Start of Lecture 9: RSL: APPLICATIVE CONSTRUCTS

A.5. Other Applicative Expressions

A.5.1. Simple let Expressions

Simple (i.e., nonrecursive) **let** expressions:

let a = $\mathcal{E}_d$ **in** $\mathcal{E}_b$(a) **end**

is an "expanded" form of:

$(\lambda a.\mathcal{E}_b(a))(\mathcal{E}_d)$

A.5.2. Recursive let Expressions

Recursive **let** expressions are written as:

let f = λa·E(f,a) **in** B(f,a) **end**
let f = (λgλa·E(g,a))(f) **in** B(f.a) **end**
let f = F(f) **in** E(f,a) **end where** F ≡ λgλa·E(g,a)
let f = **Y**F **in** B(f,a) **end where Y**F = F(**Y**F)

A.5.3. Non-deterministic let Clause

- The non-deterministic **let** clause:

let a:A · $\mathcal{P}$(a) **in** $\mathcal{B}$(a) **end**

- expresses the non-deterministic selection of a value a of type A
- which satisfies a predicate $\mathcal{P}$(a) for evaluation in the body $\mathcal{B}$(a).
- If no a:A • P(a) the clause evaluates to **chaos**.

A.5.4. Pattern and "Wild Card" let Expressions

Patterns and *wild cards* can be used:

let {a} ∪ s = set **in** ... **end**
let {a,_} ∪ s = set **in** ... **end**

let (a,b,...,c) = cart **in** ... **end**
let (a,_,...,c) = cart **in** ... **end**

let $\langle a \rangle \widehat{\ } \ell$ = list **in** ... **end**
let $\langle a,\_,b \rangle \widehat{\ } \ell$ = list **in** ... **end**

let [a↦b] ∪ m = map **in** ... **end**
let [a↦b,_] ∪ m = map **in** ... **end**

A.5.5. Conditionals

if b_expr **then** c_expr **else** a_expr
end

if b_expr **then** c_expr **end** ≡ /∗ same as: ∗/
  if b_expr **then** c_expr **else** **skip** **end**

if b_expr_1 **then** c_expr_1
elsif b_expr_2 **then** c_expr_2
elsif b_expr_3 **then** c_expr_3
...
elsif b_expr_n **then** c_expr_n **end**

case expr **of**
  choice_pattern_1 → expr_1,
  choice_pattern_2 → expr_2,
  ...
  choice_pattern_n_or_wild_card → expr_n **end**

Example 44 – Choice Pattern Case Expressions: Insert Links:
We consider the meaning of the Insert operation designators.

21. The insert operation takes an Insert command and a net and yields either a new net or **chaos** for the case where the insertion command "is at odds" with, that is, is not semantically well-formed with respect to the net.
22. We characterise the "is not at odds", i.e., is semantically well-formed, that is:

    - pre_int_Insert(op)(hs,ls),

    as follows: it is a propositional function which applies to Insert actions, op, and nets, (hs.ls), and yields a truth value if the below relation between the command arguments and the net is satisfied. Let (hs,ls) be a value of type N.

23. If the command is of the form 2oldH(hi′,l,hi′) then
    - ⋆1 hi′ must be the identifier of a hub in hs,
    - ⋆s2 l must not be in ls and its identifier must (also) not be observable in ls, and
    - ⋆3 hi″ must be the identifier of a(nother) hub in hs.
24. If the command is of the form 1oldH1newH(hi,l,h) then
    - ⋆1 hi must be the identifier of a hub in hs,
    - ⋆2 l must not be in ls and its identifier must (also) not be observable in ls, and
    - ⋆3 h must not be in hs and its identifier must (also) not be observable in hs.

25. If the command is of the form 2newH(h′,l,h″) then
    - $\star 1$ h′ — left to the reader as an exercise (see formalisation !),
    - $\star 2$ l — left to the reader as an exercise (see formalisation !), and
    - $\star 3$ h″ — left to the reader as an exercise (see formalisation !).

Conditions concerning the new link (second $\star$s, $\star 2$, in the above three cases) can be expressed independent of the insert command category.

value

21 int_Insert: Insert $\rightarrow$ N $\xrightarrow{\sim}$ N

22′ pre_int_Insert: Ins $\rightarrow$ N $\rightarrow$ **Bool**

22″ pre_int_Insert(Ins(op))(hs,ls) $\equiv$

$\star 2$ s_l(op)$\notin$ ls $\wedge$ obs_LI(s_l(op)) $\notin$ iols(ls) $\wedge$

case op **of**

23) 2oldH(hi′,l,hi″) $\rightarrow$ {hi′,hi″}$\in$ iohs(hs),

24) 1oldH1newH(hi,l,h) $\rightarrow$

hi $\in$ iohs(hs) $\wedge$ h$\notin$ hs $\wedge$ obs_HI(h)$\notin$ iohs(hs),

25) 2newH(h′,l,h″) $\rightarrow$

{h′,h″}$\cap$ hs={} $\wedge$ {obs_HI(h′),obs_HI(h″)}$\cap$ iohs(hs)={}

end

26. Given a net, (hs,ls), and given a hub identifier, (hi), which can be observed from some hub in the net, xtr_H(hi)(hs,ls) extracts the hub with that identifier.
27. Given a net, (hs,ls), and given a link identifier, (li), which can be observed from some link in the net, xtr_L(li)(hs,ls) extracts the hub with that identifier.

value

26: xtr_H: HI $\rightarrow$ N $\xrightarrow{\sim}$ H

26: xtr_H(hi)(hs,_) $\equiv$ **let** h:H•h $\in$ hs $\wedge$ obs_HI(h)=hi **in** h **end**

pre hi $\in$ iohs(hs)

27: xtr_L: HI $\rightarrow$ N $\xrightarrow{\sim}$ H

27: xtr_L(li)(_,ls) $\equiv$ **let** l:L•l $\in$ ls $\wedge$ obs_LI(l)=li **in** l **end**

pre li $\in$ iols(ls)

28. When a new link is joined to an existing hub then the observable link identifiers of that hub must be updated to reflect the link identifier of the new link.
29. When an existing link is removed from a remaining hub then the observable link identifiers of that hub must be updated to reflect the removed link (identifier).

value

aLI: H $\times$ LI $\rightarrow$ H, rLI: H $\times$ LI $\xrightarrow{\sim}$ H

28: aLI(h,li) **as** h′

pre li $\notin$ obs_LIs(h)

post obs_LIs(h′) = {li} $\cup$ obs_LIs(h) $\wedge$ non_l_eq(h,h′)

29: rLI(h′,li) **as** h

pre li $\in$ obs_LIs(h′) $\wedge$ **card** obs_LIs(h′)$\geq$2

post obs_LIs(h) = obs_LIs(h′) $\setminus$ {li} $\wedge$ non_l_eq(h,h′)

30. If the Insert command is of kind 2newH(h',l,h'') then the updated net of hubs and links, has
    - the hubs hs joined, ∪, by the set {h′,h″} and
    - the links ls joined by the singleton set of {l}.
31. If the Insert command is of kind 1oldH1newH(hi,l,h) then the updated net of hubs and links, has

    31.1 : the hub identified by hi updated, hi′, to reflect the link connected to that hub.

    31.2 : The set of hubs has the hub identified by hi replaced by the updated hub hi′ and the new hub.

    31.2 : The set of links augmented by the new link.
32. If the Insert command is of kind 2oldH(hi',l,hi'') then

    32.1–.2 : the two connecting hubs are updated to reflect the new link,

    32.3 : and the resulting sets of hubs and links updated.


```
int_Insert(op)(hs,ls) ≡
⋆i   case op of
30      2newH(h′,l,h″) → (hs ∪ {h′,h″},ls ∪ {l}),
31      1oldH1newH(hi,l,h) →
31.1       let h′ = aLI(xtr_H(hi,hs),obs_LI(l)) in
31.2       (hs\{xtr_H(hi,hs)}∪{h,h′},ls ∪{l}) end,
32      2oldH(hi′,l,hi″) →
32.1       let hsδ = {aLI(xtr_H(hi′,hs),obs_LI(l)),
32.2                  aLI(xtr_H(hi″,hs),obs_LI(l))} in
32.3       (hs\{xtr_H(hi′,hs),xtr_H(hi″,hs)}∪ hsδ,ls ∪{l}) end
⋆j   end
⋆k   pre pre_int_Insert(op)(hs,ls)
```


33. The remove command is of the form Rmv(li) for some li.
34. We now sketch the meaning of removing a link:
    (a) The link identifier, li, is, by the pre_int_Remove pre-condition, that of a link, l, in the net.
    (b) That link connects to two hubs, let us refer to them as h′ and h′.
    (c) For each of these two hubs, say h, the following holds wrt. removal of their connecting link:
        i. If l is the only link connected to h then hub h is removed. This may mean that
            - either one
            - or two hubs

            are also removed when the link is removed.

        ii. If l is not the only link connected to h then the hub h is modified to reflect that it is no longer connected to l.
    (d) The resulting net is that of the pair of adjusted set of hubs and links.


```
value
 33 int_Remove: Rmv → N ~→ N
 34 int_Remove(Rmv(li))(hs,ls) ≡
 34(a))  let l = xtr_L(li)(ls), {hi′,hi″} = obs_HIs(l) in
 34(b))  let {h′,h″} = {xtr_H(hi′,hs),xtr_H(hi″,hs)} in
 34(c))  let hs′ = cond_rmv(h′,hs) ∪ cond_rmv_H(h″,hs) in
 34(d))  (hs\{h′,h″} ∪ hs′,ls\{l}) end end end
 34(a))  pre li ∈ iols(ls)

 cond_rmv: LI × H × H-set → H-set
 cond_rmv(li,h,hs) ≡
 34((c))i)   if obs_HIs(h)={li} then {}
 34((c))ii)   else {sLI(li,h)} end
 pre li ∈ obs_HIs(h)
```

■ End of Example 44

A.5.6. Operator/Operand Expressions

$$
\begin{array}{l}
\langle \text{Expr} \rangle ::= \\
\quad\quad \langle \text{Prefix\_Op} \rangle\ \langle \text{Expr} \rangle \\
\quad\ \mid \langle \text{Expr} \rangle\ \langle \text{Infix\_Op} \rangle\ \langle \text{Expr} \rangle \\
\quad\ \mid \langle \text{Expr} \rangle\ \langle \text{Suffix\_Op} \rangle \\
\quad\ \mid \ldots \\
\langle \text{Prefix\_Op} \rangle ::= \\
\quad\ - \mid \sim \mid \cup \mid \cap \mid \textbf{card} \mid \textbf{len} \mid \textbf{inds} \mid \textbf{elems} \mid \textbf{hd} \mid \textbf{tl} \mid \textbf{dom} \mid \textbf{rng} \\
\langle \text{Infix\_Op} \rangle ::= \\
\quad\quad = \mid \neq \mid \equiv \mid + \mid - \mid * \mid \uparrow \mid / \mid < \mid \leq \mid \geq \mid > \mid \wedge \mid \vee \mid \Rightarrow \\
\quad\ \mid \in \mid \notin \mid \cup \mid \cap \mid \setminus \mid \subset \mid \subseteq \mid \supseteq \mid \supset \mid \widehat{\ } \mid \dagger \mid \circ \\
\langle \text{Suffix\_Op} \rangle ::= !
\end{array}
$$

On a Triptych of Software Development 274

End of Lecture 9: RSL: APPLICATIVE CONSTRUCTS

April 22, 2010, 16:31, Vienna Lectures, April 2010

© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark