L28: Advanced functional programming

Exercise 3

Due on 25th April 2016

Submission instructions

Your solutions for this exercise should be handed in to the Graduate Education Office by 4pm on the due date. Additionally, please email your complete compilable implementation to jeremy.yallop@cl.cam.ac.uk.

Background: bit-level 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 and use 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 }) <*>
bits 48 <*>
bits 16 <*>
bits 16 <*>
bits 1 <*>
bits 3 <*>
bits 1 <*>
bits 3 <*>
bits 8 <*>
bits 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
```

Background: the Fast Fourier Transform

The Fast Fourier Transform (FFT) is an efficient ($O(N \log N)$) variant of the $O(N^2)$ Discrete Fourier Transform (DFT), which is defined as follows for complex numbers $x_0 \ldots x_{N-1}$.

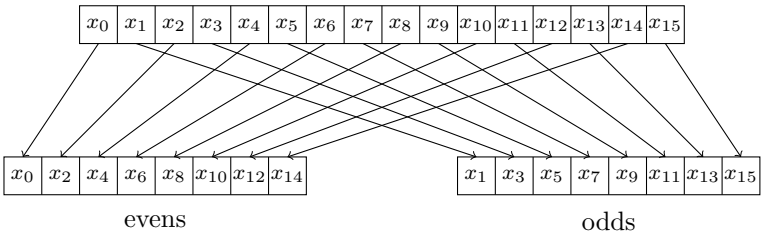
$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} nk}$$

The FFT has applications in signal processing, crystallography, efficient multiplication of large integers, solving partial differential equations, and many more areas. This exercise focuses on constructing an efficient implementation of the core algorithm using staging.

The most commonly-used variant of the FFT is known as the *Cooley-Tukey algorithm*, and operates as follows:

1. Split the input array (i.e. $x_0 \ldots x_{N-1}$) into even and odd components.

   ```
   let (evens,odds) = split arr in
   ```

2. Perform the FFT on the smaller arrays

   ```
   fft evens
   fft odds
   ```

3. For each pair of elements `x` and `y` at position `j` in the resulting arrays, compute

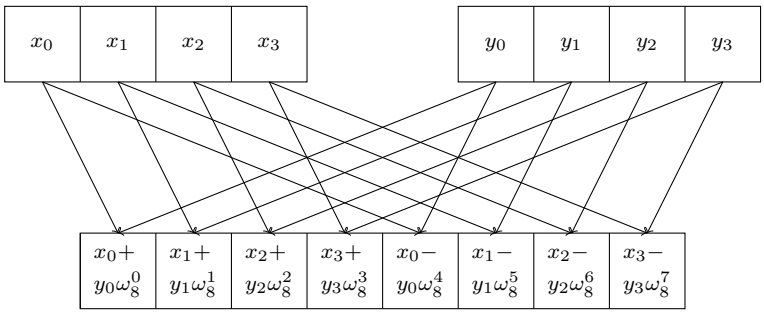
   `x` + `y`ω_n^j

   and

   `x` − `y`ω_n^j

   where `n` is the length of the input array and ω_n^0 to ω_n^n are the nth roots of unity — i.e. those numbers x for which $x^n = 1$.

4. Collect the results from step 3 as indicated in the folllowing diagram:



More comprehensive information about the FFT is available in many algorithms textbooks. However, as is often the case with staging, transforming the Fast Fourier Transform implementation into a staged version can be performed using standard techniques, without needing to understand the algorithm in detail.

1 Bit-level parsing

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

```
module type BIT_PARSER =
sig
  type ('a, 'b) parser
  val bits : (int, int64) parser
  val parse : ('a, 'b) parser -> 'a -> int64 array -> 'b
end
```

where the type `parser` represents a parser (parameterised by a value of type `'a`) which builds a value of type `'b`, the function `bits` constructs parsers for values with a specified number of bits, and the function `parse` uses parsers to read values from arrays of integers.

(*a*) The `Parser_monad` module combines the operations of `BIT_PARSER` with the monad operations to support building more complex parsers:

```
module Parser_monad :
sig
  include MONAD
  include BIT_PARSER with type ('a, 'b) parser = 'a -> 'b t
end
```

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

```
bits 6 >>= fun x ->
bits 6 >>= fun y ->
return (x, y)
```

Similarly, the `Parser_applicative` and `Parser_arrow` modules support parser construction using the `APPLICATIVE` and `ARROW` interfaces.

Use `Parser_monad`, `Parser_applicative` and `Parser_arrow` to build parsers for the following simple network packet format.

src (6 bytes)	**dst** (6 bytes)	**type** (16 bits)

(4 marks)

(*b*) It is often the case that the format of a field in a serialized data format depends on the values of previous fields. For example, consider the following format, which describes pairs of values preceded by a tag indicating the size of the values.

size (2 bits): either 0: 1 bit 1: 8 bits 2: 32 bits 3: 64 bits	**fst** (**size** bits)	**snd** (**size** bits)

Not all of the interfaces `Parser_monad`, `Parser_applicative` and `Parser_arrow` can express parsers for this format. For each interface, implement a parser if possible, or, if building a parser is impossible, explain the difficulty.

(4 marks)

(*c*) Describe a data format which can only be parsed using *one* of the interfaces `Parser_monad`, `Parser_applicative` and `Parser_arrow` and implement a parser for that format.

(3 marks)

(*d*) Parsers built using the interfaces above 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)

(*e*) Implement a module `Parser_arrow_staged` with the following interface

```
module Parser_arrow_staged :
sig
  include ARROW
  include BIT_PARSER with type ('a, 'b) parser = ('a, 'b) t
  val compile : ('a, 'b) t -> ('a -> int64 array -> 'b) code
end
```

that performs as much work as possible during code construction. In particular, you should consider reading elements from an array to be a comparatively expensive operation, and aim to minimize the number of times each element of the input array is read during parsing. (Hint: the `let_locus` and `genlet` operations may be helpful.)

[*Addendum (26th March)*: Optionally, you may like to extend the interface with

the following datatype for describing pure functions:

```
type ('a, 'b) pure =
    Const : 'a -> ('b, 'a) pure
  | Dup : ('a, 'a * 'a) pure
  | First : ('a, 'b) pure -> ('a * 'c, 'b * 'c) pure
```

where the interpretation of each constructor is as given by the following function

```
let rec interpret : type a b. (a, b) pure -> a -> b = function
    Const x -> fun y -> x
  | Dup -> fun x -> (x, x)
  | First f -> fun (x, y) -> (interpret f x, y)
```

and the `Parser_arrow_staged` interface is extended with a second version of `arr` that maps `pure` values to computations:

```
module Parser_arrow_staged :
sig
  ...
  val arr' : ('a, 'b) pure -> ('a, 'b) t
end
```

If you wish, you may also extend `pure` with constructors for other pure functions such as `id`, `assoc`, $\circ$ and `swap` as defined on p157 of the lecture notes for 15th February.]

(3 marks)

(f) Which of the three interfaces `Parser_monad`, `Parser_applicative` and `Parser_arrow` provides the most suitable basis for staged parsing?

(2 marks)

2 The Fast Fourier Transform

This question involves staging an implementation of the Fast Fourier Transform to improve its performance. The code distributed with this exercise includes an unstaged FFT implementation. Your task is to build on that implementation to generate specialized code based on statically-known information.

The principal data structure in the FFT algorithm is an array of complex numbers. Refining the structure to support partially-static data will make it possible to build an effective code generator.

(*a*) The FFT implementation in the `Fft_unstaged` module makes use of the OCaml standard library module `Complex`, which implements complex numbers. The staged FFT implementation is based on a simpler module `Complex_staged` distributed with this exercise. The current implementation of `Complex_staged` is not very efficient.

(*i*) Each of the functions in `Complex_staged` — `add`, `mul`, `sub`, etc. — operates on possibly-static data. Improve the implementations of the module to take advantage of static information when possible. For example, the expression `Complex_staged.mul (Sta x) (Sta y)` currently builds a dynamic value, but it should instead build a static value.

(Take care to consider the cases where one of the arguments to a function is a static value with special properties. For example, `mul (Sta 0.0) x` should always be `Sta 0.0`, regardless of whether `x` is static or dynamic.)

(*ii*) Improve the implementations of `complex_sd` and `dyn_complex` in the `Fft_types` module to support the case where only one component of a complex number (i.e. only the real, or only the imaginary part) is static. What new optimizations does your improved definition make possible?

(*iii*) Given three values `x`, `y` and `z`, of type `Complex_staged.t`, of which two are static and one dynamic, under what circumstances will the following expression perform some static computation?

```
Complex_staged.add x (Complex_staged.add y z)
```

Sketch briefly how you might change the definition of `Complex_staged.t` to expose more opportunities for static computation. You can use code examples if you like, but you do not need to give a full implementation.

(8 marks)

(*b*) The implementation of arrays in the `Fft_unstaged` module is based on balanced

binary trees containing complex numbers. The implementation of arrays in the `Fft_staged` module is based on balanced binary trees containing *possibly-static* complex numbers. The structure of the array is known statically, since the length is known statically, but some of the contents may only be available dynamically.

(*i*) Implement the functions `Arr.mk` and `Arr.dyn` to convert between `Arr.t` and `array code` values. In each case the order of elements in the `array` value should correspond to the order of elements in the `Arr.t` value: that is, given an `Arr.t` value like this

```
Branch (Branch (Leaf a, Leaf b), Branch (Leaf c, Leaf d))
```

the order of elements in the corresponding array should be as follows:

```
[|a; b; c; d|]
```

(*ii*) Implement the top-level function `mk` in the `Fft_staged` module to build a code generator by calling `fft` and whichever other functions you need.

(*iii*) Comment briefly on the performance differences you observe between the staged and unstaged implementations of `fft`. You might like to compare the running time or the number of operations executed for a few different array sizes.

(*iv*) To what extent is staging a useful technique for improving the performance of numerical algorithms?

(8 marks)