

lib/main/list-of.ath

```

1 load "nat-plus"
2
3 #####
4 ## Polymorphic lists
5
6 #datatype (List T) := nil | (:: T (List T))
7
8 assert (datatype-axioms "List")
9
10 module List {
11   open N
12
13   #-----
14
15   define [L L' L1 L2 L3 l l' l1 l2 p q r L M x y z x' h t t1 t2] :=
16     [?L:(List 'S1) ?L':(List 'S2) ?L1:(List 'S3) ?L2:(List 'S4) ?L2:(List 'S5)
17     ?l:(List 'S6) ?l':(List 'S7) ?l1:(List 'S8) ?l2:(List 'S9)
18     ?p:(List 'S10) ?q:(List 'S11) ?r:(List 'S12) ?L:(List 'S13)
19     ?M:(List 'S14) ?x ?y ?z ?x' ?h ?t ?t1 ?t2]
20
21
22 declare join: (T) [(List T) (List T)] -> (List T) [[(alist->list id) (alist->list id)]]
23 define ++ := join
24
25 module join {
26
27   assert left-empty := (forall q . nil join q = q)
28
29   assert left-nonempty :=
30     (forall x r q . (x :: r) join q = x :: (r join q))
31
32   define right-empty := (forall p . p join nil = p)
33
34   define right-nonempty :=
35     (forall p y q .
36       p join (y :: q) = (p join (y :: nil)) join q)
37
38   by-induction right-empty {
39     nil =>
40       (!chain [(nil join nil) = nil [left-empty]])
41   | (x :: p) =>
42     let {induction-hypothesis := (p join nil = p)}
43       (!chain [(x :: p) join nil
44         --> (x :: (p join nil)) [left-nonempty]
45         --> (x :: p) [induction-hypothesis]])
46   }
47
48   by-induction right-nonempty {
49     nil =>
50     pick-any y q
51       (!combine-equations
52         (!chain [(nil join (y :: q))
53           --> (y :: q) [left-empty]])
54         (!chain [(nil join (y :: nil)) join q
55           --> ((y :: nil) join q) [left-empty]
56           --> (y :: (nil join q)) [left-nonempty]
57           --> (y :: q) [left-empty]]))
58   | (x :: p) =>
59     let {induction-hypothesis :=
60       (forall ?y ?q .
61         p join (?y :: ?q) = (p join (?y :: nil)) join ?q)}
62     conclude (forall ?y ?q .
63       (x :: p) join (?y :: ?q) =
64       ((x :: p) join (?y :: nil)) join ?q)
65     pick-any y q
66       (!combine-equations
67         (!chain [(x :: p) join (y :: q)
68           --> (x :: (p join (y :: q))) [left-nonempty]

```

```

69         --> (x :: ((p join (y :: nil)) join q))
70             [induction-hypothesis]]
71     (!chain [((x :: p) join (y :: nil)) join q]
72         --> ((x :: (p join (y :: nil))) join q)
73             [left-nonempty]
74         --> (x :: ((p join (y :: nil)) join q))
75             [left-nonempty]))
76 }
77
78 define Associative :=
79     (forall p q r .
80         (p join q) join r = p join (q join r))
81
82 by-induction Associative {
83     nil =>
84     pick-any q r
85         (!chain [(nil join q) join r]
86             --> (q join r) [left-empty]
87             <-- (nil join (q join r)) [left-empty]))
88 | (x :: p) =>
89     let {induction-hypothesis :=
90         (forall ?q ?r . (p join ?q) join ?r =
91             p join (?q join ?r))}
92     conclude (forall ?q ?r .
93         ((x :: p) join ?q) join ?r =
94         (x :: p) join (?q join ?r))
95     pick-any q r
96         (!chain
97             [((x :: p) join q) join r]
98             --> ((x :: (p join q)) join r) [left-nonempty]
99             --> (x :: ((p join q) join r)) [left-nonempty]
100            --> (x :: (p join (q join r))) [induction-hypothesis]
101            <-- ((x :: p) join (q join r)) [left-nonempty]
102            ])
103 }
104
105 define left-singleton :=
106     (forall x p . (x :: nil) join p = x :: p)
107
108 conclude left-singleton
109 pick-any x p
110     (!chain
111         [((x :: nil) join p)
112         = (x :: (nil join p)) [left-nonempty]
113         = (x :: p) [left-empty]])
114
115 } # join
116
117 #-----
118 declare reverse: (T) [(List T)] -> (List T) [(alist->list id)]
119
120 module reverse {
121
122     assert empty := ((reverse nil) = nil)
123     assert nonempty :=
124         (forall x r . (reverse (x :: r)) = (reverse r) join (x :: nil))
125
126     define of-join :=
127         (forall p q . (reverse (p join q)) = (reverse q) join (reverse p))
128
129     define of-reverse := (forall p . (reverse (reverse p)) = p)
130
131 by-induction of-join {
132     nil =>
133     conclude (forall q . (reverse (nil join q)) =
134         (reverse q) join (reverse nil))
135     pick-any q
136         (!combine-equations
137         (!chain [(reverse (nil join q))
138             --> (reverse q) [join.left-empty]]))

```

```

139         (!chain [((reverse q) join (reverse nil))
140                  --> ((reverse q) join nil) [empty]
141                  --> (reverse q)           [join.right-empty]]])
142 | (x :: p) =>
143 let {induction-hypothesis :=
144       (forall ?q . (reverse (p join ?q)) =
145                   (reverse ?q) join (reverse p))}
146 conclude (forall ?q . (reverse ((x :: p) join ?q)) =
147                (reverse ?q) join (reverse (x :: p)))
148 pick-any q
149   (!chain [ (reverse ((x :: p) join q))
150             --> (reverse (x :: (p join q))) [join.left-nonempty]
151             --> ((reverse (p join q)) join (x :: nil))
152                 [nonempty]
153             --> (((reverse q) join (reverse p)) join (x :: nil))
154                 [induction-hypothesis]
155             --> ((reverse q) join ((reverse p) join (x :: nil)))
156                 [join.Associative]
157             <-- ((reverse q) join (reverse (x :: p)))
158                 [nonempty]]])
159 }
160
161 by-induction of-reverse {
162   nil =>
163   conclude ((reverse (reverse nil)) = nil)
164   (!chain [(reverse (reverse nil))
165            --> (reverse nil)           [empty]
166            --> nil                     [empty]])
167 | (x :: p) =>
168 conclude ((reverse (reverse (x :: p))) = (x :: p))
169 let {induction-hypothesis := ((reverse (reverse p)) = p)}
170 (!chain
171  [(reverse (reverse (x :: p)))
172   --> (reverse ((reverse p) join (x :: nil)))
173        [nonempty]
174   --> ((reverse (x :: nil)) join (reverse (reverse p)))
175        [of-join]
176   --> ((reverse (x :: nil)) join p) [induction-hypothesis]
177   --> (((reverse nil) join (x :: nil)) join p)
178        [nonempty]
179   --> ((nil join (x :: nil)) join p) [empty]
180   --> ((x :: nil) join p) [join.left-empty]
181   --> (x :: (nil join p)) [join.left-nonempty]
182   --> (x :: p) [join.left-empty]])
183 }
184
185 #-----
186 # Another relationship between reverse and join:
187
188 define join-singleton :=
189   (forall p x . (reverse (p join (x :: nil))) =
190                x :: (reverse p))
191
192 conclude join-singleton
193 pick-any p x
194   (!chain
195    [(reverse (p join (x :: nil)))
196     --> ((reverse (x :: nil)) join (reverse p)) [of-join]
197     --> (((reverse nil) join (x :: nil)) join (reverse p))
198          [nonempty]
199     --> ((nil join (x :: nil)) join (reverse p)) [empty]
200     --> ((x :: nil) join (reverse p)) [join.left-empty]
201     --> (x :: (nil join (reverse p))) [join.left-nonempty]
202     --> (x :: (reverse p)) [join.left-empty]])
203
204 #-----
205 # Another proof of reverse, using join-singleton:
206
207 by-induction of-reverse {
208   nil =>

```

```

209   conclude ((reverse (reverse nil)) = nil)
210     (!chain [(reverse (reverse nil))
211             --> (reverse nil)           [empty]
212             --> nil                     [empty]])
213 | (x :: p) =>
214   conclude ((reverse (reverse (x :: p))) = (x :: p))
215   let {induction-hypothesis := ((reverse (reverse p)) = p)}
216     (!chain
217       [(reverse (reverse (x :: p)))
218        --> (reverse ((reverse p) join (x :: nil))) [nonempty]
219        --> (x :: (reverse (reverse p)))           [join-singleton]
220        --> (x :: p)                               [induction-hypothesis]])
221     }
222 } # reverse
223
224 #=====
225 declare length: (T) [(List T)] -> N [(alist->list id)]
226
227 module length {
228
229 assert empty := (length nil = zero)
230 assert nonempty := (forall p x . length (x :: p) = S length p)
231
232 define of-join := (forall p q .
233                  length (p join q) = (length p) + (length q))
234 define of-reverse := (forall p . length reverse p = length p)
235
236 by-induction of-join {
237   nil:(List 'S) =>
238   conclude (forall ?q .
239            length (nil:(List 'S) join ?q) = (length nil:(List 'S) + (length ?q)))
240   pick-any q:(List 'S)
241     (!combine-equations
242      (!chain
243       [(length (nil join q))
244        --> (length q)           [join.left-empty]])
245      (!chain
246       [(length nil:(List 'S)) + (length q)
247        --> (zero + (length q))   [empty]
248        --> (length q)           [Plus.left-zero]))))
249 | (H:'S :: T:(List 'S)) =>
250 conclude (forall ?q . length ((H :: T) join ?q) =
251          (length (H :: T) + length ?q))
252   let {induction-hypothesis :=
253        (forall ?q . length (T join ?q) = (length T) + length ?q)}
254   pick-any q:(List 'S)
255     (!combine-equations
256      (!chain
257       [(length ((H :: T) join q))
258        --> (length (H :: (T join q))) [join.left-nonempty]
259        --> (S (length (T join q)))   [nonempty]
260        --> (S ((length T) + (length q))) [induction-hypothesis]])
261      (!chain
262       [(length (H :: T)) + (length q)
263        --> ((S (length T)) + (length q)) [nonempty]
264        --> (S ((length T) + (length q))) [Plus.left-nonzero]))))
265 }
266
267 by-induction of-reverse {
268   nil =>
269   (!chain [(length (reverse nil:(List 'S)))
270           --> (length nil:(List 'S))      [reverse.empty]])
271 | (x :: p:(List 'S)) =>
272   let {induction-hypothesis := ((length (reverse p)) = (length p))}
273   conclude (length (reverse (x :: p)) = length (x :: p))
274     (!chain
275       [(length (reverse (x :: p)))
276        --> (length ((reverse p) join (x :: nil)))
277         [reverse.nonempty]
278        --> ((length (reverse p)) + (length (x :: nil)))

```

```

279                                     [of-join]
280     --> ((length p) + (length (x :: nil))) [induction-hypothesis]
281     --> ((length p) + (S (length nil:(List 'S)))) [nonempty]
282     --> ((length p) + (S zero)) [empty]
283     --> (S ((length p) + zero)) [Plus.right-nonzero]
284     --> (S (length p)) [Plus.right-zero]
285     <-- (length (x :: p)) [nonempty]]
286 }
287
288 } # length
289
290 =====
291 # List.count: given a value x and a list, returns the number
292 # of occurrences of x in the list.
293
294 declare count: (S) [S (List S)] -> N [[id (alist->list id)]]
295
296 module count {
297 define [x x' L M] := [?x:'S ?x':'S ?L:(List 'S) ?M:(List 'S)]
298
299 assert axioms :=
300   (fun
301     [(count x nil) = zero
302      (count x (x' :: L)) = [(S (count x L)) when (x = x')
303                           (count x L) when (x /= x')]])
304
305 define [empty more same] := axioms
306
307 define of-join :=
308   (forall L M x . (count x (L join M)) = (count x L) + (count x M))
309 define of-reverse :=
310   (forall L x . (count x (reverse L)) = (count x L))
311
312 by-induction of-join {
313   nil =>
314   pick-any M x
315     (!combine-equations
316       (!chain [(count x (nil join M))
317                = (count x M) [join.left-empty]]
318               (!chain [((count x nil) + (count x M))
319                        = (zero + (count x M)) [empty]
320                        = (count x M) [Plus.left-zero]]))
321 | (y :: L) =>
322   let {ind-hyp := (forall ?M ?x . (count ?x (L join ?M)) =
323                                (count ?x L) + (count ?x ?M))}
324   conclude (forall ?M ?x . (count ?x ((y :: L) join ?M)) =
325                       (count ?x (y :: L)) + (count ?x ?M))
326   pick-any M x
327     (!two-cases
328       assume (x = y)
329         (!combine-equations
330           (!chain
331             [(count x ((y :: L) join M))
332              = (count x (y :: (L join M))) [join.left-nonempty]
333              = (S (count x (L join M))) [more]
334              = (S ((count x L) + (count x M))) [ind-hyp]])
335           (!chain
336             [((count x (y :: L)) + (count x M))
337              = ((S (count x L)) + (count x M)) [more]
338              = (S ((count x L) + (count x M))) [Plus.left-nonzero]
339              ]))
340         assume (x /= y)
341           (!combine-equations
342             (!chain
343               [(count x ((y :: L) join M))
344                = (count x (y :: (L join M))) [join.left-nonempty]
345                = (count x (L join M)) [same]
346                = ((count x L) + (count x M)) [ind-hyp]])
347             (!chain
348               [((count x (y :: L)) + (count x M))

```

```

349         = ((count x L) + (count x M))      [same]
350       ])))
351   }
352
353 by-induction of-reverse {
354   nil =>
355     pick-any x
356       (!chain [(count x (reverse nil))
357               = (count x nil)           [reverse.empty]])
358 | (y :: L) =>
359   let {ind-hyp := (forall ?x . (count ?x (reverse L)) = (count ?x L))}
360   conclude (forall ?x . (count ?x (reverse (y :: L))) =
361                 (count ?x (y :: L)))
362     pick-any x
363       (!two-cases
364         assume (x = y)
365           (!chain
366             [(count x (reverse (y :: L)))
367             = (count x ((reverse L) join (y :: nil)))
368               [reverse.nonempty]
369             = ((count x (reverse L)) + (count x (y :: nil)))
370               [of-join]
371             = ((count x L) + (S (count x nil)))
372               [ind-hyp more]
373             = ((count x L) + (S zero)) [empty]
374             = (S ((count x L) + zero)) [Plus.right-nonzero]
375             = (S (count x L))         [Plus.right-zero]
376             = (count x (y :: L))     [more]])
377         assume (x /= y)
378           (!chain
379             [(count x (reverse (y :: L)))
380             = (count x ((reverse L) join (y :: nil)))
381               [reverse.nonempty]
382             = ((count x (reverse L)) + (count x (y :: nil)))
383               [of-join]
384             = ((count x L) + (count x nil))
385               [ind-hyp same]
386             = ((count x L) + zero)    [empty]
387             = (count x L)             [Plus.right-zero]
388             = (count x (y :: L))     [same]]))
389   }
390 } # count
391
392 =====
393 # List.in (membership)
394
395 declare in: (T) [T (List T)] -> Boolean [[id (alist->list id)]]
396
397 module in {
398
399 assert empty := (forall x . ~ x in nil)
400 assert nonempty := (forall x y L . x in (y :: L) <==> x = y | x in L)
401
402 #.....
403 # Lemmas:
404
405 define head := (forall x L . x in (x :: L))
406 define tail := (forall x y L . x in L ==> x in (y :: L))
407
408 conclude head
409   pick-any x L
410     (!chain-> [(x = x)
411              ==> (x = x | x in L) [alternate]
412              ==> (x in (x :: L)) [nonempty]])
413
414 conclude tail
415   pick-any x y L
416     (!chain [(x in L)
417            ==> (x = y | x in L) [alternate]
418            ==> (x in (y :: L)) [nonempty]])

```

```

419
420 define of-singleton :=
421   (forall x y . x in (y :: nil) ==> x = y)
422
423 conclude of-singleton
424 pick-any x y
425   assume (x in (y :: nil))
426   let {C := (!chain-> [(x in (y :: nil)) ==> (x = y | x in nil)
427                       [nonempty]])}
428   (!cases C
429     assume (x = y)
430     (!claim (x = y))
431     assume (x in nil)
432     (!from-complements (x = y)
433       (x in nil) (!chain-> [true ==> (~ x in nil) [empty]])))
434
435 #.....
436 # Theorem:
437
438 define of-join :=
439   (forall L M x . x in (L join M) <==> x in L | x in M)
440
441 by-induction of-join {
442   nil =>
443     conclude (forall ?M ?x . ?x in (nil join ?M) <==>
444       ?x in nil | ?x in ?M)
445     pick-any M x
446     let {_ := (!chain->
447       [true ==> (~ x in nil) [empty]
448        <==> (x in nil <==> false) [prop-taut]])}
449     (!chain
450       [(x in (nil join M))
451        <==> (x in M) [join.left-empty]
452        <==> (false | x in M) [prop-taut]
453        <==> (x in nil | x in M) [(x in nil <==> false)]]
454   | (y :: L) =>
455     let {ind-hyp := (forall ?M ?x .
456       ?x in (L join ?M) <==> ?x in L | ?x in ?M)}
457     conclude (forall ?M ?x .
458       ?x in ((y :: L) join ?M) <==>
459       ?x in (y :: L) | ?x in ?M)
460     pick-any M x
461     (!chain
462       [(x in ((y :: L) join M))
463        <==> (x in (y :: (L join M))) [join.left-nonempty]
464        <==> (x = y | x in (L join M)) [nonempty]
465        <==> (x = y | x in L | x in M) [ind-hyp]
466        <==> ((x = y | x in L) | x in M) [prop-taut]
467        <==> (x in (y :: L) | x in M) [nonempty]])
468   }
469 } # in
470
471 #=====
472 # (List.replace L x y) returns a copy of L except that all
473 # occurrences of x are replaced by y
474
475 declare replace: (S) [(List S) S S] -> (List S) [(alist->list id) id id]
476
477 module replace {
478
479 assert axioms :=
480   (fun
481     [(replace nil x y) = nil
482      (replace (x' :: L) x y) =
483        [(y :: (replace L x y)) when (x = x')
484         (x' :: (replace L x y)) when (x /= x')]])
485
486 define [empty equal unequal] := axioms
487
488 define sanity-check1 :=

```

```

489 (forall L x y .
490   x /= y ==> (count x (replace L x y)) = zero)
491
492 define sanity-check2 :=
493 (forall L x y .
494   x /= y ==>
495   (count y (replace L x y)) = (count x L) + (count y L))
496
497 by-induction sanity-check1 {
498   nil =>
499   pick-any x y
500     assume (x /= y)
501       (!chain [(count x (replace nil x y))
502                = (count x nil)           [empty]
503                = zero                     [count.empty]])
504 | (z :: L) =>
505   pick-any x y
506     assume (x /= y)
507     let {ind-hyp := (forall ?x ?y .
508                      ?x /= ?y ==> (count ?x (replace L ?x ?y)) = zero);
509          _ := (!sym (x /= y))}
510     (!two-cases
511       assume (x = z)
512         (!chain
513           [(count x (replace (z :: L) x y))
514            = (count x (y :: (replace L x y))) [equal]
515            = (count x (replace L x y))       [count.same]
516            = zero                             [ind-hyp]])
517         assume (x /= z)
518           (!chain
519             [(count x (replace (z :: L) x y))
520              = (count x (z :: (replace L x y))) [unequal]
521              = (count x (replace L x y))       [count.same]
522              = zero                             [ind-hyp]]))
523     }
524
525 by-induction sanity-check2 {
526   nil =>
527   pick-any x y
528     assume (x /= y)
529     (!combine-equations
530       (!chain [(count y (replace nil x y))
531                = (count y nil)           [empty]
532                = zero                     [count.empty]])
533       (!chain [(count x nil) + (count y nil))
534                = (zero + zero)           [count.empty]
535                = zero                     [Plus.right-zero])))
536 | (z:'S :: L) =>
537   pick-any x:'S y
538     assume (x /= y)
539     let {ind-hyp := (forall ?x ?y .
540                      ?x /= ?y ==> (count ?y (replace L ?x ?y)) =
541                      (count ?x L) + (count ?y L));
542          _ := (!sym (x /= y))}
543     (!two-cases
544       assume (y = z)
545         (!combine-equations
546           (!chain
547             [(count y (replace (z :: L) x y))
548              = (count y (replace (y :: L) x y)) [(y = z)]
549              = (count y (y :: (replace L x y))) [unequal]
550              = (S (count y (replace L x y)))    [count.more]
551              = (S ((count x L) + (count y L))) [ind-hyp]])
552           (!chain
553             [(count x (z :: L)) + (count y (z :: L))
554              = ((count x (y :: L)) + (count y (z :: L)))
555              = ((count x L) + (count y (y :: L))) [count.same (y = z)]
556              = ((count x L) + (S (count y L)))    [count.more]
557              = (S ((count x L) + (count y L)))    [Plus.right-nonzero])))
558     assume (y /= z)

```



```

559     (!two-cases
560       assume (x = z)
561       (!combine-equations
562         (!chain
563           [(count y (replace (z :: L) x y))
564            = (count y (y :: (replace L x y))) [equal]
565            = (S (count y (replace L x y))) [count.more]
566            = (S ((count x L) + (count y L))) [ind-hyp]])
567         (!chain
568           [(count x (z :: L)) + (count y (z :: L))]
569            = ((S (count x L)) + (count y L)) [count.more count.same]
570            = (S ((count x L) + (count y L))) [Plus.left-nonzero]))
571       assume (x /= z)
572       (!combine-equations
573         (!chain
574           [(count y (replace (z :: L) x y))
575            = (count y (z :: (replace L x y))) [unequal]
576            = (count y (replace L x y)) [count.same]
577            = ((count x L) + (count y L)) [ind-hyp]])
578         (!chain
579           [(count x (z :: L)) + (count y (z :: L))]
580            = ((count x L) + (count y L)) [count.same])))
581   }
582 } # replace
583 } # List
584
585 define (alist->clist inner) :=
586   letrec {loop := lambda (L acc)
587     match L {
588       (list-of x rest) => (loop rest ((inner x) :: acc))
589       | [] => acc
590     }}
591   lambda (L)
592     match L {
593       (some-list _) => (loop (rev L) (nil))
594       | _ => L
595     }
596
597 define (clist->alist inner) :=
598   letrec {loop := lambda (L acc)
599     match L {
600       (x :: rest) => (loop rest (add (inner x) acc))
601       | nil => (rev acc)
602     }}
603   lambda (L)
604     match L {
605       (x :: rest) => (loop L [])
606       | nil => []
607       | _ => L
608     }

```