (cs : int list) (x : int) : int =
  let cons (c : int) (a : int -> int) =
    let aux x_n = c * x_n + a (x * x_n) in
    aux
  and null x_n =
    0
  in foldr cons null cs 1
```

`aux` has free variables `c`, `a`, `x`. `cons` has free variable `x`.

Step 1: make them parameters.

λ-lifting

Every function in the resulting code is closed.

```
let eval cs x =
  let cons x c a =
    let aux c a x x_n = c * x_n + a (x * x_n) in
    aux c a x
  and null x_n =
    0
  in foldr (cons x) null cs 1
```

Step 2: hoist every function to the top level.

λ-lifting

We get a group of toplevel functions, also known as supercombinators:

```
let aux c a x x_n =
  c * x_n + a (x * x_n)
let cons x c a =
  aux c a x
let null x_n =
  0
let eval cs x =
  foldr (cons x) null cs 1
```

This code contains partial applications, though. (Find them!)

λ-lifting

λ-lifting was popular in the 1980s because people were interested in abstract machines or "graph reduction" systems that dealt with partial applications directly.

Peyton Jones, The implementation of functional programming languages, 1987.

Today, these ideas are largely obsolete.

Compiling to SKI combinators

An idea from the 1950s (Curry and Feys, 1958).

The combinators S, K, I are defined as follows:

$$
\begin{array}{rcl}
S\ f\ g\ x & = & f\ x\ (g\ x) \\
K\ x\ y & = & x \\
I\ x & = & x
\end{array}
$$

What is special about them?

Compiling to SKI combinators

An idea from the 1950s (Curry and Feys, 1958).

The combinators S, K, I are defined as follows:

$$\begin{array}{rcl} S\ f\ g\ x & = & f\ x\ (g\ x) \\ K\ x\ y & = & x \\ I\ x & = & x \end{array}$$

What is special about them?

Every λ-term can be compiled to code
that involves just S, K, I and applications.

No λ-abstractions, no variables!

Compiling to SKI combinators

Apply the following rules, beginning with innermost λ-abstractions:

$$
\begin{array}{llll}
\text{replace } \lambda x.x & \text{with} & I & \\
\text{replace } \lambda x.t & \text{with} & K\ t & \text{if } x \notin fv(t) \\
\text{replace } \lambda x.(t_1\ t_2) & \text{with} & S\ (\lambda x.t_1)\ (\lambda x.t_2) &
\end{array}
$$

For instance,

$$
\begin{array}{ll}
 & (\lambda x.+\ x\ x)\ 5 \\
\to & S\ (\lambda x.+\ x)\ (\lambda x.x)\ 5 \\
\to & S\ (S\ (\lambda x.+)\ (\lambda x.x))\ (\lambda x.x)\ 5 \\
\to & S\ (S\ (K\ +)\ (\lambda x.x))\ (\lambda x.x)\ 5 \\
\to & S\ (S\ (K\ +)\ I)\ (\lambda x.x)\ 5 \\
\to & S\ (S\ (K\ +)\ I)\ I\ 5
\end{array}
$$

Recursion can be dealt with by adding the combinator Y.

Compiling to SKI combinators

S, K, I can be viewed as the instruction set of an abstract machine.

This idea was used to compile SASL and Miranda (Turner, 1976–79).

There were plans in the 1980s to do this in hardware!

Peyton Jones, The implementation of functional programming languages, 1987.

Of course, it is much more efficient to compile down to machine code for a standard (von Neumann) processor, which is highly optimized.

A few things to remember

- First-class functions are a key feature of modern prog. languages.
- Closure conversion and defunctionalization help understand them.
- Like objects, closures bundle code and data
  — "behavior" and "state", in object-oriented speak.
- We have encountered a semantic preservation statement and discussed its exact meaning. Always question what you have proved!