CS410/510
Compiling Advanced Languages
Spr2000 Lecture Notes
Towards Abstract Machines II

Automating the conversion to Pretty-normal form

A naive, local translation algorithm is easy. This version does left-to-right translation.

```
open Pnf
let rec translate (e: Tcore.exp) : Pnf.exp =
  match e with
    Tcore.Var x -> Triv(Var x)
  | Tcore.Int i -> Triv(Int i)
  | Tcore.Abs(v,b) -> Triv(Abs(v,translate b))
  | Tcore.App(Tcore.Abs(v,b),a) ->
      Let(v,translate a, translate b)
  | Tcore.App(e1,e2) =>
      let x1 = gensym() and x2 = gensym() in
      Let(x1, translate e1,
              Let(x2, translate e2,
                      App(Var x1,Var x2)))
  | Tcore.Record(lel) =>
      let rec f (lel,xl) =
        match lel with
          (l,e)::ler ->
              let x = gensym() in
                   Let(x,translate e,
                     f(ler,xl@[(l,Var x)]))
        | [] -> Record(xl) in
      f(lel,[])
  | ...
```


Smarter Translations

The naive algorithm creates many needless names. A better algorithm would not name expressions that are already trivial. Here's a try:

```
open Pnf
let rec translate (e: Tcore.exp) : Pnf.exp =
  match e with
    ...
  | Tcore.App(e1,e2) ->
     let tr1 = translate e1 and tr2 = translate e2 in
     match (tr1,tr2) of
          (Triv t1,Triv t2) -> App(t1,t2)
        | (Triv t1, _) =>
            let val x2 = gensym()
            in Let(x2,tr2, App(t1,Var x2))
        | (_,Triv t2) =>
            let val x1 = gensym()
            in Let(x1,tr1, App(Var x1,t2))
        | (_,_) =>
            let val x1 = gensym() and x2 = gensym()
            in Let(x1,tr1,
                      Let(x2,tr2,
                             App(Var x1,Var x2)))
```

This approach scales badly! But how can we abstract over the process of "naming where necessary?"

A neater, higher-order approach

Idea: pass a function that says "what to do" with the trivial expression.

```
open Pnf
let rec translate (e:Tcore.exp) : Pnf.exp =
  match e with
    Tcore.Var v -> Triv(Var v)
  | ...
  | Tcore.App(e1,e2) ->
        trivialize e1 (fun t1 ->
          trivialize e2 (fun t2 ->
            App(t1,t2)))
  | ...

and trivialize
      (e:Tcore.exp)
      (k:Pnf.trexp -> Pnf.exp) : Pnf.exp =
  match e with
    Tcore.Var v -> k(Var v)
  | Tcore.Int i -> k(Int i)
  | Tcore.Abs(v,t,e) -> k(Abs(v,t,translate e))
  | _ ->
     let x = gensym() in
     Let(x,translate e,k(Var x))
```

Tail-calls

Consider the following program to compute the length of a list:

```
let rec len = function
    (h::t) -> len t + 1
  | [] -> 0 in
print_int (len [1;2])
end
```

Function `len` is recursive, but not tail-recursive. But we can convert `len` into a tail-recursive function, as follows:

```
let rec len' = function
   (h::t,k) -> len'(t, fun i -> k (i+1))
 | ([],k) -> k 0 in
len' ([1;2], fun x -> print_int x)
end
```

This rather odd code was constructed by giving `len'` an additional argument, `k`, of type `int`→`unit`. Instead of returning its "result" value, `len'` passes it to `k`.

Notice that **every** call in `len'` is a **tail-call**. Note too that `len'` only returns after `print_int` is invoked; in essence, it needn't return at all. This is because `k` is serving the same role as a return address: saying "what to do next."

Comparison of Computations

```
print(len [1;2]) →
print(len [2] + 1) →
print((len [] + 1) + 1) →
print((0 + 1) + 1) →
print(1 + 1) →
print 2
```

$$\texttt{len'([1;2],}\lambda\texttt{x.print x)} \to$$
$$\texttt{len'([2],}\lambda\texttt{i}_1\texttt{.(}\lambda\texttt{x.print x)(i}_1\texttt{ + 1))} \to$$
$$\texttt{len'([],}\lambda\texttt{i}_2\texttt{.(}\lambda\texttt{i}_1\texttt{.(}\lambda\texttt{x.print x)(i}_1\texttt{ + 1))(i}_2\texttt{ + 1))} \to$$
$$\texttt{(}\lambda\texttt{i}_2\texttt{.(}\lambda\texttt{i}_1\texttt{.(}\lambda\texttt{x.print x)(i}_1\texttt{ + 1))(i}_2\texttt{ + 1)) 0} \to$$
$$\texttt{(}\lambda\texttt{i}_1\texttt{.(}\lambda\texttt{x.print x)(i}_1\texttt{ + 1))(0 + 1)} \to$$
$$\texttt{(}\lambda\texttt{i}_1\texttt{.(}\lambda\texttt{x.print x)(i}_1\texttt{ + 1)) 1} \to$$
$$\texttt{(}\lambda\texttt{x.print x)(1 + 1)} \to$$
$$\texttt{(}\lambda\texttt{x.print x) 2} \to$$
$$\texttt{print 2}$$

Continuation-passing Style

Functions like `k` are (loosely) called **continuations** and programs written using them are said to be in (a form of) **continuation-passing style (CPS)**.

We may choose to write programs explicitly in CPS because it makes it easy to express a particular algorithm (for example, the Tcore to Pnf translation from earlier) or because it clarifies the control structure of the program (as we will show later) In general, we may want to write only parts of the program, or individual functions, in this style.

Note that CPS is just a subset of ordinary λ-calculus, that happens to make heavy use of the (existing) enormous power of λ-abstractions.

Remarkably, we can also systematically convert **any** λ-calculus program into an equivalent CPS program. (Details omitted.)

But first, why "continuation"?

Continuations

Broadly speaking, a **continuation** is a representation of a dynamic program point, i.e., the state of the program at some particular time during execution.

A program's state at a particular dynamic point must contain exactly the information needed to "continue" execution of the program from that point until the program completes.

Continuations are needed in denotational semantics to give an (abstract) meaning to programs in languages that support control flow operators. They can also be used to describe the operational semantics, i.e., execution behavior, of programs at various levels of abstraction.

If we are describing execution in terms of a low-level machine, we can think of a continuation for a program point as the machine state at that point: the values of the pc, the sp, and other registers.

If we are describing execution in a an interpreter implemented in a functional language, we can think of a continuation for a program point as a **function**, which, given the **value** of the expression being computed at that point, returns the result of the entire program. (Continuations are also usually treated as functions of this type in denotational semantics.)

Implicit and Explicit Continuations

Our definition of continuations refers to a description of **internal** state. These continuations are **implicit** in our execution model of the program. For example, the functions just referred to exist in the **interpreter** for a program, not in the program itself.

The function `k` we introduced in `len'` is **not** a continuation in this sense; it appears explicitly in the program. But we can justify calling it a continuation because it **does** correspond to an (implicit) continuation in the original program `len`!

In fact, we can view the process of rewriting `len` into `len'` as one of **making internal continuations explicit**. Systematic rewriting of this kind may be a useful step during language compilation, because programs in CPS may be easier to optimize or process than ordinary "direct-style" programs. Orbit (a Scheme compiler) and SML/NJ are examples of FL compilers that use this technique.

In the context of such systems, saying a program is in "CPS" also means that all function and constructor arguments are "trivial" expressions, as in Pretty-Normal Form.

Meanwhile, we'll continue to use the term "explicit continuation" to refer informally to functions like `k`, even when they are not generated systematically from a direct-style program.

First-class Continuations

Some languages (including Scheme and SML/NJ, but not Caml) permit us to "capture" the current dynamic program state (implicit continuation) into an explicit data object. This is done using a special expression form such as `callcc` (for "call-with-current-continuation"), which binds a representation of the current state to a variable.

These so-called **first-class continuations** behave much like functions (indeed in Scheme they **are** functions): invoking one restores the program's state to the time of capture. This is very handy for implementing user-defined control operators such as exceptions and coroutines.

To the programmer, a first-class continuation is a black box (just like an ordinary function); it can be passed around and invoked, but its internals cannot be inspected. The actual representation, and the cost of performing captures and resets of the current state, depend on the compiler's implementation technology.

Note that C has a similar mechanism in the form of `setjmp/longjmp`; the `jmp_buf` effectively contains a concrete representation for a continuation! However, it is not a completely first-class continuation, since it ceases to be valid if the user program returns out of the routine in which the `setjmp` was established.

Continuations at Many Levels

In summary, the term "continuation" can be used in many different, though related, ways, which we should try to keep straight:

• Abstract continuation functions may be used to describe the semantics of a language.

• We can program in an explicit "continuation-passing" style, as exemplified by `len'`.

• We can convert ordinary ("direct style") programs into CPS representations in order to make them more "machine-like" and/or easier to optimize.

• If our language permits, we can "capture" or "reify" program state (the current implicit continuation) into an explicit data object (a "first-class continuation") which can later be used to restore that state.

A CPS Interpreter

As an application of CPS, consider a modification of our PNF interpreter that passes explicit continuations:

```
let rec eval (env:value Env.env)
             (e:exp)
             (k:value -> value) : value =
  let rec trivial (e:trexp) : value = ...
  and eval0 (e:exp) (k:value->value) : value =
    match e with
      Triv tre -> k (trivial tre)
    | Let(x,a,b) ->
       let k' = fun v -> eval (Env.extend env x v) b k in
       eval0 a k'
    | App(tre1,tre2) ->
        apply (trivial tre1) (trivial tre2) k
    | ...
    | Variant(l,tre) -> k (Vvariant(l,trivial tre))
    | ...  in
  eval0 e k

and apply (v1:value) (v2:value) (k:value->value) : value =
  match v1 with
    Vclos (id,exp,env) ->
        eval (Env.extend (!env) id v2) exp k
  | Vfn f -> k (f v2)

let idk = fun v -> v

let evaluate (e:exp) : value = eval initial_env e idk
```

From continuations to stacks

This interpreter is **completely** tail-recursive, so it can be implemented with a simple loop, just like a "real" instruction dispatch unit in hardware. The only non-trivial continuations are built in `Let`, to record where to assign the result of a nested non-tail computation.

If we analyze the forms of possible continuations, we see there are only two: the initial identity continuation `idk`, defined as

```
fun v -> v
```

and the continuations built in `Let`, which are always of the same form

```
fun v -> eval (Env.extend env x v) b k
```

where only the free variables `x`,`b`, `env` and `k` can differ. This suggests that we can replace the first-class closure functions with simple, efficient data structure (namely a stack of free variable records) and an explicit dispatch function:

```
type stack = Empty
           | Frame of var * exp * value Env.env * stack

let return k v =
    match k with
      Empty -> v
    | Frame(x,b,env,k) -> eval (Env.extend env x v) b k
```


Stack-based Interpreter

```
let rec eval (env:value Env.env)
             (e:exp)
             (k:stack) : value =
    let rec trivial (e:trexp) : value = ...
    and eval0 (e:exp) (k:stack) : value =
      match e with
        Triv tre -> return k (trivial tre)
      | Let(x,a,b) ->
          let k' = Frame(x,b,env,k) in
          eval0 a k'
      | App(tre1,tre2) ->
          apply (trivial tre1) (trivial tre2) k
      | Variant(l,tre) ->
          return k (Vvariant(l,trivial tre))
      | ...  in
    eval0 e k

and apply (v1:value) (v2:value) (k:stack) : value =
    match v1 with
      Vclos (id,exp,env) ->
        eval (Env.extend (!env) id v2) exp k
    | Vfn f -> return k (f v2)

and return k v =
    match k with
      Empty -> v
    | Frame(x,b,env,k) -> eval (Env.extend env x v) b k

let idk = Empty
let evaluate (e:exp) : value = eval initial_env e idk
```


PNF Revisited

The most interesting event in our PNF interpreter is clearly `Let` evaluation. But some `Let` definitions are simpler than others. If the defining expression is a trivial expression, or is an expression kind that itself can only mention trivial expressions (i.e., a `Record`, `Select`, or `Variant`), then we can "unroll" the interpreter code to avoid making a recursive call (or building a new continuation or frame) when evaluating the `Let`, e.g.,

```
let rec eval env (e:exp) =
  match e with
    Let(x,Triv trexp,b) ->
      eval (extend env (x,trivial env trexp)) b
  | Let(x,Variant(l,tre),b) ->
      eval (extend env
                  (x,Vvariant(l,trivial tre))) b
  | ...
```

If the defining expression is a `Switch`, a `Fix`, or a nested `Let`, we cannot do this unrolling, but we can simply outlaw such defining expressions by making a further restriction on our source language. This restricted language is called **A-normal form**. Any ordinary λ-language program can be transformed into this form, but doing so may require duplicating code or introducing new functions. (Details omitted.)

Payoff

After all this, we can write a simplified interpreter that invokes itself recursively (or builds a frame) **only** for `Let`'s where the defining expression is a function application. This corresponds exactly to all non-tail-recursive function applications, i.e., to just the places where we would expect a "stack machine" to make a recursive call.