Continuation Semantics

Fake imperative programming using CPS, dictionaries, and datatypes

15-150 M21

Lecture 0806
06 August 2021

Acknowledgements

In this lecture, I use a lot of code and ideas developed by others.

- Red/Black Trees for dictionaries: this semester's implementation was adapted from the spring 2020 version, by Mike Erdmann & Frank Pfenning
- Monadic Parser Combinators: core parser code by Matthew McQuaid, for the course 98-317 (spring 2020)
- Continuation Semantics for while programs: I based my code off of notes and lectures by Steve Brookes for the course 15-314/812 (spring 2020)

Another crazy idea:
Implement a programming language in SML


while
c1;c2
skip
i:= i + 1
fn
datatype
k:int->'a
Dict.dict

Demonstration:

FC

In your terminal shell:

```
make repl
Standard ML of New Jersey v110...
...
[New bindings added.]
- FC.Runfile "programs/filename.fc";
```

```
- FC.RunfileWith "programs/filename.fc"
                 [("bvarname", FC.BOOL true),
                 ("ivarname", FC.INT 5)];
```

It takes a couple steps to do this.

1. Represent the FC code in a syntax SML can understand
2. Design a mechanism for how to mimic mutable state in SML
3. Write (CPS!) functions which "run" the SML representation of the FC code

The first step is more involved (and sophisticated) than we can get into here, so we'll mainly focus on the latter two steps.

- `cExp.sml` – the SML syntax of FC
- `FC.sml` – the core logic
- `programs/*.fc` – example files (written in FC)
- `sources.cm`
- `lib`
  - `parse.sml` – code for parsing FC to its SML representation (here be ~~dragons~~ monads)
  - `DICT.sig` & `dict.sml` – code for dictionaries (0716)

0 Representing FC programs in SML

The three expression types

We represent FC programs using three `datatype`s:

- `cExp`: represents commands. The program as a whole is represented by a value of type `cExp`. These are built up from some basic commands via various operations.
- `iExp`: represents integer expressions, which could be a variable name, an integer constant, or various arithmetic combinations of other integer expressions.
- `bExp`: represents boolean expressions, which could be a variable name, a boolean constant, boolean operations on other boolean expressions, or comparisons between integer expressions.

0806.0 (cExp.sml)

```
4 datatype iExp = iVAR of string
5               | CONST of int
6               | PLUS of iExp * iExp
7               | TIMES of iExp * iExp
8               | NEG of iExp
9               | DIV of iExp * iExp
```

0806.1 (cExp.sml)

```
12 datatype bExp = bVAR of string
13               | TRUE
14               | FALSE
15               | EQ of iExp * iExp
16               | LT of iExp * iExp
17               | GT of iExp * iExp
18               | AND of bExp * bExp
19               | NOT of bExp
20               | OR of bExp * bExp
```

0806.2 (cExp.sml)

```
23 datatype cExp = SKIP
24              | ABORT
25              | ASSIGNB of string * bExp
26              | ASSIGNI of string * iExp
27              | THEN of cExp * cExp
28              | IFTHENELSE of bExp* cExp *cExp
29              | WHILE of bExp * cExp
30              | PRINT of iExp
31              | ASSERT of bExp
32              | RETURN of iExp
```

We've written some code which

1. Reads the `.fc` file
2. Builds a single value of type `cExp` representing the code

```
(*Accepts a string of FC code and parses it *)
val fcParser.parse
  : string ->(cExp -> 'a) -> (unit -> 'a) ->'a
(*Accepts a filename and reads FC code in it*)
val fcParser.fileParse
  : string ->(cExp -> 'a) -> (unit -> 'a) ->'a
val fcParser.showParse : string -> cExp
```

Note: The parser is currently somewhat buggy. I'm working on improving it.

Note: Parsing is a really interesting topic. Learn more about it if you get the chance!

Steps

✓ Represent the FC code in a syntax SML can understand

2 Design a mechanism for how to mimic mutable state in SML

3 Write (CPS!) functions which "run" the SML representation of the FC code

1 Representing States as Dictionaries

Recall:
Dictionaries

0806.3 (lib/DICT.sig)

```
21 signature DICT =
22 sig
23   structure Key : EQ
24
25   type 'a entry = Key.t * 'a
26   type 'a dict
27
28   val empty : 'a dict
29
30   exception ExistingEntry
31   val insert : 'a entry * 'a dict -> 'a dict
32   val overwrite : 'a entry * 'a dict -> 'a dict
33   val lookup : 'a dict -> Key.t -> 'a option
34 end
```

0806.4 (lib/dict.sml)

```
13 structure StringOrd : ORD =
14 struct
15   type t = string
16   val cmp = String.compare
17 end
18
19 functor RBDict (KeyOrd : ORD) :> DICT
20   where type Key.t = KeyOrd.t =
21 struct
22   structure Key = cmpEqual(KeyOrd)
23   type 'a entry = Key.t * 'a
```

```
structure Dict = RBDict(StringOrd)
```

We want to simulate a “mutable state”, where variables are set to certain values and can be modified later.

Idea:

Pass a dictionary around

We want to simulate a "mutable state", where variables are set to certain values and can be modified later.

- All our functions will take in a dictionary as an argument, representing the "current state"
- Set a variable `x` to `v : t` by putting

```
val D' = Dict.overwrite D ("x",v)
```

  and then using `D'` as the state from then on (e.g. passing to other functions)
- Query the current value of `x` by putting

```
val xVal = Dict.lookup D "x"
```

  If `xVal` is `SOME v` then `x` is currently set to `v`. If `xVal` is `NONE`, then `x` is currently unbound.

0806.5 (FC.sml)

```
8 datatype var = BOOL of bool | INT of int
```

So a `var Dict.dict` stores booleans and integers, tagged with their types.

- If `Dict.lookup D "x"` is `SOME(BOOL b)`, then `x` is set a boolean-valued variable, whose value is currently `b`.
- If `Dict.lookup D "x"` is `SOME(INT n)`, then `x` is an integer-valued variable whose current value is `n`.

Steps

- ✓ Represent the FC code in a syntax SML can understand
- ✓ Design a mechanism for how to mimic mutable state in SML
- 3 Write (CPS!) functions which "run" the SML representation of the FC code

5 Minute Break

2 Execution

0806.6 (FC.sml)

```
12 datatype Type = Bool | Int
13 datatype error =
14       TypeError of string * Type * Type
15     | UnboundVar of string
16     | AssertionError
17     | Abort
18     | DivZero
19     | NoReturn
```

```
interpret : cExp
          -> (error -> 'a)
          -> (int -> 'a)
          -> 'a
```

REQUIRES: true
ENSURES:

`interpret input panic success`

evaluates to `success(n)` if executing the command `input` returns `n`. If executing `input` encounters an error `e`, then

`interpret input panic success` $\Longrightarrow$ `panic e`.

```
fun interpret input panic success =
  let
    fun evalB (D:var Dict.dict) (b:bExp)
              (k:bool -> 'a) : 'a = ...
    fun evalI (D:var Dict.dict) (e:iExp)
              (k:int -> 'a) : 'a = ...
    fun exec  (D:var Dict.dict) (c:cExp)
              (k:var Dict.dict -> 'a):'a
              = ...
```

evalI is CPS to the core!

0806.7 (FC.sml)

```
26 fun evalI (D : var Dict.dict)
27           (e : iExp) (k:int -> 'a) =
28   case e of
29     (CONST n) => k n
30   | (PLUS(e1,e2)) =>
31       evalI D e1 (fn v1 =>
32       evalI D e2 (fn v2 =>
33       k(v1+v2)))
34   | (TIMES(e1,e2)) =>
35       evalI D e1 (fn v1 =>
36       evalI D e2 (fn v2 =>
37       k(v1*v2)))
```

evalI is CPS to the core!

0806.8 (FC.sml)

```
40  | (NEG(e')) =>
41      evalI D e' (fn v => k(~v))
42  | (DIV(e1,e2)) =>
43      evalI D e2 (fn 0 => panic DivZero
44                   | v2 => evalI D e1 (fn v1=>
45                             k(v1 div v2)))
46  | (iVAR i) =>
47      (case (Dict.lookup D i) of
48         (SOME(INT v)) =>  k v
49       | (SOME _) =>
50            panic (TypeError (i,Int,Bool))
51       | NONE =>
52            panic (UnboundVar i))
```

and so is evalB!

0806.9 (FC.sml)

```
55 fun evalB (D:var Dict.dict)
56          (b:bExp) (k:bool->'a) =
57   case b of
58     TRUE => k true
59   | FALSE => k false
60   | (EQ(e1,e2)) =>
61          evalI D e1 (fn v1 =>
62          evalI D e2 (fn v2 =>
63          k(v1=v2)))
64   | (LT(e1,e2)) =>
65          evalI D e1 (fn v1 =>
66          evalI D e2 (fn v2 =>
67          k(v1<v2)))
   | (GT(e1,e2)) =>
```

and so is evalB!

0806.10 (FC.sml)

```
78  | (OR(b1,b2)) =>
79        evalB D b1 (fn v1 =>
80        evalB D b2 (fn v2 =>
81        k(v1 orelse v2)))
82  | (NOT(b')) =>
83        evalB D b' (fn v => k (not v))
84  | (bVAR(i)) =>
85      (case (Dict.lookup D i) of
86         (SOME(BOOL v)) => k v
87       | (SOME(INT _)) =>
88            panic (TypeError(i,Bool,Int))
89       | NONE =>
90            panic (UnboundVar i))
```

Note:

Hard-coding syntactic sugar in parser

0806.11 (FC.sml)

```
94  fun exec (D : var Dict.dict) (c : cExp)
95           (k : var Dict.dict -> 'a) : 'a =
96    case c of
97      SKIP => k D
98    | ABORT => panic Abort
99    | (ASSIGNB(s,b)) =>
100       evalB D b (fn vb =>
101       k (Dict.overwrite((s,BOOL vb),D)))
102   | (ASSIGNI(i,e)) =>
103       evalI D e (fn v =>
104       k (Dict.overwrite((i,INT  v),D)))
```

0806.12 (FC.sml)

```
107   | (THEN(c1,c2)) =>
108       exec D c1 (fn D' =>
109       exec D' c2 k)
110   | (IFTHENELSE(b,c1,c2)) =>
111      evalB D b
112        (fn true => exec D c1 k
113         | false => exec D c2 k)
114   | (WHILE(b,c')) =>
115      evalB D b
116        (fn true => exec D c' (fn D' =>
117                    exec D' c k)
118         | false => k D)
```

0806.13 (FC.sml)

```
121   | (RETURN e) => evalI D e success
122   | (ASSERT b) => evalB D b
123                     (fn true => k D
124                       | false => panic AssertionError)
125   | (PRINT e) =>
126       evalI D e
127        (fn v => (print((Int.toString v) ^"\n");
128                  k D))
```

How to interpret

```
fun interpret input panic success =
  let
    fun evalB (D:var Dict.dict) (b:bExp)
              (k:bool -> 'a) : 'a = ...
    fun evalI (D:var Dict.dict) (e:iExp)
              (k:int -> 'a) : 'a = ...
    fun exec  (D:var Dict.dict) (c:cExp)
              (k:var Dict.dict -> 'a):'a
             = ...
              (* calls success to return*)
  in
    exec (Dict.empty) input
      (fn _ => panic NoReturn)
  end
```

```
fun interpretWith initDict input panic success =
  let
    fun evalB (D:var Dict.dict) (b:bExp)
              (k:bool -> 'a) : 'a = ...
    fun evalI (D:var Dict.dict) (e:iExp)
              (k:int -> 'a) : 'a = ...
    fun exec  (D:var Dict.dict) (c:cExp)
              (k:var Dict.dict -> 'a):'a
             = ...
              (* calls success to return*)
  in
    exec initDict input
      (fn _ => panic NoReturn)
  end
val interpret = interpretWith Dict.empty
```

- More types than just int and bool
- More constructs, more sugar
- More language features

Thank you!