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

Interpreters

An interpreter is a program that, given a representation of an **expression**, computes a representation of the **value** denoted by that expression.

We can view an interpreter as:

• a mechanism for evaluating a source expression

• a mechanism for defining the **meaning** of (all) source expressions

That is, an interpreter is a kind of operational semantics that has the added advantage of being executable.

To characterize an interpreter, we must specify both the **source language** being interpreted and the **implementation language** in which the interpreter itself is written.

In essence, an interpreter implements the features of the source language in terms of the features of the implementation language. An interpreter is useful for (computation, definition) only if we already (can compute in, understand) the implementation language.

Aside: we can generally turn a denotational semantics directly into an interpreter, provided we use a functional implementation language.

Source language

For this discussion, we'll use a subset of our core λ-calculus as the source language. Unless otherwise stated, we'll want it to have call-by-value semantics.

Our implementation language will be Caml (which is call-by-value) Initially, we'll restrict ourselves to the **pure** subset (i.e., no `ref` cells or mutable fields).

Here's the familiar, somewhat arbitrary, **representation** of the source language in Caml (ignoring types for now):

```
type var = string

type exp
= Var of var
| Abs of var * exp
| App of exp * exp
| Int of int
| Fix of (var * exp) list * exp
| Record of (label * exp) list
| Select of label * exp
| Variant of label * exp
| Switch of exp * (label * exp) list
```


Substitution-based Interpreters

We've already seen several kinds of interpreters. Perhaps the simplest approach to interpretation is based on **substitution**, in which the interpreter models β- and δ-reduction just as it is done on paper.

```
fun subst = ...

exception Bad_prim
exception Bad_switch
exception Bad_select

let rec eval e =
  match e with
    Var v -> Var v
  | Abs(v,e) -> Abs(v,e)
  | App(e1,e2) -> apply(eval e1,eval e2)
  | Int i -> Int i
  | Record(el) ->
     Record(List.map (fun (l,e) -> (l,eval e)) el)
  | ...

and apply (e1,e2) =
    match e1 with
      Var "Iadd" -> ...
    | ...
    | Abs(v,e) => eval(subst e2 v e)
```

Graph Reduction

We have no distinct set of values; computation is described as a process of repeated **rewriting** of source expressions. Because Caml represents variant `type`s using a graph structure, we are essentially performing **graph reduction** when we execute our interpreter on a source program.

This implementation of graph reduction has two key characteristics:

• When we substitute an expression for the bound parameter during an application, we are actually substituting a **pointer** to that expression. In particular, if the parameter appears several times in the body of the abstraction, we don't need to copy it — we just repeat the pointer.

• On the other hand, we **do** copy much of the body of the abstraction each time we apply it. (This is actually done by the `subst` function.) We have to do this, because the function may be applied multiple times, and the original body must be left as a "template" from which copies can be made when needed. This is unpleasantly expensive.

• Recursion needs special treatment, best done using the Y combinator.

Call-by-name

As presented, our interpreter is call-by-value because of the form of the `App` rule. Notice that we rely on Caml's own call-by-value semantics when expressing the behavior of the source language. (But what is the order of argument evaluation??)

We can turn our call-by-value interpreter into a **call-by-name** interpreter by slight changes in the evaluation rules. (What changes, exactly?)

Graph reduction is very well-suited to call-by-name evaluation, because the problem of finding the next substitution point is so simple; it is always as the end of the "left spine" of the graph.

(In call-by-value, by contrast, we may have to do quite a bit of pattern matching to find the next (i.e., leftmost innermost) substitution point.)

More on lazy evaluation later.

Environment-based Interpreters

If we didn't alter λ-abstraction bodies (by substitution) whenever we did an application, there would be no need to copy them. But how can we make a single copy of the body do for every instantiation of the parameters?

A straightforward solution is to interpret the body in the context of an **environment**, an auxiliary structure mapping variables to values. Now, when interpreting an application, we leave the function body alone and merely update the environment with a new **binding** of the function's formal parameter to the actual argument value.

This means a huge potential savings in the cost of function application, but it comes at a cost: a λ-expression containing free variables is no longer meaningful on its own — an environment must be available to interpret the variables. In particular, when we program with λ-abstractions (functions) as values, we must expect to manipulate the body expression and its correct environment **together**.

Environment-based Interpreter

```
type value =
 Vint of int
| Vrecord of (Tcore.label * value) list
| Vvariant of Tcore.label * value
| Vfn of (value -> value)

let rec eval (env:value Env.env) : exp -> value =
  let rec eval0 (e: exp) : value =
    match e with
      Var id -> Env.lookup env id
    | Abs(id,e1) ->
         Vfn (fun x -> eval (Env.extend env id x) e)
    | App(e1,e2) ->
        let v1 = eval0 e1 in
          let v2 = eval0 e2 in
        apply v1 v2
    | Int i -> Vint i
    | Record el ->
         Vrecord(List.map (fun (l,e) -> (l,eval0 e)) el)
    | ...  in
    eval0

  and apply v1 v2 =
    match v1 with
    | Vfn f -> f v2
    | _ -> raise Bad_app

let prims_env =
 Env.extend (Env.Extend Env.empty "Iadd" (Vfn (fun ...))) ...

let evaluate exp = eval prims_env exp
```

How environments work

Note that we now have an explicit syntax for values that is different from that for expressions. Although most of the value constructors correspond directly to source language terms (int, record, etc.), this is **not** true for **functions**, which are represented as Caml functions.

Environments works like this:

• Evaluation takes place in the context of a "current" environment.

• References to variables (Var) are interpreted as lookups in the current environment.

• Abstractions (Abs) are turned into Caml functions describing what to do when the abstraction is applied, namely:

• First create a new environment in which the formal parameter is bound to the actual argument value, and then evaluate the body of the function in this new environment.

(We'll defer recursion for a moment.)

Meta-circularity

Interpreting (source) functions entirely in terms of Caml functions is cute, but it doesn't "say much" about functions. Indeed, since we can think of Caml itself as little more than syntactic sugar for a core λ-calculus (and this is in fact how Ocaml is implemented), we have really succeeded only in expressing core calculus functions (indirectly) in terms of core calculus functions!

This sort of interpreter, where the source and implementation languages are (more or less) the same, is called **meta-circular**. There's nothing illogical or paradoxical about such interpreters. If we already have a working Caml system and just want to execute core-calculus programs, our interpreter works just fine.

But if we're trying to write a translator for the core-calculus from scratch, this kind of interpreter is not of much help. E.g., it tells us nothing about how to actually **represent** functional values at runtime, since it just uses the "underlying" implementation system's representation, which remains hidden.

As an aside, meta-circular interpreters don't tell us much about the **meaning** of the source language either, since they define the language's behavior in terms of its own behavior!

Explicit Function Representation

If we examine our `eval` rule for functions:

```
Abs(id,_,e1) ->
     Vfn (fun x -> eval (Env.extend env id x) e)
```

we see that the right-hand side depends only on `id`,`e`, and `env`. This suggests that we can represent abstraction values in a compact data structure:

```
type value =
   ... VClos of var * exp * env
```

The `Abs` rule gets simpler:

```
Abs(v,e) -> VClos(v,e,env)
```

but the **apply** function gets more complicated:

```
and apply v1 v2 =
  match v1 with
    Vclos (id,exp,env) ->
        eval (Env.extend env id v2) exp
  | Vfn f -> f v2  (* primitives *)
  | _ -> raise Bad_app
```

A **closure**, i.e., the combination of a **code body** plus an **environment** describing the values of the code's free variables, is the standard way to represent functional values in eager functional languages.

Recursive Closures

Consider the trivial source expression

```
let rec f = λx.f x in f 3
```

which translates to the core language expression

```
Fix(["f"],[Abs("x",App(Var "f",Var "x"))],
    App(Var "f",Int 3))
```

Here `(f 3)` is evaluated in an environment mapping `f` to a **closure** value of the form

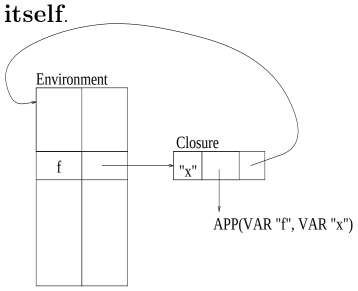
```
VCLos("x", App(Var "f",Var "x"), env)
```

where `env` describes the free variables of the closure expression, namely `f`.

The problem is that `env` must map `f` to an identical closure value. This in turn must contain an environment mapping `f` to an identical closure value, etc., etc. Thus we appear to need an infinitely large environment!

Tying the Knot

One solution is allow the closure's environment to **point to itself**.



We cannot build a cyclic structure like this in the purely functional subset of ML. To do it, we must make either environments or closures **mutable**. For example,

```
datatype value =
   ... Vclos of var * exp * env ref
```

We have these eval rules:

```
  | Abs(v,e) -> Vclos(v, e, ref env)
  | Fix([f,Abs(x,e)],b) =>
      let knotenv = ref env in
      let newenv =
            Env.extend (!env)
                        (f,VCLos(x,e,knotenv)) in
      knotenv := newenv;
      eval newenv b
```


For `apply`, we simply dereference the `env`:

```
  let apply v1 v2 =
      match v1 with
        Vclos (id,exp,env) ->
           eval (Env.extend (!env) id v2) exp
      | ...
```

Mutually recursive functions can be handled simply by making each function closure point to the same `knotenv`: which is set to the original environment repeatedly extended with each function in turn.

Towards Abstract Machines

We now have an interpreter that can evaluate our core λ-calculus, and hence most eager FL programs. Since the interpreter is written in Caml, it is usable for computation if we have a working Caml system, and usable as a semantics to the extent that we already understand Caml.

We want to develop a much lower-level, though still abstract, model in which to evaluate core calculus programs.

Here are some of the main differences between our current system and a real Von Neumann machine:

• λ-calculus expressions are tree-shaped; machine code is linear. In particular, **evaluation order** in the λ-calculus is defined only by the interpreter whereas it is explicit in machine code.

• The λ-calculus permits anonymous expressions to be used as arguments to operators, whereas machines require all values to be given a named address (e.g., a register or memory location).

• Our interpreter uses an abstract description of environments supporting variable lookup by name, and a naive representation for closures. On real machines, we need a much more efficient and explicit representation of environments and closures in terms of records and selection from them.

• Our interpreter supports flexible record and variant subtyping in terms of labels; on real machines, we need to have more efficient representations based on small integer offsets or tags.

• Our interpreter does recursive evaluation of arguments with the implicit aid of Caml's function-call mechanism. Real machines support recursive calls using an **explicit stack** of activation records.

• In fact, even stacks and calls are not built in to most machines; calls and returns must be synthesized from simpler instructions such as jumps. (Since calls are very frequent in the λ-calculus, we want very efficient machine code for them, so it is worth considering the very lowest levels of their implementation.)

Compilation Strategy

Our **strategy** is as follows:

• define a more restrictive version of the λ-language (i.e., a language with fewer expression forms and/or more restrictions on how they are used)

• give an explicit algorithm for **translating** the existing λ-language into the restricted language.

• show how the simpler language can be interpreted by a simpler, more "machine-like" interpreter, having a simpler control structure, manipulating simpler values, and relying less on the particular semantics of the implementation language.

• repeat these steps until our interpreter is simple enough to be called an "abstract machine," making our language abstract machine code.

The resulting abstract machine is both **defined** and **implemented** by the final interpreter in Caml. But we should be able to give other, perhaps more efficient implementations, e.g., in other languages. Or, we can translate the abstract machine code into another low-level language, such as C or assembler.

The composition of the translations defined at each stage constitute a **compiler** for the original λ-language.

Along the way, we can make significant **optimizations** to the program, at various stages in the compilation process.

Serious and Trivial

Our first restricted subset of the λ-language will require that arguments to applications and other basic operations be **named** and that the expression be **flattened** so that the order of evaluation is unambiguous and explicit. We'll call this restricted subset **pretty-normal form** or **PNF**.

These restrictions correspond to the machine-level notions that intermediate results must be assigned to **temporaries** and that order of evaluation is **fixed**.

We don't really want to name all arguments, since some of them are already very simple. So we distinguish two kinds of expressions:

• **Trivial** expressions are those that can be evaluated without any significant computation; these are variables, constants, abstractions, and primops. (These expressions are sometimes called **syntactic values**, since their value representation is the same as their source representation.)

• All other expressions are **serious**, because it requires serious work to evaluate them.

We also want to be lenient about arguments to applications having the form of a **let** binding, under the correspondence:

`let x = a in b` $\equiv$ `(`λ`x.b) a`

Pretty-normal Form

Pretty-normal form is the subset of λ-language where the following subexpressions are restricted to be trivial:

• both `App` arguments, except where the `App` is part of a `Let`
• all `Record` arguments
• the `Select` and `Variant` argument
• the variant argument to `Switch`

Note also that the defining expressions in `Fix`es were already restricted to be `Abs` expressions, and hence are trivial.

To convert an arbitrary λ-language expression to Pretty-normal form, serious expressions appearing in the restricted positions must be replaced by trivial expressions. This can be done by giving **names** to these serious expressions, i.e., `Let`-binding variables to them, and then using the name in place of the original expression. The order in which names are bound is important.

Example of converting to PNF

For example,

```
(f a)(g b)
```

might be translated to

```
let t = f a
in let u = g b
   in t u
   end
end
```

Note that this particular sequence of definitions corresponds to evaluating the subexpressions of the original from left to right. We could just as well choose to evaluate from right to left, producing:

```
let u = g b
in let t = f a
   in t u
   end
end
```

One way or the other, the Pretty-normal form code **fixes** the order of evaluation unambiguously.

Expressing Pretty-Normal form

Pretty-normal form expressions are valid λ-language expressions, so they can be represented by our existing abstract syntax. But we can also use a specialized version of the syntax that separates out the trivial expressions:

```
type trexp =
    Var of var
  | Abs of var * typ * exp
  | Int of int

and exp  =
    Triv of trexp
  | Let of var * typ * exp * exp
  | App of trexp * trexp
  | Fix of (var * trexp * typ) list * exp
  | Record of row
  | Select of label * trexp
  | Variant of label * trexp
  | Switch of trexp * row

and row = (label * trexp) list
```

(Assume that `var`, `label` and `typ` are shared with original expression language.)

This allows us to use the Caml type system to make sure that we've got trivial expressions in the right places.

Interpreting Pretty-Normal Form

Since Pretty-normal form expressions are valid λ-language expressions (modulo changes in syntax), they can be interpreted by our existing λ-language interpreters. But the whole point in defining this form is that there are simpler, more efficient, more machine-like interpreters for PNF expressions.

We use an auxiliary routine that converts trivial expressions to values (this should look "trivial"!)


```
type value =
      Vint of int
    | Vrecord of (label * value) list
    | Vvariant of label * value
    | Vfn of (value -> value) (* prims *)
    | Vclos of var * exp * value Env.env ref

let rec eval (env:value Env.env) : exp -> value =
   let rec trivial (e:trexp) =
      match e with
        Var id -> Env.lookup env id
      | Abs(id,_,e1) -> Vclos(id,e1,ref env)
      | Int i -> Vint i
   and eval0 (e: exp) : value =
      match e with
        Triv trexp -> trivial trexp
      |       Let(x,_,a,b) ->
          let a' = eval0 a in
          eval (Env.extend env x a') b
      |       App(tre1,tre2) ->
          let v1 = trivial tre1 in
          let v2 = trivial tre2 in
          apply v1 v2
      | Record el ->
          Vrecord(List.map
              (fun (l,tre) -> (l,trivial tre)) el)
      | ...
and apply v1 v2 =
    match v1 with
      Vclos (id,exp,env) ->
           eval (Env.extend (!env) id v2) exp
    | ...
```


Tail recursion

This interpreter displays an important property: it is almost completely **tail-recursive**. With the exception of the code for `Let`, all calls to `eval` appear in tail-position.

It is well-known that tail-recursive functions can be converted to iterative ones (in an appropriate setting). That is, functions of the form:

```
f(x) {...; f(y)}
```

can be rewritten as:

```
f(x) {
top: ...;
     x := y;
     goto top }
```

The latter function can be executed without the aid of a stack, and hence with much greater efficiency.

If `eval` were wholly tail-recursive, it would represent a realistic machine with a single global environment and program counter. Recursive calls to `eval` model invocations of an (an indefinite number of) **copies** of a realistic machine!

So we want to get rid of all non-tail calls; the Pretty-normal form interpreter localizes them to `Let` expressions, which is a good first step.

We'll consider a further restriction shortly.