1: (* lambda2anf.sml
2: *
3: * Conversion from straight Lambda to ANF.
4: *
5: * Copyright (c) 2005 by Matthias Blume (blume@tti-c.org)
6: *)
7: structure LambdaToANF : sig
8:  
9:   val convert : Lambda.function -> ANF.function
10: end = struct
11:  
12:     structure L = Lambda
13:     structure A = ANF
14:     
15:     fun convert f =
16:       let
17:         fun tmpvar NONE = LVar.new "tmp"
18:         | tmpvar (SOME v) = LVar.clone v
19:       fun ijump (f, x) = A.JUMP (Purity.Impure, (A.VAR f, [x]))
20:       fun exp (L.VALUE v, _, k) = k v
21:         | exp (L.LET (v, e, b), v0, k) =
22:             exp (e, SOME v, fn x =>
23:                 A.BIND (v, x, exp (b, v0, k)))
24:         | exp (L.FIX (fl, b), v0, k) =
25:             A.FIX (map function fl, exp (b, v0, k))
26:         | exp (L.ARITH (aop, e1, e2), v0, k) =
27:           let
28:             val v = tmpvar v0
29:             in
30:                 exp (e1, NONE,
31:                     fn x1 =>
32:                         exp (e2, NONE,
33:                             fn x2 =>
34:                                 A.ARITH (aop, x1, x2, v, k (A.VAR v))))
35:           end
36:         | exp (L.RECORD (m, el), v0, k) =
37:           let
38:             val v = tmpvar v0
39:             in
40:                 explist (el,
41:                     fn xl =>
42:                         A.RECORD (m, xl, v, k (A.VAR v)))
43:             end
44:         | exp (L.SELECT (e, i, m), v0, k) =
45:           let
46:             val v = tmpvar v0
47:             in
48:                 exp (e, NONE,
49:                     fn x =>
50:                         A.SELECT (x, i, m, v, k (A.VAR v)))
51:             end
52:         | exp (L.UPDATE (e1, i, e2), _, k) =
53:           let
54:             val v = tmpvar v0
55:             in
56:                 exp (e1, NONE,
57:                     fn x1 =>
58:                         exp (e2, NONE,
59:                             fn x2 =>
60:                                 A.UPDATE (x1, i, x2, k (A.INT 0))))
61:             end
62:         | exp (L.CMP (cop, e1, e2, et, ef), v0, k) =
63:           let
64:             val f = tmpvar NONE
65:             val v = tmpvar v0
66:             val et’ = exp (et, v0, fn x => ijump (f, x))
67:             val ef’ = exp (ef, v0, fn x => ijump (f, x))
68:             in
69:                 exp (e1, NONE,
70:                     fn x1 =>
71:                         exp (e2, NONE,
72:                             fn x2 =>
73:                                 A.FIX ([{ f = (f, [v], k (A.VAR v) ) }, inl = false ],
74:                                     A.CMP (cop, x1, x2, et’, ef’))])
75:             end
76:         | exp (L.APP (ta, e, el), v0, k) =
77:           let
78:             val v = tmpvar v0
79:             in
80:                 exp (e, NONE,
81:                     fn x =>
82:                         explist (el,
83:                             fn xl =>
84:                                 A.CALL (ta, [v], (x, xl), k (A.VAR v))))
85:             end
86:     end
87:  
88:     and exp’ (L.LET (v, e, b)) =
89:       exp (e, SOME v, fn x => A.BIND (v, x, exp’ b))
90:     | exp’ (L.FIX (fl, b)) =
A.FIX (map function fl, exp' b) =
| exp' (L.CMP (cop, e1, e2, et, ef)) =
  exp (e1, NONE, fn x1 =>
    exp (e2, NONE, fn x2 =>
      A.CMP (cop, x1, x2, exp' et, exp' ef)))
| exp' (L.APP (ta, e, el)) =
  exp (e, NONE, fn x =>
    explist (el, fn x1 =>
      A.JUMP (ta, (x, x1))))
| exp' e = exp (e, NONE, fn x => A.VALUES [x])
and explist ([], k) = k []
| explist (e :: el, k) =
  exp (e, NONE, fn x => explist (el, fn x1 => k (x :: x1)))
and function (f, vl, b, inl) = { f = (f, vl, exp' b), inl = inl }
in function f
end