Computer Science 131
Programming Languages

October 24, 2000

Structures and Functors

Review

- A collection of definitions can be packaged into a *structure*.
- A *signature* is a corresponding collection of specifications of structure components.
- A structure satisfies a signature if it contains definitions matching each specification.
- Signature ascription `:>` can be used to "seal" a structure, hiding all information not in the signature.

Recall: Interface for Dictionaries

```
signature DICT = sig
   type key
   type 'a dict
   val empty : 'a dict
   val insert : 'a dict * key * 'a -> 'a dict
   val lookup : 'a dict * key -> 'a
end
signature INTDICT =
   DICT where type key = int
signature STRINGDICT =
   DICT where type key = string
```

Functors

- Parameterized structures
  - A way to build structures
- For example, definitions of structures satisfying **`INTDICT`** and **`STRINGDICT`** will share most of the same code
  - We can create a functor to generate dictionaries for keys of different types.
  - Given information about keys, create dictionary module

A Dictionary Functor

```
functor Dict(type t
             val eq : t * t -> bool) =
  struct
    type key = t
    type 'a dict = (key * 'a) list
    val empty = []
    fun insert(...) = ...
    fun lookup_helper(...) = ...
    fun lookup(...) = ...lookup_helper...
  end
```

Functor Applications

```
functor Dict(type t
             val eq : t * t -> bool) = ...

structure StringDict =
  Dict(type t = string
       fun eq(s1:string,s2:string) = (s1=s2))

structure IntDict =
  Dict(type t = int
       fun eq(n1:t,n2:t) = (n1=n2))
```

A Dictionary Functor

```
functor Dict(type t
             val eq : t * t -> bool) =
  struct
    type key = t
    type 'a dict = (key * 'a) list
    val empty = []
    fun insert(...) = ...
    fun lookup_helper(...) = ...
    fun lookup(...) = ...lookup_helper...
  end
```

A Dictionary Functor

```
functor Dict(type t
             val eq : t * t -> bool) =
  struct
    type key = t
    type 'a dict = (key * 'a) list
    val empty = []
    fun insert(...) = ...
    fun lookup_helper(...) = ...
    fun lookup(...) = ...lookup_helper...
  end :> DICT     (* Is this right??? *)
```

A Dictionary Functor

```
functor Dict(type t
             val eq : t * t -> bool) =
  struct
    type key = t
    type 'a dict = (key * 'a) list
    val empty = []
    fun insert(...) = ...
    fun lookup_helper(...) = ...
    fun lookup(...) = ...lookup_helper...
  end :> DICT where type key = t
```

Alternative Interface

```
signature KEY = sig
   type key
   val eq : key * key -> bool
end
signature DICT = sig
   structure K : KEY
   type 'a dict
   val empty : 'a dict
   val insert : 'a dict * K.key * 'a -> 'a dict
   val lookup : 'a dict * K.key -> 'a
end
signature INTDICT = DICT where type K.key = int
```

Alternative Dictionary Functor

```
functor Dict(structure Key : KEY) =
  struct
    structure K = Key
    type 'a dict = (K.key * 'a) list
    val empty = []
    fun insert(...) = ...
    fun lookup_helper(...) = ...
    fun lookup(...) = ...lookup_helper...
  end :> DICT where type K.key = Key.key
```

Plumbing

- Consider the signatures

```
signature LEXER = sig
  type token
  val lex : string -> token list
end

signature PARSER = sig
  type token
  type absyn
  val parse : token list -> absyn
end
```

Plumbing

- Suppose we want to combine two such structures into the front-end of a compiler:

```
signature FRONTEND =
    sig
       type absyn
       val doit : string -> absyn
    end
```

Plumbing

- Is the following OK?

```
functor FrontEnd(structure Lexer : LEXER
                 structure Parser : PARSER) =
  struct
     type absyn = Parser.absyn
     fun doit(s) = Parser.parse(Lexer.lex s)
  end
```

Plumbing

- Oops…no guarantee that `Lexer.token = Parser.token`
- The functor will not typecheck.

```
functor FrontEnd(structure Lexer : LEXER
                 structure Parser : PARSER) =
  struct
     type absyn = Parser.absyn
     fun doit(s) = Parser.parse(Lexer.lex s)
  end
```

Plumbing

- We can *require* that **`Lexer.token = Parser.token`**

```
functor FrontEnd(structure Lexer : LEXER
                 structure Parser : PARSER
                 sharing type Lexer.token =
                                     Parser.token)
= struct
     type absyn = Parser.absyn
     fun doit(s) = Parser.parse(Lexer.lex s)
  end
```

Extended Example

- Network protocols
  - Conceptually built in layered fashion
  - Each layer provides more functionality
  - Example: TCP/IP over ethernet
    - Ethernet: Communication among machines on one wire
    - IP: Multi-hop communication, fragmentation/reassembly
    - TCP: Ports, retransmission, checksum, byte stream

FoxNet

- Idea: See what happens when each layer is a separate SML structure.
  - Is there a common interface that all layers can expect from the lower layer and export to the next layer?

```
structure Eth = struct .... end
structure Ip   = struct ... Eth.send ... end
structure Tcp = struct ... Ip.send ... end
```

FoxNet

- Better idea: Code layers as SML *functors*.
  - Order of layers not hard-coded into protocols

```
structure Eth = struct ... end
functor MakeIp(structure Lower : PROTOCOL) =
    struct ...  Lower.send ...  end
functor MakeTcp(structure Lower : PROTOCOL) =
    struct ...  Lower.send ...  end

structure Ip = MakeIp(structure Lower = Eth)
structure Tcp = MakeTcp(structure Lower = Ip)
```

The **`PROTOCOL`** Signature

- What should the PROTOCOL signature contain?
  - Want to be as generic as possible to allow maximum flexibility in layering.

```
structure TcpOverEth =
   MakeTcp(structure Lower = Eth)
```

A first attempt

```
signature PROTOCOL = sig
   type address
   type data
   val send    : address * data -> unit
   val receive : address -> data
end
```

- Overly naive

Supporting Connections

```
signature PROTOCOL = sig
   type address
   type data
   type connection
   val open    : address -> connection
   val send    : connection * data -> unit
   val receive : connection -> data
   val close   : connection -> unit
end
```

- Can we optimize multiple sends/receives?

Staged Functions

```
signature PROTOCOL = sig
   type address
   type data
   type connection
   val open    : address -> connection
   val send    : connection -> (data -> unit)
   val receive : connection -> (unit -> data)
   val close   : connection -> unit
end
```

- But user always applies functions immediately

Pre-applied Functions

```
signature PROTOCOL = sig
   type address
   type data
   type connection = {send   : data -> unit,
                      receive: unit -> data,
                      close  : unit -> unit}
   val open    : address -> connection
end
```

Automated **`open`** and **`close`**

```
signature PROTOCOL = sig
   type address
   type data
   type connection = {send   : data -> unit,
                      receive: unit -> data}
   val connect:
      address -> (connection -> unit) -> unit
end
```

Upcalls

- In this design, incoming data must be buffered until **`receive`** is called.
- Alternative design: specify a function to be called every time data arrives.
  - Called an "upcall" because system makes call to application code, rather than vice-versa
  - Function may choose to buffer data, or immediately send reply, or other options

Upcalls for data

```
signature PROTOCOL = sig
   type address
   type data
   type connection = {send : data -> unit}
   type handlers =
      {main_thread : connection -> unit,
       data_hander : connection * data -> unit}

   val connect:
      address -> handlers -> unit
end
```

Other Issues

- Passive connections
  - Install handler function for external connections.
- TCP
  - Definition refers to some IP details (checksum)
  - Specialized **`IPLIKE_PROTOCOL`** to expose these

```
functor MakeTcp
  (structure Lower : IPLIKE_PROTOCOL) = ...
```

  - TCP over Ethernet requires glue code
- Other layers: ARP, ICMP
- Other optimizations: TCP checksums, etc.