CS442 Assignment 3

Nissan Pow
20187246
npow

March 9, 2007

Part A

```
exception NOMORE

abstype 'a set = Set of unit -> unit -> 'a
with
  fun newset L =
    let
      fun foo t () =
        let
          val tail = t
        in
          if null (!tail) then raise NOMORE
          else
            let
              val head = ref (hd (!tail))
            in
              (tail := tl (!tail); !head)
            end
        end
    in
      (Set (fn () => foo (ref L)))
    end

  fun mknext (Set f) =
    f ()

  fun member e S =
    let
      val iter_S = mknext S
      fun check () =
        e = iter_S () orelse check () handle NOMORE => false
    in
      check ()
    end
```

```
fun empty S =
  let
    fun check _ = false
  in
    check (mknext S ()) handle NOMORE => true
  end

fun intersect S T =
  let
    val iter_S = mknext S
    fun check () =
      let
        val head = iter_S ()
      in
        if member head T handle NOMORE => false then head
        else check ()
      end
  in
    (Set (fn () => fn () => check ()))
  end

fun diff S T =
  let
    val iter_S = mknext S
    fun check () =
      let
        val head = iter_S ()
      in
        if not (member head T) handle NOMORE => false then head
        else check ()
      end
  in
    (Set (fn () => fn () => check ()))
  end

fun union S T =
  let
    fun u2 S T =
      let
        val iter_S = mknext S
        val iter_T = mknext T
      in
        (Set (fn () => fn () => iter_S () handle NOMORE => iter_T ()))
      end
  in
    diff (u2 S T) (intersect S T)
```

```
    end

  fun forall S f =
    let
      val iter_S = mknext (!S)
    in
      S := (Set (fn () => fn () => f (iter_S ())))
    end
end

(* some testcases
fun times2 x = x * 2;
val setA = newset [1,2,3];
val seqA = mknext setA;
val refA = ref setA;
forall refA times2;
val seqB = mknext (!refA);
val setB = newset [1,2,3]
val setC = newset [3,5,6]
val unionAB = union setA setB
val seqAB = mknext unionAB

val intAB = intersect setA setB
*)
```

Discussion

1. Yes it can be done. We need to make a new datatype to contain strings and integers, like so:

   ```
   datatype StringOrInt = Int of int | String of string;
   ```

   Then we can do stuff like: val myList = [Int 5, String "hi", Int 10, String "bye"];
   newset myList

2. No this cannot be done. Although we can create a list of the polymorphic functions, ML will not allow us to create a set of polymorphic functions due to the value restriction.

Part B

```
signature NFASIG = sig
  eqtype Q
  eqtype Sigma
  val init : Q
  val delta : Q * Sigma -> Q list
  val accepting : Q list
end

functor NFA (Spec : NFASIG) : sig
```

```
  val run : Spec.Sigma list -> bool
end =
struct
  fun run l =
    let
      fun run' s [] = s
      |   run' s (x::xs) =
            run' (foldl (fn (state, i) => (Spec.delta(state, x)) @ i) nil s) xs

      val final = run' [Spec.init] l

      fun member _ [] = false
      |   member x (y::ys) = x = y orelse member x ys

      fun intersect x y =
        foldl (fn (a,b) => a orelse b) false (map (fn a => member a y) x)
    in
      intersect final Spec.accepting
    end
end

signature DFASIG = sig
  eqtype Q
  eqtype Sigma
  val init : Q
  val delta : Q * Sigma -> Q
  val accepting : Q list
end

functor DFA (Spec : DFASIG) : sig
  val run : Spec.Sigma list -> bool
end =
struct
  fun run l =
    let
      fun run' s [] = s
      |   run' s (x::xs) = run' (Spec.delta(s, x)) xs

      val final = run' Spec.init l

      fun member _ [] = false
      |   member x (y::ys) = x = y orelse member x ys
    in
      member final Spec.accepting
    end
end
```

```
functor DFAtoNFA (Spec : DFASIG) : NFASIG =
struct
  type Q = Spec.Q
  type Sigma = Spec.Sigma
  val init = Spec.init
  fun delta (state, x) = [Spec.delta (state, x)]
  val accepting = Spec.accepting
end

(* NFA to recognize binary strings that end in zero *)
structure EndsInZero_NFA : NFASIG =
struct
  exception Bad
  type Sigma = int
  datatype Q = A | B | C
  val init = A
  val accepting = [B]

  fun delta (A, 1) = [A]
  |   delta (A, 0) = [B,C]
  |   delta (B, 1) = [A]
  |   delta (B, 0) = [B]
  |   delta (C, 1) = [A]
  |   delta (C, 0) = [C]
  |   delta _ = raise Bad
end

(* DFA to recognize binary strings with an even number of zeros
   and an odd number of ones *)
structure Even0sOdd1s_DFA : DFASIG =
struct
  exception Bad
  type Sigma = int
  datatype QAux = ODD | EVEN
  type Q = QAux * QAux
  val init = (EVEN, EVEN)
  val accepting = [(EVEN, ODD)]

  fun toggle ODD = EVEN
  |   toggle _ = ODD

  fun delta ((x,y), 0) = (toggle x, y)
  |   delta ((x,y) , 1) = (x, toggle y)
  |   delta _ = raise Bad
end

(* computes cartesian product of lists A and B *)
```

```
fun cartprod A B =
  foldr (fn (a,b) => (foldr (fn (c,d) => (a,c) :: d) nil B) @ b) nil A

functor Intersection (structure Spec1 : DFASIG
                              and Spec2 : DFASIG
(*                          sharing type Spec1.Q = Spec2.Q*)
                        sharing type Spec1.Sigma = Spec2.Sigma) : DFASIG =
struct
  type Q = Spec1.Q * Spec2.Q
  type Sigma = Spec1.Sigma
  val init = (Spec1.init, Spec2.init)
  fun delta ((s1, s2), x) = (Spec1.delta(s1, x), Spec2.delta(s2, x))
  val accepting = cartprod Spec1.accepting Spec2.accepting
end

(* DFA to recognize binary strings with an even number of zeros *)
structure Even0s_DFA : DFASIG =
struct
  exception Bad
  type Sigma = int
  datatype Q = EVEN | ODD | START
  val init = START
  val accepting = [EVEN]

  fun delta (START, 1) = EVEN
  |   delta (START, 0) = ODD
  |   delta (ODD, 1) = ODD
  |   delta (ODD, 0) = EVEN
  |   delta (EVEN, 1) = EVEN
  |   delta (EVEN, 0) = ODD
  |   delta _ = raise Bad
end

structure foo = Intersection (structure Spec1=Even0s_DFA and Spec2=Even0sOdd1s_DFA)
structure fooInterp = DFA(foo)
```

Discussion

1. The problem is that with intersection, we enforce the alphabets of the 2 DFAs to be the same, but with union they may be different.

Transcript

```
- structure even = DFA(Even0s_DFA);
structure even : sig val run : Spec.Sigma list -> bool end
- even.run [1,1,1];
GC #0.0.0.1.4.55:   (0 ms)
val it = true : bool
```

```
- even.run [1,0];
val it = false : bool
- even.run [0,0,1,1,0,0,1];
val it = true : bool


- structure s = NFA(EndsInZero_NFA);
structure s : sig val run : Spec.Sigma list -> bool end
- s.run [1,1,1];
val it = false : bool
- s.run [1,1,1,0];
val it = true : bool
- s.run [1,0,1,0,1];
val it = false : bool
- s.run [1,0,1,1,1,0,0,0];
val it = true : bool
- s.run [1,0,1,1,1,0,0,0,1,0,0,0,1,0,1,0,1,0];
val it = true : bool


- structure foo = Intersection (structure Spec1=Even0s_DFA and Spec2=Even0sOdd1s_DFA);
structure foo : DFASIG
- structure fooInterp = DFA(foo);
structure fooInterp : sig val run : Spec.Sigma list -> bool end
- fooInterp.run [0,0,0,0,0];
val it = false : bool
- fooInterp.run[0,0,1,1,0,1];
val it = false : bool
- fooInterp.run[0,0,0,1,1,1,0];
val it = true : bool
- fooInterp.run[1,1,1];
GC #0.0.0.1.4.61:   (0 ms)
val it = true : bool
- fooInterp.run[0,1];
val it = false : bool
- fooInterp.run[1,0,0];
val it = true : bool
```

Part C

```
data Term = Var String | Abs String Term | App Term Term | Prim Primitive | INT Int
data Primitive = IsZero | Succ deriving Show

data SContents = Stack String Term [EContents] | StackTerm Term deriving Show
data EContents = Env String SContents deriving Show
data CContents = Control Term | Apply deriving Show
```

```
data DContents = Dump [SContents] [EContents] [CContents] deriving Show
data SECDConfig = SECD ([SContents], [EContents], [CContents], [DContents])

-- make Term printable in readable format
instance Show Term where
  show (Var x) = x
  show (Abs x e) = "\\" ++ x ++ "." ++ show e
  show (App m n) = "(" ++ show m ++ ")(" ++ show n ++ ")"
  show (Prim p) = show p
  show (INT p) = show p

-- print contents of SECD machine
instance Show SECDConfig where
  show (SECD (s,e,c,d)) =
    "S = " ++ show s ++ "\nE = " ++ show e ++ "\nC = " ++ show c ++ "\nD = " ++ show d

-- lookup variable in environment; no free variables allowed
lookUp :: (String,[EContents]) -> SContents
lookUp (var,(Env v s):envs)
  | var==v = s
  | otherwise = lookUp(var,envs)

-- maps one step of the SECD configuration to the one that immediately follows it
secdOneStep :: SECDConfig -> SECDConfig

secdOneStep (SECD (s, e, (Control (Var x)):c', d))
  = SECD (lookUp(x,e):s, e, c', d)

secdOneStep (SECD (s, e, (Control (Abs x m)):c', d))
  = SECD ((Stack x m e):s, e, c', d)

secdOneStep (SECD (s, e, (Control (App m n)):c', d))
  = SECD (s, e, (Control n):(Control m):Apply:c', d)

secdOneStep (SECD ((StackTerm (Prim prim)):(StackTerm n):s', e, Apply:c', d))
  = SECD (s', e, (Control $ applyPrim prim n):c', d)

secdOneStep (SECD ((Stack x m e1):n:s', e, Apply:c', d))
  = SECD ([], (Env x n):e1, [Control m], (Dump s' e c'):d)

secdOneStep (SECD ([m], e, [], (Dump s' e' c'):d'))
  = SECD (m:s', e', c', d')

secdOneStep (SECD ([m], e, [], []))
  = SECD ([m], e, [], [])

-- handle primitives and integers
```

```
secdOneStep (SECD (s, e, (Control (INT i)):c', d))
  = SECD ((StackTerm (INT i)):s, e, c', d)

secdOneStep (SECD (s, e, (Control (Prim p)):c', d))
  = SECD ((StackTerm (Prim p)):s, e, c', d)

applyPrim :: Primitive -> Term -> Term

applyPrim IsZero (INT i) = (Abs "x" (Abs "y" (Var (if i==0 then "x" else "y"))))
applyPrim Succ (INT i) = (INT (i+1))

get :: SContents -> Term
get (StackTerm t) = t
get (Stack x t e) = foldl subst (Abs x t) e

-- performs lambda calculus substitution as described in notes
subst :: Term -> EContents -> Term
subst (Var y) (Env x t)
    | x == y    = (get t)
    | otherwise = Var y
subst (Abs y m) (Env x t)
    | x == y    = Abs x m
    | otherwise = Abs y (subst m (Env x t))
subst (App m n) (Env x t) =
    App (subst m (Env x t)) (subst n (Env x t))

reduce :: Term -> Term
reduce x
  = get $ stack $ r1 (SECD ([], [], [Control x], []))
    where
      r1 (SECD (s,e,[],[])) = SECD (s,e,[],[])
      r1 (SECD (s,e,c,d)) = r1 (secdOneStep (SECD (s,e,c,d)))
      stack (SECD ([s],_,_,_)) = s

s1 = secdOneStep

-- Test Cases
lc_true = Abs "x" (Abs "y" (Var "x"))
lc_false = Abs "x" (Abs "y" (Var "y"))
lc_not = Abs "b" (App (App (Var "b") lc_false) lc_true)
lc_iff = Abs "b" (Abs "t" (Abs "f" (App (App (Var "b") (Var "t")) (Var "f"))))
lc_cons = Abs "h" (Abs "t" (Abs "s" (App (App (Var "s") (Var "h")) (Var "t"))))
lc_car = Abs "l" (App (Var "l") lc_true)
lc_cdr = Abs "l" (App (Var "l") lc_false)
lc_nil = Abs "s" lc_true
lc_null = Abs "l" (App (Var "l") (Abs "h" (Abs "t" lc_false)))
```

```
cn_0 = (Abs "f" (Abs "z" (Var "z")))
cn_I = (Abs "f" (Abs "z" (App (Var "f") (Var "z"))))
cn_II = (Abs "f" (Abs "z" (App (Var "f") (App (Var "f") (Var "z")))))
cn_III = (Abs "f" (Abs "z" (App (Var "f") (App (Var "f") (App (Var "f") (Var "z"))))))
cn_plus = (Abs "m" (Abs "n" (Abs "s"
            (Abs "z" (App (App (Var "m") (Var "s"))
            (App (App (Var "n") (Var "s")) (Var "z")))))))
cn_mult = (Abs "m" (Abs "n" (Abs "f" (App (Var "n") (App (Var "m") (Var "f"))))))
cn_pow = (Abs "m" (Abs "n" (App (Var "n") (Var "m"))))
cn_zero = (Abs "m" (App (App (Var "m") (Abs "x" lc_false)) lc_true))
cn_zz = (App (App lc_cons cn_0) cn_0)
cn_ss = (Abs "p" (App (App lc_cons (App lc_cdr (Var "p")))
                      (App (App cn_plus cn_I) (App lc_cdr (Var "p")))))
cn_prd = (Abs "m" (App lc_car (App (App (Var "m") cn_ss) cn_zz)))
cn_sub = (Abs "m" (Abs "n" (App (App (Var "n") cn_prd) (Var "m"))))

-- converts church numeral to decimal format
to_Int = Abs "n" (App (App (Var "n") (Prim Succ)) (INT 0))

e1 = App to_Int (App (App cn_sub cn_III) cn_II)
e2 = App to_Int (App (App cn_mult cn_III) cn_III)
e3 = App to_Int (App (App cn_pow cn_II) cn_III)
e4 = App (Prim Succ) (INT 0)
```

Transcript

```
Main> reduce (App to_Int (App (App cn_sub cn_III) cn_II))
1
Main> reduce (App to_Int (App (App cn_mult cn_III) cn_III))
9
Main> reduce (App to_Int (App (App cn_pow cn_II) cn_III))
8
Main> reduce (App (Prim Succ) (INT 0))
1
Main> reduce (App (Prim IsZero) (INT 0))
\x.\y.x
Main> reduce (App (Prim IsZero) (INT 1))
\x.\y.y
Main> reduce (App to_Int (App (App cn_pow (App (App cn_pow cn_II) cn_III)) cn_II))
64
```