Advanced Functional Programming exercise 3

Due: Friday 24 April 2015

Jeremy Yallop

March 10, 2015

Background

This exercise involves improving the efficiency of two programs by the use of staging. Pages 1–5 provide background information about the programs and information about setting up your environment. The subsequent pages contain the questions. The skeleton code for the exercise is available from the course web page:

`http://www.cl.cam.ac.uk/teaching/1415/L28/assessment.html`

Binary parsing

Parsing binary data — image formats, network packets, bytecode sequences, etc. — is a common programming need. In contrast to textual data, which is built from characters, binary data is built from bits. For example, a GIF image file starts with the following 104-bit sequence:

- (48 bits) either the string ”GIF87a” or the string ”GIF89a”
- (16 bits) the (logical) image width
- (16 bits) the (logical) image height
- (1 bit) whether a colour table is present
- (3 bits) the size of the colour table
- (1 bit) the sort order of the colour table
- (3 bits) the colour resolution
- (8 bits) the background colour
- (8 bits) the pixel aspect ratio

In this exercise you'll build various simple libraries for parsing binary data. Here is a parser for the GIF format written using one of the libraries:

```
pure (fun version width height colour_table
          colour_bits sortflag bps bg aspect_ratio →
        { version; width; height; colour_table;
          colour_bits; sortflag; bps; bg; aspect_ratio }) <*>
int64 48 <*>
int64 16 <*>
int64 16 <*>
int64 1 <*>
int64 3 <*>
int64 1 <*>
int64 3 <*>
int64 8 <*>
int64 8
```

Staging The performance of binary parsing can be improved by staging, since the format of binary data is typically known in advance of the availability of the data itself.

Practicalities You may find the `genlet` library described in lectures useful when staging your parsing library. You can install `genlet` using OPAM as follows. First, switch to the BER MetaOCaml compiler:

```
opam switch 4.02.1+BER
eval `opam config env`
```

Next, update OPAM's package list and install `genlet`:

```
opam update
opam install genlet
```

In OCaml 4.02 you can see the interface to a module using the `#show_module` command in the top level. Here's a sample session that shows how to view the interface of `genlet`:

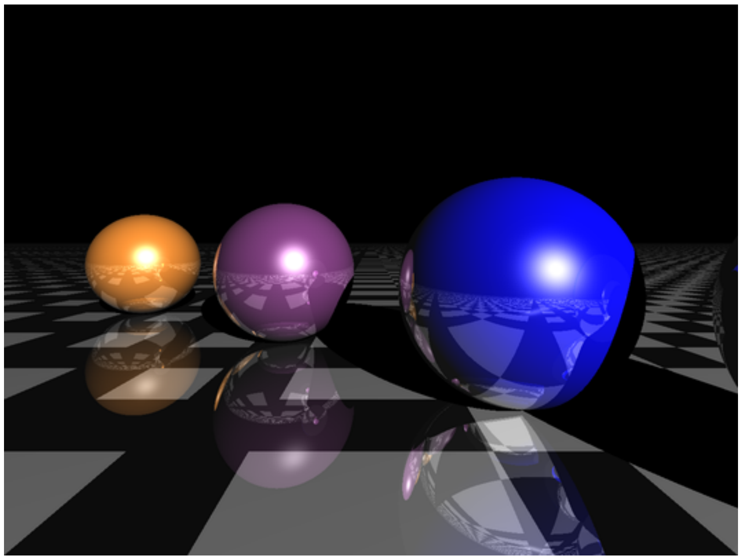
```
$ metaocaml
BER MetaOCaml toplevel, version N 102
        OCaml version 4.02.1

# #use "topfind";;
- : unit = ()
Findlib has been successfully loaded. Additional directives:
  #require "package";;      to load a package
  #list;;                   to list the available packages
  #camlp4o;;                to load camlp4 (standard syntax)
  #camlp4r;;                to load camlp4 (revised syntax)
  #predicates "p,q,...";;   to set these predicates
  Topfind.reset();;         to force that packages will be reloaded
  #thread;;                 to enable threads

- : unit = ()
```

```
# #require "genlet";;
/home/jeremy/.opam/4.02.1+BER/lib/delimcc: added to search path
/home/jeremy/.opam/4.02.1+BER/lib/delimcc/delimcc.cma: loaded
/home/jeremy/.opam/4.02.1+BER/lib/genlet: added to search path
/home/jeremy/.opam/4.02.1+BER/lib/genlet/genlet.cma: loaded
# #show_module Gengenlet;;
module Gengenlet :
  sig
    val genlet : 'a code -> 'a code
    val let_locus : (unit -> 'w code) -> 'w code
  end
```

Ray tracing



Ray tracing is a technique for generating photorealistic images by simulating the interactions of rays of light with the objects in a scene. The inputs to a ray tracer are the objects in the scene and details about the position and orientation of the camera and light sources. The ray tracer determines where the rays of light intersect the various objects in the scene and computes the shadows, reflections and refractions that result.

Ray tracing is computationally expensive, since the interaction of a ray of light with a reflective object can produce further rays, which require recursive applications of the algorithm. However, it is possible to improve the performance of a general-purpose ray tracer by staging, since some of the data remains fixed throughout the algorithm's execution. In particular, the scene — i.e. the positions and properties of the objects in the image — can be treated as static for staging purposes.

In this exercise you'll improve the performance of an existing ray tracing implementation by staging. It will **not** be necessary to understand the technical details of ray tracing; it is sufficient to know which data is available statically and use standard techniques to transform the central function from a general-purpose ray tracer into a code generator that generates a ray tracer specialised to a particular scene.

Practicalities The ray tracer uses the `camlimages` library. It is possible to complete the exercise without the library, but if you would like to use the ray tracer to actually build images then you can install the library using OPAM:

```
opam install camlimages
```

Byte code vs native code The standard OCaml distrbiution comes with two compilers: `ocamlc` turns OCaml source into bytecode which can be executed by a virtual machine, and `ocamlopt` turns OCaml source into native code. The current release of MetaOCaml has better support for bytecode than for native code[1], so the ray tracer distributed with the exercise uses the bytecode compiler rather than the native code compiler. This makes the ray tracer artificially slow, but we are less interested in the absolute speed of the program than in the relative speed improvements that can be achieved by staging.

[1]`http://okmij.org/ftp/ML/MetaOCaml.html#native`

1 Bit-level parsing

This question involves building bit-level parsers based around the following interface:

```
module type BIT_PARSER =
sig
  type _ t
  val int64 : bits:int → int64 t
  val parse : 'a t → int64_array → 'a
end
```

The `BIT_PARSER` signature includes a type `'a t` of parsers, a function `int64` for constructing parsers for values with a specified number of bits, and a function `parse` which uses parsers to read values from arrays of integers.

(*a*) (*i*) Bit-level parsers can be implemented using a function `extract_bits` that reads integers from arbitrary positions in an array:

```
val extract_bits : int * int → int64_array → int64
```

The first argument to `extract_bits` represents the range of bits to read from the array. For example,

- `extract_bits (0,8) arr` reads the first 8 bits of the first element of `arr`.
- `extract_bits (65,67) arr` reads bits 2 and 3 of the second element of `arr`.
- `extract_bits (60,70) arr` reads an integer which is built from the last 4 bits of the first element of `arr` and the first 6 bits of the second element of `arr`.

Give an implementation of `extract_bits`.

[2 marks]

(*ii*) The operations of `BIT_PARSER` together with the monad operations support combining parsers to build more complex parsers:

```
module type BIT_PARSERM =
sig
  include MONAD
  include BIT_PARSER with type 'a t := 'a t
end
```

For example, here is a parser which reads two 6-bit integers and returns them as a pair:

```
int64 6 >>= fun x →
```

```
int64 6 >>= fun y →
 return (x, y)
```

Use `extract_bits` to write an implementation of `BIT_PARSERM`:

```
module Bit_parserM : BIT_PARSERM = ...
```

[3 marks]

(*iii*) Combining `BIT_PARSER` with the applicative interface gives an alternative interface for building composable parsers:

```
module type BIT_PARSERA =
sig
  include APPLICATIVE
  include BIT_PARSER with type 'a t := 'a t
end
```

For example, here is a parser built using `BIT_PARSERA`:

```
pure (fun x y → (x, y)) <*> int64 6 <*> int64 6
```

Give an implementation of `BIT_PARSERA`:

```
module Bit_parserA : BIT_PARSERA = ...
```

[3 marks]

(*iv*) The two parser interfaces above are not equivalent. Show that `BIT_PARSERM` is the more powerful of the two by describing a data format which can be parsed using `BIT_PARSERM`, but not using `BIT_PARSERA`, and give a parser for the format.

[3 marks]

(*b*) (*i*) Parsers built using the interfaces in part (*a*) are typically less efficient than hand-written code. We can eliminate the extra overhead introduced by the abstraction using staging. Give an implementation of the code-generation function `extract_bits_staged` that performs as much work as possible during code construction:

```
val extract_bits_staged :
  int * int → int64_array code → int64 code
```

[3 marks]

(*ii*) The `BIT_PARSERA` interface can be extended to support staging by adding a function `compile`:

```
module type BIT_PARSERA_STAGED =
sig
  include BIT_PARSERA
  val compile : 'a t → (int64_array → 'a) code
end
```

Give an implemntation of `BIT_PARSERA_STAGED` that performs as much work as possible during code generation:

```
module Bit_parserA_staged : BIT_PARSERA_STAGED = ...
```

[4 marks]

(*iii*) Reading elements from an array is a comparatively expensive operation. Extend your implementation of `Bit_parserA_staged` (using the genlet library and or otherwise) so that each element of the input array is read at most once during parsing. (You may find it helpful to write a second implementation of `extract_bits_staged` with an interface more suited to `let` insertion.)

[4 marks]

(*iv*) The `BIT_PARSERA` interface is a more suitable starting point for a staged implementation than `BIT_PARSERM`. Briefly explain why.

[3 marks]

2 Ray tracing

This question involves staging a ray tracer to improve its performance.

(*a*) The functions `+|`, `-|`, `*|` and `/|` perform element-wise vector arithmetic. For example, `+|` is defined as follows:

```
let (+|) l r = {x=l.x+.r.x; y=l.y+.r.y; z=l.z+.r.z}
```

Write staged versions of the vector arithmetic functions:

```
val ( +|@ ) : float3 sd → float3 sd → float3 sd
val ( -|@ ) : float3 sd → float3 sd → float3 sd
val ( *|@ ) : float3 sd → float3 sd → float3 sd
val ( /|@ ) : float3 sd → float3 sd → float3 sd
```

where `sd` is the type of possibly-static values

```
type _ sd =
    Sta : 'a → 'a sd
  | Dyn : 'a code → 'a sd
```

Your implementations should generate efficient code by making use of statically-available information. For example, the result of the following expression

```
Sta {x=1.0; y=1.0; z=1.0} *|@ Dyn .<v>.
```

should be the value `Dyn .<v>.`.

(Hint: You may like to begin by writing staged versions of the floating-point arithmetic operations `+.`, `-.`, `*.` and `/.`.)

[7 marks]

(*b*) Write a staged version of the inner product function, `dot`, making use of statically-available information to generate efficient code:

```
val dot_ : float3 sd → float3 sd → float sd
```

[1 mark]

(*c*) The `intersect` function computes the point at which a ray intersects a sphere or a plane.

Write a staged version of `intersect`, making use of statically-available information to generate efficient code:

```
val intersect_ :
  float3 code → float3 code → obj → float code
```

[5 marks]

(d) The trace_ray function uses intersect to implement the ray tracing algorithm.

Write a staged version of trace_ray, making use of statically-available information to generate efficient code:

```
val trace_ray_ :
  specular:float →
  camera:float3 →
  ambient:float →
  light_colour:float3 →
  light_position:float3 →
  scene:obj list →
  float3 code → float3 code → (obj * float3 * float3 * float3)
     option code
```

[10 marks]

(e) (i) Comment briefly on the performance differences you observe between the staged and unstaged implementations of trace_ray [1 mark]

(ii) To what extent is staging a useful technique for improving the performance of ray tracing libraries? [1 mark]