Metaprogramming assignment 3

Optimising embedded languages

Due at noon on Thursday 29th November 2018

This exercise uses the BER MetaOCaml compiler, which you can install via opam. The end of this document has more detailed installation instructions.
1 Optimizing tagless final embedded languages

This exercise focuses on various implementations of a simple embedded DSL, $\text{EXP}$. Each question involves developing a fresh implementation of $\text{EXP}$ to give a new semantics to the DSL, such as evaluation, compilation, partial evaluation, transformation or normalization.

The following diagram shows the implementations provided in the accompanying file (indicated with dotted borders) and developed during this exercise. An arrow from A to B indicates that the implementation B is based on A — that is, that B is obtained by modifying (a copy of) A.

- **tracer (CBV)** (3 marks)
- **tracer (CBN)** (3 marks)
- **CPS** (4 marks)
- **CPS+**
- **evaluator**
- **compiler**
- **partial evaluator**
- **higher-order partial evaluator**
- **partial evaluator + equality** (3 marks)
- **normalizer** (3 marks)

PE with support for ints + bools

- **(a) effect-tracing interpreter (CBV)** (3 marks)
- **(b) effect-tracing interpreter (CBN)** (3 marks)
- **(c) PE with support for $\lambda$** (4 marks)
- **(d) PE using term equality** (3 marks)
- **(e) CPS-based evaluator** (4 marks)
- **(f) parameterized CPS-based evaluator** (3 marks)
- **(g) term normalizer** (3 marks)
The language is a simple expression language with a single effect, for printing:

\[
\begin{align*}
e, e_1, e_2, \ldots & \ ::= \ x, y, z \quad \text{variables} \\
\lambda x . e & \quad \text{functions} \\
e_1 \ e_2 & \quad \text{applications} \\
\text{false, true} & \quad \text{booleans} \\
\text{if } e_1 \ e_2 \ e_3 & \quad \text{conditionals} \\
1, 2, \ldots & \quad \text{integers} \\
e_1 + e_2 & \quad \text{addition} \\
\text{printv} & \quad \text{printing}
\end{align*}
\]

The addition of the printing effect makes it possible to distinguish between call-by-name and call-by-value evaluation orders.
(a) Tracer (Call By Value)

The tagless style makes it easy to implement embedded languages directly, using the facilities of the host (meta) language to implement the facilities of the embedded (object) language. For example, the `eval` implementation implements the `print` operation in `EXP` using OCaml’s `print_int` function.

However, it is sometimes more flexible to implement DSLs less directly. The `Trace` implementation of `EXP` is an alternative CBV evaluator that collects a list of the (print) effects performed when evaluating a term rather than executing them directly:

\[
\text{module Trace : EXP with type 'a t = 'a trace = ...}
\]

Complete the implementation of `Trace` and check its behaviour on some examples:

```ocaml
# Trace.(app (app (lam (fun b ->
    lam (fun e -> if_ b e (print (int 1))))))
    (bool false))
(print (int 2)));;
- : unit trace = Unit ((), [2; 1])
```

(b) Tracer (Call By Name)

Eager languages like OCaml evaluate function arguments before calling the functions, following the so-called Call by Value (CBV) evaluation order.

However, other evaluation orders are possible, too: using Call by Name (CBN) evaluation, arguments are only evaluated at the point where they are used in the called function, not before the call.

Complete the implementation of `TraceCBN`, an alternative version of `Trace` using call by name evaluation and check its behaviour on some examples:

```ocaml
# TraceCBN.(app (app (lam (fun b ->
    lam (fun e -> if_ b e (print (int 1))))))
    (bool false))
(print (int 2)));;
- : unit trace = Unit ((), [1])
```

The notation `M.(e)` is short for `let open M in e`, which makes the definitions from the module `M` available for use in `e`. 

OCaml hint

OCaml hint

OCaml hint

OCaml hint

OCaml hint

OCaml hint

OCaml hint

OCaml hint
(c) Partial evaluator (higher-order)

In the lectures we saw an example of a simple partial evaluator that reduced arithmetic and boolean expressions, but not functions.

A more advanced partial evaluator can simplify functions too. For example, the following application can be simplified because the function is statically known:

\[(\lambda x. x + z) 4 \leadsto 4 + z\]

Here are some more examples that a higher-order partial evaluator can simplify:

\[
\lambda x. 4 + ((\lambda y. y + 1) 2) \leadsto \lambda x. 7
\]

\[
\lambda x. ((\lambda f. (\lambda z. f z))(\lambda y. y + x)) 5 \leadsto \lambda x. 5 + x
\]

Complete the partial evaluator \(PE\) so that it simplifies these expressions and check its behaviour on some examples:

```ocaml
# resid PE.(lam (fun x ->
  add (int 4)
  (app (lam (fun y -> add y (int 1)))
  (int 2))));;
- : (_weak8 -> int) code = <fun x_31 -> 7>.
```

Heed warnings about cross stage persistence. They often indicate a staging mistake — e.g. writing \(~x\) where \.`x` was intended.

(d) Partial evaluator, improved

There are several ways to further enhance the partial evaluator. One is to eliminate both branches of a conditional when they are known to be the same:

\[\text{if } e_1 e \text{ e } \leadsto e\]

However, it is not always possible to determine when two terms are the same; code values cannot be inspected. The function `equalp` is a best-efforts equality function, returning `Yes`, `No` or `Unknown` to indicate whether its arguments are equal:

```ocaml
type equal = Yes | No | Unknown
let rec equalp : type a. a static -> a static -> equal = ...
```

Here are some examples of `equalp`’s behaviour:

- `equalp (Int 3) (Int 3) \sim Yes`
- `equalp (Bool false) (Bool true) \sim No`
- `equalp (Fun f) Unknown \sim Unknown`

Complete `equalp` and use it to implement an improved partial evaluator `PE2` that simplifies conditionals where both branches are the same.
(e) CPS evaluator

As discussed in lectures, writing an evaluator in Continuation-Passing Style (CPS) has a number of advantages. One advantage we’ll explore a little here is that CPS and partial evaluation interact to better optimize programs.

Complete the module CPS to implement a CPS evaluator for EXP:

```ocaml
type 'a cps = {k: 'b, ('a -> 'b) -> 'b}
module CPS : EXP with type 'a t = 'a cps = ...
```

Check the behaviour of your evaluator on some examples:

```ocaml
# CPS.(lam (fun f -> app f (app f (int 3)))) .k (fun x -> x). succ ;;
- : int = 5
```

OCaml records like `cps` can have polymorphic fields, which can simulate universally quantified types like $\forall b. (a \rightarrow b) \rightarrow b$:

(f) CPS evaluator, modularized

It can be useful to write EXP implementations in a way that supports composition.

The CPS module can be made composable by parameterizing it by another implementation of EXP:

```ocaml
module CPS2 (E: EXP) = ...
```

While CPS uses host language operations (+, if, &c.) to implement the operations of EXP, CPS2 should instead use the operations of $E$ (E.add, E.if_, E.lam, &c.).

Complete the implementation of CPS2.

Test your implementation by applying CPS2 to other modules (Compile, PE, PE2).

```ocaml
# module CPSPE = CPS2(PE2);;
... # CPSPE.(lam (fun b -> add (int 2) (if_ b (int 3) (int 4)))) .k (fun x -> x);;
- : (bool -> int) sd = {sta = Fun <fun>; dyn = <fun x_15 -> if x_15 then 5 else 6>. }
```
Finally, we consider an alternative way of normalizing terms.

The **Normal** implementation of **EXP** transforms every term into a form with the following properties:

- Every non-trivial non-value expression (uses of `add` and `print`, and function calls) is `let`-bound. In the following example the expressions `f 3` and `x₁ + 4` are `let`-bound:

  ```ocaml
  add (app f (int 3)) (int 4)  →
  let x₁ = f 3 in
  let x₂ = x₁ + 4 in
  x₂
  ```

- No value expressions (variables, constants, lambdas) are `let`-bound.

- The function part of an application is always a variable, not a lambda.

Complete the implementation **Normal** and test its behaviour on some examples:

```ocaml
# residn Normal.(lam (fun x ->
  app (lam (fun c -> c))
  (add (add (int 3) x) x))));;
- : (int -> int) code = .< fun x_31 -> let x_32 = 3 + x_31 in
  let x_33 = x_32 + x_31 in
  x_33>.

# residn Normal.(lam (fun b ->
  lam (fun x ->
    add (int 4)
    (if_ b (int 0)
      (app (lam (fun x -> x))
        (add (int 3) x))))));;
- : (bool -> int -> int) code = .<
  fun x_24 ->
  fun x_25 ->
    if x_24
      then let x_28 = 4 + 0 in x_28
    else (let x_26 = 3 + x_25 in let x_27 = 4 + x_26 in x_27)>.
```

8
MetaOCaml: what you need to know

MetaOCaml is an extension of OCaml with support for quotation-based code generation. This page describes the installation and use of MetaOCaml, along with a brief summary of the language constructs needed for the exercise.

How to install MetaOCaml

Installing MetaOCaml is a two-step process:

1. Install opam, the OCaml package manager, following the instructions here: https://opam.ocaml.org/doc/Install.html

2. Use opam to install the MetaOCaml compiler:

   opam switch 4.07.1+BER
   eval $(opam env)

   (If you have difficulty installing 4.07.1+BER you might try the previous version 4.04.0+BER instead.)

How to run MetaOCaml

Type metaocaml to start the MetaOCaml top level:

   $ metaocaml
   BER MetaOCaml toplevel, version N 107
   OCaml version 4.07.1

#

Within the top level, type #use "tagless.ml";; to load the code.

You can evaluate individual expressions and definitions, too, and MetaOCaml will print their type and values. Follow each phrase with a double semicolon.

   # let f x = x + 1;;
   val f : int -> int = <fun>
   # f 3;;
   - : int = 4
   # let x = .< 1 + 2 >. in .< .˜x + .˜x >.;;
   - : int code = .<(1 + 2) + (1 + 2)>.
MetaOCaml syntax

Function definitions

Here is a recursive function, with name \( f \) and type \( \forall a. \text{bool} \to a \to a \):

```ocaml
let rec f : type a. bool -> a -> a = 
  fun b x -> if b then f (not b) x else x
```

You can often omit the type:

```ocaml
let rec f = 
  fun b x -> if b then f (not b) x else x
```

or, even more concisely:

```ocaml
let rec f b x = 
  if b then f (not b) x else x
```

If \( f \) is not recursive, you can omit \texttt{rec}, too:

```ocaml
let f b x = if b then x else ()
```

Data types and pattern matching

A data type is defined by giving a signature for each constructor:

```ocaml
type 'a option = 
  None : 'a option 
| Some : 'a -> 'a option
```

There are two constructors for \texttt{option}:

- \texttt{None} (no arguments, returns \texttt{`a option})
- \texttt{Some} (argument of type \texttt{`a}, returns \texttt{`a option})

Examine values by pattern matching:

```ocaml
match x, y with 
  | Some a, Some b -> a + b 
  | Some a, None -> a 
  | None , b -> b 
  | None , None -> 0
```

Modules

MetaOCaml programs are built of modules:

```ocaml
module Ints = struct 
  type 'a t = Int : int -> int t 
  let int x = Int x 
end
```

Modules have types called \texttt{signatures}

```ocaml
module type INTS = sig 
  type 'a t 
  val int : int -> int t 
end 
module I = ( Ints : INTS )
```

Modules can be parameterized by other modules

```ocaml
module IntList(I: INTS ) = struct 
  type t = Nil : t 
  | Cons : int I.t * t -> t 
  let rec length = ... 
end 
module L = IntList(I)
```

Records

Declare a records by listing fields & types:

```ocaml
type ('a,'b) pair = { one : 'a; two : 'b}
```

The \texttt{pair} type has two type parameters, \texttt{`a} and \texttt{`b}, and two fields: \texttt{one} of type \texttt{`a} and \texttt{two} of type \texttt{`b}.

Construct records by values for fields:

```ocaml
let p = { one = 3; two = "four" }
```

and access fields using projection:

```ocaml
print_endline p.two
```
Quotations

MetaOCaml provides two constructs for building code values.

**Brackets** (.< ... >.) delay the evaluation of an expression to build a piece of code:

```
.< 1 + 2 >.
```

The inserted expression must have code type.

If an expression `e` has type `t` then `<e>` has type `t code`.

**Escape** (."`) is for inserting one piece of code into another:

```
let x = .< 1 + 2 >. in
```

You can run a piece of code using the `Runcode.run` function:

```
Runcode.run .< 1 + 2 >. ~> 3
```

MetaOCaml supports **open code**: quotations with free variables. In this example, `<x>` contains a free variable.

```
.< fun x -> .˜(f .<x>.) >.
```

(But note that `x` is bound in an outer scope!)