

lib/basic/property-management.ath

```

1 #####
2 # Property management functions
3 #
4 # These functions provide dynamic storage, retrieval, and indexing of
5 # sentences according to the theory or theories with which they are
6 # associated.
7
8 module propman {
9
10 (define search-stack (cell []))
11
12 (define plist-list (cell []))
13
14 # (print-prop-name name) prints name, which can be either a meta-id
15 # or a name of form [prefix name], where prefix and name are meta-id's.
16
17 (define (print-prop-name name)
18   (match name
19     ([_prefix _n] (seq (print "[" (write-val _prefix) (print " ")
20                        (write-val _n) (print "]"))))
21     (_ (write-val name))))
22
23 # The following procedures are used in reporting the error when an
24 # attempt is made to access a sentence via (f n) where either f has
25 # not been defined via concrete or theory, or, if it has, no
26 # sentence has been defined with name n.
27
28 (define (defined? plist name)
29   (letrec ((loop (lambda (alist)
30                     (match alist
31                       ([] false)
32                       ((split [_x _y] _rest)
33                        (check ((equal? _y name) true)
34                              (else (loop _rest)))))))
35     (loop (ref plist))))
36
37 (define (error-name-not-defined-in plist-name prop)
38 # let {_ := (print (join "\nError: in " (tail (val->string plist-name)) "\n"));
39 #      _ := (write prop);
40 #      _ := (print "not found\n");
41 #      _ := (!proof-error "Property not defined.\n")}
42   ())
43
44 (define (error-plist-not-defined name)
45   (let ((_ (!proof-error
46             (join "\nError: searching for " (val->string name)
47                  " in undefined property list\n"))))
48     ()))
49
50 (define (get-plist-name alist)
51   (match alist
52     ((split [_x _y] _rest)
53      (check ((equal? _y 'selfname) _x)
54              (else (get-plist-name _rest)))))
55
56 # The following procedures retrieve the sentence associated with
57 # name in plist. The difference is that get reports an error if it
58 # doesn't find the name, whereas get-abstract simply fails (thus it
59 # should always be used within a try expression with alternatives that
60 # do further searching).
61
62 (define (get plist name)
63   (letrec ((loop (lambda (alist)
64                     (match alist
65                       ([] (check ((defined? plist-list plist)
66                                   (error-name-not-defined-in
67                                    (get-plist-name (ref plist)) name))
68                                   (loop (ref plist))))
69                       ((split [_x _y] _rest)
68                        (check ((equal? _y name) true)
69                              (else (loop _rest)))))))
70     (loop (ref plist))))

```

```

68         (else (error-plist-not-defined name))))
69     ((split [_x _y] _rest)
70      (check ((equal? _y name) _x)
71              (else (loop _rest))))))
72 (loop (ref plist)))
73
74 (define (get-abstract plist name)
75   (letrec ((loop (lambda (alist)
76                     (match alist
77                       ((split [_x _y] _rest)
78                        (check ((equal? _y name) _x)
79                                (else (loop _rest))))
80                       ([] (halt))))))
81     (loop (ref plist)))
82
83 # Data structure symbol-index is an updatable index of symbol-value
84 # pairs, where the value associated with a symbol is a cell containing
85 # the list of all names of sentences in which the symbol appears.
86
87 (define symbol-index (cell []))
88
89 (define (in-symbol-index? sym)
90   (letrec ((loop (lambda (pair-list)
91                     (match pair-list
92                       ([] false)
93                       ((split [_namelist _name] _rest)
94                        (check ((equal? _name sym)
95                                true)
96                                (else (loop _rest))))))
97     (loop (ref symbol-index)))
98
99 ## Procedure index-sym makes an entry in symbol-index: given symbol s,
100 ## it adds name to the list of names associated with s.
101
102 (define (index-sym name sym)
103   (check ((in-symbol-index? sym)
104          (let ((names (get-abstract symbol-index sym)))
105              (check ((member? name (ref names)) ())
106                      (else (set! names (add name (ref names))))))
107          (else (set! symbol-index (add (cell [name])
108                                         (add sym (ref symbol-index))))))
109
110 ## Procedure index-syms makes entries in symbol-index: for each symbol
111 ## sym in sentence, it adds name to the list of names associated with s.
112
113 (define (index-syms name sentence)
114   (map-proc (lambda (sym) (index-sym name sym))
115             (get-prop-syms sentence)))
116
117 ## Procedure index-syms1 makes entries in symbol-index: for each
118 ## symbol sym in sentence, it adds sentence to the list of sentences
119 ## associated with s.
120
121 (define (index-syms1 sentence)
122   (map-proc (lambda (sym) (index-sym sentence sym))
123             (get-prop-syms sentence)))
124
125 ## Procedure retrieve-props returns the list of all properties
126 ## associated with symbol sym in symbol-index, i.e., those that
127 ## contain sym.
128
129 (define (retrieve-props sym)
130   (check ((member? sym (ref symbol-index))
131          (ref (get-abstract symbol-index sym)))
132          (else [])))
133
134 ## Procedure retrieve-props* returns the list of all indexed properties
135 ## that each contain all of the symbols in list-of-syms.
136
137 (define (retrieve-props* list-of-syms)

```

```

138 (intersection* (map retrieve-props list-of-syms)))
139
140 define Module-Table := (table 100)
141
142 (define (index-module M)
143   let {M := (join "\" M "\"")}
144   # (map-proc index-syms1 (filter (map lambda (x) (try (apply-module M x) ())
145   # (module-domain M))
146   # prop?))
147   let {_ := (process-input-from-string
148     (join "(table-add Module-Table [\" M
149     M \" x) () (module-domain \" M \") prop?))" ))}
150   # (table-add Module-Table [M --> (table (map lambda (x) [(try (apply-module M x) ()) x]
151   # (module-domain M)))]))
152 (process-input-from-string
153 (join "(table-add Module-Table [\" M
154 \" --> (table (map lambda (x) [(try (apply-module \"
155 M \"x) ()) x] (module-domain \" M \")))]" )))
156
157 define sentence-names :=
158   lambda (s)
159     (filter
160       (map lambda (pair)
161         match pair {
162           [_M _T] => (try (join _M "." (table-lookup _T s)) false)
163         }
164       (table->list Module-Table))
165     lambda (x) (negate (equal? x false)))
166
167 # Using new module indexing implemented in r1891:
168 define sentence-names :=
169   lambda (s)
170     let {names := (reverse-lookup s)}
171     check {
172       (equal? names []) =>
173         let {_ := (build-inverted-index)}
174         (reverse-lookup s)
175       | else => names
176     }
177
178 define print-names :=
179   lambda (s)
180     match (sentence-names s) {
181       (list-of name rest) =>
182         let {_ := (print name);
183         _ := (map-proc
184           lambda (name) (print " and " name)
185           rest)}
186       (print "\n")
187     }
188
189 ## Procedure print-index-props (pip for short) prints all indexed
190 ## properties (those asserted or defined in modules) that contain all
191 ## of the symbols in syms.
192
193 define print-indexed-properties :=
194   lambda (syms)
195     let {_ := (print "\nStored sentences that contain ");
196     _ := match syms {
197       (list-of sym rest) =>
198         let {_ := (print (symbol->string sym));
199         _ := (map-proc
200           lambda (sym) (print " and " (symbol->string sym))
201           rest)}
202       (print ":\n")
203     }
204     (map-proc
205       lambda (sent)
206         match (sentence-names sent) {
207         (list-of name rest) =>

```

```

208         let {_ := (print name);
209             _ := (map-proc
210                 lambda (name) (print "  and " name)
211                 rest);
212             _ := (print ":\n ");
213             _ := (indent-print 4 sent)}
214         (print "\n")
215     }
216     (retrieve-props* syms))
217
218 (define pip print-indexed-properties)
219
220 (define (close-props name-value-list)
221     (letrec ((loop (lambda (L M)
222                     (match L
223                         ((split [_x _y] _rest)
224                          (loop _rest (add (close _y) (add _x M))))
225                         ([] M))))))
226         (loop name-value-list [])))
227
228 #####
229 # Procedures for storing, retrieving, indexing, and adapting properties
230 # of a noncategorical theory.
231
232 ## The following conventions for parameter names are used:
233
234 ## - op is a symbol used as an operator name in terms. In some
235 ## procedures it can also be a list of such symbols
236
237 ## - newnames is a list of the form [x_1 y_1 x_2 y_2 ...] where the
238 ## x_i and y_i are symbols used as operator names in terms. Its
239 ## meaning in these procedures is that y_i is a replacement for x_i.
240
241 ## (replace op newnames): If op is a symbol, returns the replacement
242 ## for op specified in newnames, if any (i.e., if op occurs as an x_i
243 ## in newnames then y_i, otherwise op). If op is a list of symbols,
244 ## returns their replacements as specified in newnames.
245
246 (define (replace op newnames)
247     (match op
248         ((some-list L) (map (lambda (x) (replace x newnames)) L))
249         (_ (match newnames
250             ([] op)
251             ((split [x y] rest)
252              (check ((equal? x op) y)
253                      (else (replace op rest)))))))
254
255 ## (term-adapt term newnames): returns the term resulting from
256 ## replacing each operator symbol in term as specified in newnames.
257
258 (define (term-adapt term newnames)
259     (match term
260         ((op (some-list args))
261          (make-term (replace op newnames)
262                     (map (lambda (t) (term-adapt t newnames)) args)))
263         ((some-var v) v)
264         ((some-symbol x) (replace x newnames)))
265
266 ## (prop-adapt term newnames): returns the sentence resulting
267 ## from replacing each operator symbol in prop as specified in
268 ## newnames.
269
270 (define (prop-adapt prop newnames)
271     (match prop
272         ((some-var v) v)
273         ((not _P) (not (prop-adapt _P newnames)))
274         (((some-sent-con pc) _P1 _P2)
275          (pc (prop-adapt _P1 newnames) (prop-adapt _P2 newnames)))
276         (((some-quant quant) _v _B)
277          (quant _v (prop-adapt _B newnames)))

```

```

278      (_ (term-adapt prop newnames))))
279
280  ## (renaming newnames): returns a procedure r such that (r p), where p
281  ## is a sentence, returns the sentence that results from
282  ## replacing the operators in p as specified in newnames. Or if p is
283  ## a list of sentences [p_1 p_2 ...], r returns [(r p_1) (r p_2)
284  ## ...]. Lastly, if p is the meta-id 'selflist, r returns newnames.
285
286  (define (renaming-core newnames)
287    (lambda (p)
288      # (let ((_ (print "B:")) (_ (write p)) (_ (write newnames)))
289          (match p
290            ((some-list L) (map (lambda (p) (prop-adapt p newnames)) L))
291            ('selflist newnames)
292            (_ (prop-adapt p newnames)))))
293    #)
294
295  (define (renaming newnames)
296    (match newnames
297      ((some-map m) (renaming-core (map get-symbol (flatten (map-key-values m)))))
298      (_ (renaming-core (map get-symbol newnames)))))
299
300  (define no-renaming
301    (renaming []))
302
303  ## Symbol Axiom is used to mark the sentences defined in a pfun as
304  ## axioms (i.e., must be either asserted or proved outside of the
305  ## theory in which they appear). External can be used as a synonym
306  ## for Axiom; so can Definition.
307
308  (declare Axiom Boolean)
309
310  (define External Axiom)
311
312  (define Definition Axiom)
313
314  ## (resolve name prefix): If name is of the form [pref n] and pref
315  ## equals prefix, returns n, otherwise fails.
316
317  (define (resolve name prefix)
318    (match name
319      ([_first _second]
320       (check
321        ((equal? _first prefix)
322         _second))))
323
324  ## (find name lst) searches lst, of the form [n_1 v_1 n_2 v_2 ...] for
325  ## the first n_i that equals name and returns the corresponding v_i.
326  ## It fails if there is no such n_i.
327
328  (define (find lst name)
329    (match lst
330      ((split [_name1 _value] _rest)
331       (check ((equal? name _name1)
332              _value)
333              (else (find _rest name)))))
334
335  ## (change-ops adapt newnames) returns a new adapter procedure that
336  ## first renames according to the one defined by newnames, then
337  ## according to the given adapter
338
339  (define (change-ops adapt newnames)
340    (o adapt (renaming newnames)))
341
342  # For several of the procedures defined below, the following data
343  # structures are used.
344  #
345  ## A tp-list is a list of the form [[f_1 p_1] [f_2 p_2] ... [f_j p_j]
346  ## 'superiors t_1 t_2 ... t_k], where each f_i is a pfun defined with the
347  ## abstract procedure, each p_i is either the symbol Axiom or a proof

```

```

348  ## method, and each t_i is a tfun or an adapted theory. Either j = 0
349  ## or k = 0 is permitted, but not both.
350
351  ## A tfun is a procedure that encapsulates a tp-list and also is
352  ## reflective: (tfun 'selfname) returns the name (a meta-id) given to
353  ## the procedure when it was defined, and (pfun 'selfcell) returns its
354  ## tp-list.
355
356  ## An adapted theory is list of the form [basetheory prefix newnames],
357  ## where basetheory is a tfun, prefix is a name, and newnames is a
358  ## list of operator replacements as previously described.
359
360  ## A tp-list is constructed by the theory procedure and is searched by
361  ## get-from-theory (which is used by the theory procedure).
362
363  ## (get-from-theory tp-list n): If f_i is the first pfun in tp-list
364  ## such that (f_i n) is defined, returning t_i, then the pair [t_i
365  ## p_i] is returned. If (f_i n) fails for every f_i, the ancestor
366  ## theories are searched. An ancestor that is a tfun is searched by
367  ## searching its tp-list recursively, while one of the form
368  ## [basetheory prefix newnames] is searched by searching for the name
369  ## (resolve name prefix) in basetheory, and if [t p] is thus returned
370  ## then [t' p'] is the pair returned, where t' is ((renaming newnames)
371  ## t) and p' is
372  #
373  ## (method (name adapt)
374  ##   (!p (resolve name prefix) (change-ops adapt newnames)))
375
376  define get-from-theory :=
377    lambda (tp-list prop)
378      letrec {loop :=
379        lambda (alist)
380          match alist {
381            (list-of _a _rest) =>
382              match _a {
383                [_x _y] => try {[check {(member? prop _x) => prop}
384                               _y]
385                             | (loop _rest)
386                             }
387              | 'superiors => (ancestor-loop _rest)
388            }
389          };
390        ancestor-loop :=
391          lambda (theories)
392            match theories {
393              (list-of _th _more) =>
394                match _th {
395                  (some-proc th) =>
396                    (try (get-from-theory (th 'selfcell) prop)
397                      (ancestor-loop _more))
398                  | [_basetheory _prefix _newnames] =>
399                    let {prop' := (resolve prop _prefix)}
400                    try {
401                      match (_basetheory prop') {
402                        [_t _p] =>
403                          let {t' := ((renaming _newnames) _t)}
404                          [t' match _p {
405                            Axiom => Axiom
406                            | _ => method (prop adapt)
407                               (!_p prop'
408                                (change-ops adapt
409                                 _newnames))
410                          }]
411                      } # match
412                    | (ancestor-loop _more)
413                  } # try
414                } # match
415              } # match
416          } #letrec
417      (loop (ref tp-list))

```

```

418
419 ## (index-theory theory-name tp-list) indexes the tp-list's toplevel
420 ## property lists L_i. It ignores ancestor theories.
421
422 (define (index-theory theory-name tp-list)
423   (match tp-list
424     ((list-of [_L _p] _rest)
425      (seq (map-proc index-syms1 _L)
426            (index-theory theory-name _rest)))
427     ((list-of 'superiors _rest) ())
428     ([ ] ())))
429
430 ## (theory-properties tfun) returns the sentences in
431 ## the plist in tfun (at the top level and in superiors), an alist.
432
433 (define (theory-properties tfun)
434   (letrec ((helper (lambda (alist)
435                      (match alist
436                        ((list-of _x _rest)
437                         (match _x
438                          ([_t _p] (join _t (helper _rest)))
439                          ('superiors (fold join
440                                             (map theory-properties _rest)
441                                             []))))))
442            (match alist
443              ([ ] []))))))
444   (match tfun
445     ((some-proc f)
446      (helper (ref (f 'selfcell))))
447     ([_basetheory _prefix _newnames]
448      (let ((adapt (renaming _newnames)))
449        (map (lambda (x)
450               (match x
451                ((some-sent p) (adapt p))
452                (_ (add _prefix [x]))))
453              (theory-properties _basetheory))))))
454   )
455 ## (index-adapted theory-name superiors) indexes each tfun or adapted
456 ## theory in superiors.
457
458 (define (index-adapted theory-name superiors)
459   (letrec ((loop1
460             (lambda (basetheory-name prefix newnames)
461               (match newnames
462                 ((split [_n _n'] _rest)
463                  (seq
464                   (loop2 (retrieve-props _n) basetheory-name prefix _n')
465                   (loop1 basetheory-name prefix _rest)))
466                 ([ ] ())))))
467             (loop2 (lambda (prop-names basetheory-name prefix n')
468                     (match prop-names
469                       ((list-of [_pname _p] _more)
470                        (seq
471                         (check ((equal? _pname basetheory-name)
472                                (index-sym [theory-name [prefix _p]] n'))
473                                (else ()))
474                         (loop2 _more basetheory-name prefix n'))
475                       ([ ] ())))))
476             (match superiors
477               ((list-of _th _rest)
478                (seq (match _th
479                      ((some-proc p) ())
480                      ([_basetheory _prefix _newnames]
481                       (loop1 (_basetheory 'selfname) _prefix _newnames)))
481                    (index-adapted theory-name _rest)))
482               ([ ] ())))))
483
484 (define (index-adapted theory-name superiors)
485   (letrec ((loop
486             (lambda (tprops th)
487               (match tprops

```

```

488         ((split [_pname _prop] _rest)
489         (seq
490         (match th
491         ([_basetheory _prefix _newnames]
492         (index-syms [theory-name _pname] ((renaming _newnames) _prop))))
493         (loop _rest th)))
494         ([ ] ())))))
495 (match superiors
496 ((list-of _th _rest)
497 (seq (match _th
498 ((some-proc p) ())
499 ([_basetheory _prefix _newnames]
500 (loop (theory-properties _th) _th)))
501 (index-adapted theory-name _rest)))
502 ([ ] ())))))
503
504 ## (theory superiors tp-list theory-name) defines a tfun with name theory-name,
505 ## toplevel pairs [f_i p_i] as given in tp-list, and ancestor theories as
506 ## given in superiors.
507
508 define theory :=
509   lambda (superiors tp-list theory-name)
510   let {new-plist := (cell (add 'superiors superiors));
511       _ := (set! plist-list (add theory-name (add new-plist (ref plist-list))));
512       _ := match tp-list {
513         (some-list p) => (set! new-plist (add [p Axiom] (ref new-plist)))
514         | (some-proc p) => (set! new-plist (add p (ref new-plist)))
515       };
516   #   _ := (index-adapted theory-name superiors);
517   pfun := lambda (prop)
518     match prop {
519       'selfname => theory-name
520       | 'selfcell => new-plist
521       | _ => try {let { _ := (set! search-stack (add [theory-name prop]
522                                                         (ref search-stack)))}
523                 (get-from-theory new-plist prop)
524                 |
525                 ()
526                 }
527     }
528   }
529   pfun
530
531 ## (theory-axioms theory) returns a list of the properties (without
532 ## their names) that are labelled Axiom in theory, which may be either
533 ## a tfun or an adapted theory.
534
535 (define (theory-axioms theory)
536   (letrec ((helper (lambda (alist)
537                     (match alist
538                     ((list-of _x _rest)
539                     (match _x
540                     ([_t Axiom] (join _t (helper _rest)))
541                     ([_ _] (helper _rest)))
542                     'superiors (fold join
543                                     (map theory-axioms _rest)
544                                     [ ]))))))
545   (match theory
546   ((some-proc tfun)
547   (remove-duplicates (helper (ref (tfun 'selfcell)))))
548   ([_basetheory _prefix _newnames]
549   (let ((adapt (renaming _newnames)))
550     (map (lambda (x)
551           (match x
552           ((some-sent p) (adapt p))
553           (_ (add _prefix [x]))))
554          (theory-axioms _basetheory))))))
555
556
557

```



```

558 ## (theory-axiom-names theory) returns a list of the names of
559 ## properties that are labelled Axiom in theory, which may be either
560 ## a tfun or an adapted theory.
561
562 (define (theory-axiom-names theory)
563   (letrec ((helper (lambda (alist)
564                     (match alist
565                       ((list-of _x _rest)
566                        (match _x
567                         ([_t Axiom] (join [_t] (helper _rest)))
568                         ([_ _] (helper _rest))
569                         ('superiors (fold join
570                                           (map theory-axiom-names _rest)
571                                           []))))))
572             (match theory
573               ((some-proc tfun)
574                (remove-duplicates (helper (ref (tfun 'selfcell)))))
575               ([_basetheory _prefix _newnames]
576                (let ((adapt (renaming _newnames)))
577                  (map (lambda (x)
578                        (match x
579                          ((some-sent p) (adapt p))
580                          (_ (add _prefix [x]))))
581                      (theory-axiom-names _basetheory)))))))
583
584 (define (theory-theorem-names theory)
585   (letrec ((helper (lambda (alist)
586                     (match alist
587                       ((list-of _x _rest)
588                        (match _x
589                         ([_ Axiom] (helper _rest))
590                         ([_t _] (join [_t] (helper _rest)))
591                         ('superiors (fold join
592                                           (map theory-theorem-names _rest)
593                                           []))))))
594             (match theory
595               ((some-proc tfun)
596                (remove-duplicates (helper (ref (tfun 'selfcell)))))
597               ([_basetheory _prefix _newnames]
598                (let ((adapt (renaming _newnames)))
599                  (map (lambda (x)
600                        (match x
601                          ((some-sent p) (adapt p))
602                          (_ (add _prefix [x]))))
603                      (theory-theorem-names _basetheory)))))))
605
606 ## theory-clone, for defining a theory by adapting a single theory,
607 ## and adding no new axioms.
608
609 (define (theory-clone adapted-theory theory-name)
610   (theory [adapted-theory] [] theory-name))
611
612 ## (print-theory theory) prints on the terminal a list of the sentences
613 ## (with their names) in theory, which may be either a tfun or an adapted
614 ## theory.
615
616 define sentence-name :=
617   lambda (M s)
618     (join M "." (table-lookup (table-lookup Module-Table M) s))
619
620 # Using new module indexing implemented in r1891:
621 define sentence-name :=
622   lambda (M s)
623     let {unqualified-name := match (head (rev (sentence-names s))) {
624       (split _ (list-of \. name)) => name
625     }}
626     (join M "." unqualified-name)
627

```

```

628 (define (name->string pfun)
629   (tail (val->string (pfun 'selfname))))
630
631 (define (print-theory theory)
632   letrec {helper := lambda (alist theory-name)
633           match alist {
634             (list-of _x _rest) =>
635             match _x {
636               [_t _p] =>
637               let {_ := (map-proc
638                       lambda (s)
639                         (print "\n" (join (sentence-name theory-name s)
640                                           ":\n" (val->string s) "\n\n"))
641                       _t)}
642                 (helper _rest theory-name)
643               | 'superiors => (map-proc print-theory _rest)
644               | _ => ()
645             }
646           }}
647   match theory {
648     (some-proc f) => let {_ := (print "\n");
649                        _ := (print (join (name->string f) ":\n\n"))
650                        (helper (ref (f 'selfcell)) (name->string f))
651     | [basetheory prefix newnames] =>
652       (map-proc lambda (x)
653                 let {_ := (print (sentence-name (name->string basetheory) x));
654                     _ := (print ":\n");
655                     _ := (write-val ((renaming newnames) x))
656                 (print "\n\n")
657                 (theory-properties basetheory))
658     })
659
660 (define (theory-list? plist)
661   (member? 'superiors (ref plist)))
662
663 (define (get-prop full-name)
664   (match full-name
665     ([_pname _name]
666      (let ((plist (find (ref plist-list) _pname))
667            (prop (check ((theory-list? plist)
668                          (head (get-from-theory plist _name))))
669            (else (get-abstract plist _name))))))
670     (prop)))
671
672 (define (print-name full-name)
673   (match full-name
674     ([_pname _name]
675      (seq
676        (print (fold join ["\n(" (tail (val->string _pname)) " " ]
677                        []))
678        (write-val _name) (print ")"))
679      (_ (print (tail (symbol->string full-name))))))
680
681 ## (get-property P adapt tfun) finds the property P in the theory
682 ## described by tfun, and returns the adapted property (adapt P). An
683 ## error is reported on the terminal if P is not found. All
684 ## properties added to the theory when it is defined or evolved are
685 ## searched, as well all properties added to any of the theory's
686 ## ancestor theories when they are defined or evolved. If P is of the
687 ## form [id P'], where id is a meta-id, id must match a prefix of a
688 ## theory of the form [basetheory id r], where r is a renaming, and
689 ## basetheory will be searched for (r P').
690
691 define get-property :=
692   lambda (prop adapt tfun)
693   #   let {_ := (debug "prop" prop)}
694       (adapt (head (tfun prop)))
695
696 ## (!property name adapt theory) finds the pair [t p] named by name
697 ## in theory, and if t is not already in the assumption base it

```

```

698 ## attempts to prove it using p. An error is reported
699 ## on the terminal if n is not found or if t is not in the
700 ## assumption base and p is the symbol Axiom rather than a proof method.
701 ## All properties added to the theory when it is defined or
702 ## evolved are searched, as well all properties added to any of the
703 ## theory's ancestor theories when they are defined or evolved.
704 ## test-proof is the same method except that it proceeds with
705 ## the proof without checking to see if the sentence is in the
706 ## assumption base (to allow repeated execution of a proof method even if
707 ## its goal has already been proved).
708
709 (define (error-in-axiom-use method-name name prop)
710   (dlet (( _ (seq (print "\nError: ") (print method-name)
711                   (print " method applied to a sentence labeled Axiom:\n\n")
712                   (write-val name) (print "\n")
713                   (write-val prop) (error "\n"))))
714     (!true-intro)))
715
716 (define (print-theory-name theory)
717   (match theory
718     ((some-proc tfun)
719      (print (name->string tfun)))
720     ([_basetheory _prefix _newnames]
721      (seq
722        (print "[" (print-theory-name _basetheory) (print " ") (write-val _prefix)
723          (print " (renaming ") (write-val _newnames) (print "])")))))
724
725 (define (prove-property sent adapt theory)
726   (dmatch theory
727     ((some-proc tfun)
728      (dlet (([_t _p] (tfun sent))
729              (t' (adapt _t)))
730        (dcheck
731          ((holds? t') # t' is already in the AB
732           (!claim t'))
733          (else # use its accompanying proof p' to prove it
734             (dlet (( _ (seq (print "Subsidiary proof:\n")
735                             (write-val sent) (print " in ")
736                             (print-theory-name tfun) (print "\n")
737                             (write t'))))
738               (dseq
739                 (dmatch _p
740                   (Axiom (!error-in-axiom-use "property" sent t'))
741                   ( _ (conclude t' (!_p sent adapt))))
742                 (dlet (( _ (seq (print "Done with subsidiary proof of\n")
743                                 (write-val sent) (print " in ")
744                                 (print-theory-name tfun) (print "\n"))))
745                   (!claim t'))))))))
746     ([_basetheory _prefix _newnames]
747      (!prove-property (resolve sent _prefix) (change-ops adapt _newnames)
748                       _basetheory)))
749
750 (define property prove-property) # temporarily, for backward compatibility
751
752 (define (test-proof sent adapt theory)
753   let { _ := (print "\nTesting proof of ");
754         _ := match sent {
755           [_prefix _sent] =>
756             (print (join "[" (val->string _prefix) " "
757                           (head (rev (sentence-names _sent)))) "])")
758           | _ => (print (head (rev (sentence-names sent))))
759         };
760         _ := (print ":\n")}
761   match theory {
762     (some-proc tfun) =>
763       let {[_t _p] := (tfun sent);
764             t' := (adapt _t)}
765       match _p {
766         Axiom => (!error-in-axiom-use "test-proof" sent t')
767         | _ => conclude t'

```

```

768         (!_p sent adapt)
769     }
770 | [_basetheory _prefix _newnames] =>
771     (!test-proof (resolve sent _prefix) (change-ops adapt _newnames)
772       _basetheory)
773 })
774
775 (define property-test test-proof ) # temporarily, for backward compatibility
776
777 # Tests all proofs in a theory:
778
779 define test-proofs :=
780   lambda (theory)
781     (map-proc
782       lambda (name)
783         let {s := (!test-proof name no-renaming theory);
784             _ := (print "\nTheorem:\n");
785             _ := (write-val s);
786             _ := (print "\n")}
787         ()
788       (theory-theorem-names theory))
789
790 # (using lemma name) produces a binary method for proving a lemma within an
791 # implication chain. N.B.: doesn't work in an equality chain.
792
793 define using :=
794   lambda (lemma name)
795     method (p q)
796       let {L := (!lemma name)}
797       (!chain-last [p ==> q [L]])
798
799 # Instance checking procedures and a higher-order proof method.
800
801 (define (higher-order-checking)
802   (let ((checked-instances (cell []))
803         (instance-check
804          (lambda (instance theory)
805            (let ((result-list
806                  (map (lambda (prop)
807                        (let ((prop' (instance prop)))
808                          (let ((result (holds? prop')))
809                            result))))
810                  (theory-axioms theory))))
811          (negate (member? false result-list))))
812     (checked-instance?
813      (lambda (instance theory)
814        (member? [(instance 'selflist) (theory 'selfcell)]
815                  (ref checked-instances))))
816     (record-instance-check
817      (lambda (instance theory)
818        (check ((checked-instance? instance theory) ())
819               ((instance-check instance theory)
820                (set! checked-instances (add [(instance 'selflist)
821                                               (theory 'selfcell)]
822                                              (ref checked-instances))))
823              (else ())))))
824     [instance-check checked-instance? record-instance-check]))
825
826 (define hoc (higher-order-checking))
827
828 # (instance-check instance theory) returns true if for every axiom p
829 # in (theory-axioms theory), (instance p) is in the AB.
830 (define instance-check (first hoc))
831
832 # (checked-instance? instance theory) returns true if [i t], where
833 # i is (instance 'selflist) and t is (theory 'selfname), is in the
834 # recorded instances table.
835 (define checked-instance? (second hoc))
836
837 # (record-instance-check instance theory) records the pair [i t], where

```

```

838 # i is (instance 'selflist) and t is (theory 'selfname),
839 # if (instance-check instance theory) succeeds. Otherwise, does nothing.
840 (define record-instance-check (third hoc))
841
842 # print-instance-check is a version of instance-check that prints
843 # the results of the checking.
844
845 (define (print-instance-check instance theory)
846   (let ((result-list
847         (map (lambda (prop)
848               (let ((prop' (instance prop)))
849                 (seq (print "Checking\n") (write-val prop') (print "\n\n")
850                     (let ((result (holds? prop')))
851                       (seq
852                        (check ((equal? result false)
853                              (print "Error: this hasn't been proved!\n\n"))
854                        (else ()))
855                       result))))))
856         (theory-axioms theory))))
857   (negate (member? false result-list))))
858
859 # by-instance-check is a primitive method that provides a limited
860 # form of higher order proof based on theory instance checking.
861 #
862 # Suppose [T P] = (theory name). Then if instance is a checked
863 # instance of theory, (!by-instance-check name instance theory)
864 # puts T into the assumption base. If instance is not already
865 # a checked instance, it is checked, and if it passes the check
866 # it is recorded as a checked instance, and T is put into the
867 # assumption base. Otherwise, the proof attempt fails.
868 #
869 # This version narrates its operation since it is still experimental.
870
871 (primitive-method (by-instance-check name instance theory)
872   (seq (print "\nProving ") (print-prop-name name)
873        (print " by checking instance") (write (instance 'selflist))
874        (print "of theory ") (print-theory-name theory) (print "\n")
875        (check ((checked-instance? instance theory)
876               (get-property name instance theory))
877               ((instance-check instance theory)
878                (seq (print "...instance check succeeded - recording it for future use.\n")
879                    (record-instance-check instance theory)
880                    (get-property name instance theory)))
881               (else (seq (print "Failed.\n")
882                          true))))))
883
884 # (!prove-assuming M assumptions) executes nullary method M in the context of
885 # assuming every property in assumptions.
886 (define (prove-assuming M assumptions)
887   (dmatch assumptions
888     ((list-of _prop _more)
889      (assume _prop
890              (!prove-assuming M _more))))
891   ([] (!M))))
892
893 # (!check-proofs prop-names proof-method theory) executes (!proof-method name no-renaming))
894 # for each name in prop-names, in the context of assuming every property in
895 # (theory-axioms theory).
896 (define (check-proofs prop-names proof-method theory)
897   (dmatch prop-names
898     ((list-of _name _more)
899      (dlet ((_ (seq (print "\nChecking proof of ") (print-prop-name _name) (print "\n"))))
900        (dseq (!prove-assuming (method ()) (conclude (get-property _name no-renaming theory)
901                                                       (!proof-method _name no-renaming)))
902              (theory-axioms theory))
903        (!check-proofs _more proof-method theory))))
904   ([] (!true-intro))))
905
906 # This version of evolve checks the given proof method, and doesn't put theorem&proof
907 # in the theory if any of the proofs in it fails. This checking is necessary for

```

```

908 # soundness of by-instance-check. (But there is still a hole, since there is nothing
909 # to prevent the theory from being updated without checking, as in the former
910 # version of evolve. Need to plug this....)
911 (define (evolve tfun theorem&proof)
912   (let ((plist (tfun 'selfcell))
913         (theory-name (tfun 'selfname))
914         (sentences (first theorem&proof)))
915     (seq
916       (check (not-equal? (second theorem&proof) Axiom)
917         (seq
918           (set! plist (add theorem&proof (ref plist)))
919           (dtry (!check-proofs sentences (second theorem&proof) tfun)
920             (dlet (( _ (seq (set! plist (tail (ref plist)))
921                             (print "\nChecking failed; evolve not done.\n"))))
922               (!claim false))))))
923       (else
924         (set! plist (add theorem&proof (ref plist))))))
925     (map-proc (lambda (s) (index-syms [theory-name s] s))
926               sentences)
927     ())))
928
929 ## Temporarily use the old version while converting to subsorted Athena, as
930 ## it makes it somewhat easier to debug proofs.
931 ## (evolve tfun theorem&proof) adds theorem&proof to the plist in tfun and
932 ## indexes the sentences in its theorem part.
933
934 (define (evolve tfun theorem&proof)
935   (let ((plist (tfun 'selfcell))
936         (theory-name (tfun 'selfname))
937         (sentences (head theorem&proof)))
938     (seq
939       (set! plist (add theorem&proof (ref plist)))
940       (map-proc (lambda (s) (index-syms [theory-name s] s))
941                 sentences)
942       tfun)))
943
944 define chain-help :=
945   method (given L modifier)
946     letrec {insert-given := lambda (P)
947               match P {
948                 (some-sent _) => try {(given P) | P}
949                 | (some-method _) => P
950                 | (some-list L) => (map insert-given L)
951                 | _ => (given P)
952               };
953       insert-givens :=
954         lambda (L)
955           match L {
956             (split [_direction _y _P] _rest) =>
957               (add _direction (add _y (add (insert-given _P)
958                                             (insert-givens _rest))))
959             | [] => []
960           }
961     }
962     match L {
963       (list-of _t _rest) => (!generic-chain (add _t (insert-givens _rest))
964                                             modifier)
965     }
966
967 define process-fun-def :=
968   lambda (F)
969     let {_ := (map-proc
970               lambda (D)
971                 match D {
972                   [name axiom] =>
973                     (process-input-from-string
974                      (join "(define " (tail (val->string name))
975                            (val->string axiom) " ")))
976                 }
977               (second F))}

```

```

978     (map second (second F))
979
980 define process-fun-def :=
981   lambda (F)
982     let {axioms := (map second (second F))}
983     #   _ := (process-input-from-string
984     #       (join "define ["
985     #           (fold join
986     #               (map lambda (x) (join (tail (val->string x)) " "))
987     #               (map first (second F)))
988     #           [])
989     #       "]" := "
990     #       (val->string axioms)))}
991   axioms
992
993 #define abstract-fun-def := lambda (L) (process-fun-def (fun-def L))
994
995 define (proof-tools adapt Theory) :=
996   let {get := lambda (P) (get-property P adapt Theory);
997   prove := method (P) (!prove-property P adapt Theory);
998   chain := method (L) (!chain-help get L 'none);
999   chain-> := method (L) (!chain-help get L 'last);
1000   chain<- := method (L) (!chain-help get L 'first)}
1001   [get prove chain chain-> chain<-]
1002
1003 } # module propman
1004
1005 # export names of user-level procedures:
1006
1007 define [print-names print-indexed-properties pip close-props renaming
1008 no-renaming Axiom External Definition resolve change-ops
1009 using higher-order-checking hoc instance-check checked-instance?
1010 print-instance-check by-instance-check prove-assuming check-proofs
1011 chain-help] :=
1012 [propman.print-names propman.print-indexed-properties propman.pip
1013 propman.close-props propman.renaming propman.no-renaming propman.Axiom
1014 propman.External propman.Definition propman.resolve propman.change-ops
1015 propman.using propman.higher-order-checking
1016 propman.hoc propman.instance-check propman.checked-instance?
1017 propman.print-instance-check propman.by-instance-check
1018 propman.prove-assuming propman.check-proofs
1019 propman.chain-help]

```