lib/basic/property-management.ath

```
##################################################################
# Property management functions
#
# These functions provide dynamic storage, retrieval, and indexing of
# sentences according to the theory or theories with which they are
# associated.

module propman {

(define search-stack (cell []))

(define plist-list (cell []))

# (print-prop-name name) prints name, which can be either a meta-id
# or a name of form [prefix name], where prefix and name are meta-id's.

(define (print-prop-name name)
  (match name
    ([_prefix _n] (seq (print "[") (write-val _prefix) (print " ")
                       (write-val _n) (print "]")))
    (_ (write-val name))))

# The following procedures are used in reporting the error when an
# attempt is made to access a sentence via (f n) where either f has
# not been defined via concrete or theory, or, if it has, no
# sentence has been defined with name n.

(define (defined? plist name)
  (letrec ((loop (lambda (alist)
                  (match alist
                    ([] false)
                    ((split [_x _y] _rest)
                     (check ((equal? _y name) true)
                            (else (loop _rest))))))))
    (loop (ref plist))))

(define (error-name-not-defined-in plist-name prop)
#  let {_ := (print (join "\nError: in " (tail (val->string plist-name)) "\n"));
#       _ := (write prop);
#       _ := (print "not found\n");
#       _ := (!proof-error "Property not defined.\n")}
  ())

(define (error-plist-not-defined name)
  (let ((_ (!proof-error
            (join "\nError: searching for " (val->string name)
                  " in undefined property list\n"))))
    ()))

(define (get-plist-name alist)
  (match alist
    ((split [_x _y] _rest)
     (check ((equal? _y 'selfname) _x)
            (else (get-plist-name _rest))))))

# The following procedures retrieve the sentence associated with
# name in plist.  The difference is that get reports an error if it
# doesn't find the name, whereas get-abstract simply fails (thus it
# should always be used within a try expression with alternatives that
# do further searching).

(define (get plist name)
  (letrec ((loop (lambda (alist)
                  (match alist
                    ([] (check ((defined? plist-list plist)
                                (error-name-not-defined-in
                                 (get-plist-name (ref plist)) name))
```

```
                                (else (error-plist-not-defined name))))
                     ((split [_x _y] _rest)
                      (check ((equal? _y name) _x)
                             (else (loop _rest)))))))
    (loop (ref plist))))

(define (get-abstract plist name)
  (letrec ((loop (lambda (alist)
                   (match alist
                     ((split [_x _y] _rest)
                      (check ((equal? _y name) _x)
                             (else (loop _rest))))
                     ([] (halt))))))
    (loop (ref plist))))

# Data structure symbol-index is an updatable index of symbol-value
# pairs, where the value associated with a symbol is a cell containing
# the list of all names of sentences in which the symbol appears.

(define symbol-index (cell []))

(define (in-symbol-index? sym)
  (letrec ((loop (lambda (pair-list)
                   (match pair-list
                     ([] false)
                     ((split [_namelist _name] _rest)
                      (check ((equal? _name sym)
                              true)
                             (else (loop _rest)))))))
    (loop (ref symbol-index))))

## Procedure index-sym makes an entry in symbol-index: given symbol s,
## it adds name to the list of names associated with s.

(define (index-sym name sym)
  (check ((in-symbol-index? sym)
          (let ((names (get-abstract symbol-index sym)))
            (check ((member? name (ref names)) ())
                   (else (set! names (add name (ref names)))))))
         (else (set! symbol-index (add (cell [name])
                                       (add sym (ref symbol-index)))))))

## Procedure index-syms makes entries in symbol-index: for each symbol
## sym in sentence, it adds name to the list of names associated with s.

(define (index-syms name sentence)
  (map-proc (lambda (sym) (index-sym name sym))
            (get-prop-syms sentence)))

## Procedure index-syms1 makes entries in symbol-index: for each
## symbol sym in sentence, it adds sentence to the list of sentences
## associated with s.

(define (index-syms1 sentence)
  (map-proc (lambda (sym) (index-sym sentence sym))
            (get-prop-syms sentence)))

## Procedure retrieve-props returns the list of all properties
## associated with symbol sym in symbol-index, i.e., those that
## contain sym.

(define (retrieve-props sym)
  (check ((member? sym (ref symbol-index))
          (ref (get-abstract symbol-index sym)))
         (else [])))

## Procedure retrieve-props* returns the list of all indexed properties
## that each contain all of the symbols in list-of-syms.

(define (retrieve-props* list-of-syms)
```

```
  (intersection* (map retrieve-props list-of-syms)))

define Module-Table := (table 100)

(define (index-module M)
  let {M := (join "\"" M "\"")}
#    (map-proc index-syms1 (filter (map lambda (x) (try (apply-module M x) ())
#                                       (module-domain M))
#                                  prop?))
  let{_ := (process-input-from-string
             (join "(map-proc index-syms1 (filter (map lambda (x) (try (apply-module "
                   M " x) ())(module-domain " M ")) prop?))"))}
#  (table-add Module-Table [M --> (table (map lambda (x) [(try (apply-module M x) ()) x]
#                                              (module-domain M)))])
  (process-input-from-string
   (join "(table-add Module-Table [" M
         " --> (table (map lambda (x) [(try (apply-module "
         M "x) ()) x] (module-domain " M ")))])")))

define sentence-names :=
  lambda (s)
    (filter
     (map lambda (pair)
            match pair {
             [_M _T] => (try (join _M "." (table-lookup _T s)) false)
            }
          (table->list Module-Table))
     lambda (x) (negate (equal? x false)))

# Using new module indexing implemented in r1891:
define sentence-names :=
  lambda (s)
    let {names := (reverse-lookup s)}
    check {
      (equal? names []) =>
        let {_ := (build-inverted-index)}
        (reverse-lookup s)
    | else => names
    }

define print-names :=
  lambda (s)
    match (sentence-names s) {
      (list-of name rest) =>
      let {_ := (print name);
           _ := (map-proc
                 lambda (name) (print "  and " name)
                 rest)}
      (print "\n")
    }

## Procedure print-index-props (pip for short) prints all indexed
## properties (those asserted or defined in modules) that contain all
## of the symbols in syms.

define print-indexed-properties :=
  lambda (syms)
    let {_ := (print "\nStored sentences that contain  ");
         _ := match syms {
                (list-of sym rest) =>
                  let {_ := (print (symbol->string sym));
                       _ := (map-proc
                             lambda (sym) (print "  and " (symbol->string sym))
                             rest)}
                  (print ":\n")
              }}
    (map-proc
      lambda (sent)
        match (sentence-names sent) {
          (list-of name rest) =>
```

```
                   let {_ := (print name);
                        _ := (map-proc
                              lambda (name) (print "  and " name)
                              rest);
                        _ := (print ":\n  ");
                        _ := (indent-print 4 sent)}
                   (print "\n")
        }
      (retrieve-props* syms))

(define pip print-indexed-properties)

(define (close-props name-value-list)
  (letrec ((loop (lambda (L M)
                   (match L
                     ((split [_x _y] _rest)
                      (loop _rest (add (close _y) (add _x M))))
                     ([] M)))))
    (loop name-value-list [])))

#########################################################################
# Procedures for storing, retrieving, indexing, and adapting properties
# of a noncategorical theory.

## The following conventions for parameter names are used:

##  - op is a symbol used as an operator name in terms. In some
##  procedures it can also be a list of such symbols

##  - newnames is a list of the form [x_1 y_1 x_2 y_2 ...] where the
##  x_i and y_i are symbols used as operator names in terms. Its
##  meaning in these procedures is that y_i is a replacement for x_i.

## (replace op newnames): If op is a symbol, returns the replacement
## for op specified in newnames, if any (i.e., if op occurs as an x_i
## in newnames then y_i, otherwise op).  If op is a list of symbols,
## returns their replacements as specified in newnames.

(define (replace op newnames)
  (match op
    ((some-list L) (map (lambda (x) (replace x newnames)) L))
    (_ (match newnames
         ([] op)
         ((split [x y] rest)
          (check ((equal? x op) y)
                 (else (replace op rest))))))))

## (term-adapt term newnames): returns the term resulting from
## replacing each operator symbol in term as specified in newnames.

(define (term-adapt term newnames)
  (match term
    ((op (some-list args))
     (make-term (replace op newnames)
      (map (lambda (t) (term-adapt t newnames)) args)))
    ((some-var v) v)
    ((some-symbol x) (replace x newnames))))

## (prop-adapt term newnames): returns the sentence resulting
## from replacing each operator symbol in prop as specified in
## newnames.

(define (prop-adapt prop newnames)
  (match prop
    ((some-var v) v)
    ((not _P) (not (prop-adapt _P newnames)))
    (((some-sent-con pc) _P1 _P2)
     (pc (prop-adapt _P1 newnames) (prop-adapt _P2 newnames)))
    (((some-quant quant) _v _B)
     (quant _v (prop-adapt _B newnames)))
```

```
    (_ (term-adapt prop newnames))))

## (renaming newnames): returns a procedure r such that (r p), where p
## is a sentence, returns the sentence that results from
## replacing the operators in p as specified in newnames.  Or if p is
## a list of sentences [p_1 p_2 ...], r returns [(r p_1) (r p_2)
## ...].  Lastly, if p is the meta-id 'selflist, r returns newnames.

(define (renaming-core newnames)
  (lambda (p)
#    (let ((_ (print "B:")) (_ (write p)) (_ (write newnames)))
      (match p
        ((some-list L) (map (lambda (p) (prop-adapt p newnames)) L))
        ('selflist newnames)
        (_ (prop-adapt p newnames)))))
#)

(define (renaming newnames)
  (match newnames
    ((some-map m) (renaming-core (map get-symbol (flatten (map-key-values m)))))
    (_ (renaming-core (map get-symbol newnames)))))

(define no-renaming
  (renaming []))

## Symbol Axiom is used to mark the sentences defined in a pfun as
## axioms (i.e., must be either asserted or proved outside of the
## theory in which they appear).  External can be used as a synonym
## for Axiom; so can Definition.

(declare Axiom Boolean)

(define External Axiom)

(define Definition Axiom)

## (resolve name prefix): If name is of the form [pref n] and pref
## equals prefix, returns n, otherwise fails.

(define (resolve name prefix)
  (match name
    ([_first _second]
     (check
      ((equal? _first prefix)
       _second)))))

## (find name lst) searches lst, of the form [n_1 v_1 n_2 v_2 ...] for
## the first n_i that equals name and returns the corresponding v_i.
## It fails if there is no such n_i.

(define (find lst name)
  (match lst
    ((split [_name1 _value] _rest)
     (check ((equal? name _name1)
             _value)
            (else (find _rest name))))))

## (change-ops adapt newnames) returns a new adapter procedure that
## first renames according to the one defined by newnames, then
## according to the given adapter

(define (change-ops adapt newnames)
  (o adapt (renaming newnames)))

# For several of the procedures defined below, the following data
# structures are used.
#
## A tp-list is a list of the form [[f_1 p_1] [f_2 p_2] ... [f_j p_j]
## 'superiors t_1 t_2 ... t_k], where each f_i is a pfun defined with the
## abstract procedure, each p_i is either the symbol Axiom or a proof
```

```
## method, and each t_i is a tfun or an adapted theory. Either j = 0
## or k = 0 is permitted, but not both.

## A tfun is a procedure that encapsulates a tp-list and also is
## reflective: (tfun 'selfname) returns the name (a meta-id) given to
## the procedure when it was defined, and (pfun 'selfcell) returns its
## tp-list.

## An adapted theory is list of the form [basetheory prefix newnames],
## where basetheory is a tfun, prefix is a name, and newnames is a
## list of operator replacements as previously described.

## A tp-list is constructed by the theory procedure and is searched by
## get-from-theory (which is used by the theory procedure).

## (get-from-theory tp-list n): If f_i is the first pfun in tp-list
## such that (f_i n) is defined, returning t_i, then the pair [t_i
## p_i] is returned.  If (f_i n) fails for every f_i, the ancestor
## theories are searched.  An ancestor that is a tfun is searched by
## searching its tp-list recursively, while one of the form
## [basetheory prefix newnames] is searched by searching for the name
## (resolve name prefix) in basetheory, and if [t p] is thus returned
## then [t' p'] is the pair returned, where t' is ((renaming newnames)
## t) and p' is
#
## (method (name adapt)
##   (!p (resolve name prefix) (change-ops adapt newnames)))

define get-from-theory :=
  lambda (tp-list prop)
    letrec {loop :=
             lambda (alist)
               match alist {
                 (list-of _a _rest) =>
                   match _a {
                     [_x _y] => try {[check {(member? prop _x) => prop}
                                      _y]
                                     | (loop _rest)
                                     }
                   | 'superiors => (ancestor-loop _rest)
                   }
               };
             ancestor-loop :=
               lambda (theories)
                 match theories {
                   (list-of _th _more) =>
                     match _th {
                       (some-proc th) =>
                         (try (get-from-theory (th 'selfcell) prop)
                              (ancestor-loop _more))
                     | [_basetheory _prefix _newnames] =>
                       let {prop' := (resolve prop _prefix)}
                       try {
                         match (_basetheory prop') {
                           [_t _p] =>
                             let {t' := ((renaming _newnames) _t)}
                             [t' match _p {
                                   Axiom => Axiom
                                 | _ => method (prop adapt)
                                           (!_p prop'
                                                (change-ops adapt
                                                            _newnames))
                                 }]
                         } # match
                       | (ancestor-loop _more)
                       } # try
                     } # match
                   } # match
           } #letrec
    (loop (ref tp-list))
```

```
## (index-theory theory-name tp-list) indexes the tp-list's toplevel
## property lists L_i.  It ignores ancestor theories.

(define (index-theory theory-name tp-list)
  (match tp-list
    ((list-of [_L _p] _rest)
     (seq (map-proc index-syms1 _L)
          (index-theory theory-name _rest)))
    ((list-of 'superiors _rest) ())
    ([] ())))

## (theory-properties tfun) returns the sentences in
## the plist in tfun (at the top level and in superiors), an ralist.

(define (theory-properties tfun)
  (letrec ((helper (lambda (alist)
                     (match alist
                       ((list-of _x _rest)
                        (match _x
                          ([_t _p] (join _t (helper _rest)))
                          ('superiors (fold join
                                            (map theory-properties _rest)
                                            []))))
                       ([] [])))))
    (match tfun
      ((some-proc f)
       (helper (ref (f 'selfcell))))
      ([_basetheory _prefix _newnames]
       (let ((adapt (renaming _newnames)))
         (map (lambda (x)
                (match x
                  ((some-sent p) (adapt p))
                  (_ (add _prefix [x]))))
              (theory-properties _basetheory)))))))

## (index-adapted theory-name superiors) indexes each tfun or adapted
## theory in superiors.

(define (index-adapted theory-name superiors)
  (letrec ((loop1
            (lambda (basetheory-name prefix newnames)
              (match newnames
                ((split [_n _n'] _rest)
                 (seq
                  (loop2 (retrieve-props _n) basetheory-name prefix _n')
                  (loop1 basetheory-name prefix _rest)))
                ([] ()))))
           (loop2 (lambda (prop-names basetheory-name prefix n')
                    (match prop-names
                      ((list-of [_pname _p] _more)
                       (seq
                        (check ((equal? _pname basetheory-name)
                                (index-sym [theory-name [prefix _p]] n'))
                               (else ()))
                        (loop2 _more  basetheory-name prefix n')))
                      ([] ())))))
    (match superiors
      ((list-of _th _rest)
       (seq (match _th
              ((some-proc p) ())
              ([_basetheory _prefix _newnames]
               (loop1 (_basetheory 'selfname) _prefix _newnames)))
            (index-adapted theory-name _rest)))
      ([] ()))))

(define (index-adapted theory-name superiors)
  (letrec ((loop
            (lambda (tprops th)
              (match tprops
```

```
                ((split [_pname _prop] _rest)
                 (seq
                  (match th
                    ([_basetheory _prefix _newnames]
                     (index-syms [theory-name _pname] ((renaming _newnames) _prop))))
                  (loop _rest th)))
                ([] ()))))))
    (match superiors
      ((list-of _th _rest)
       (seq (match _th
              ((some-proc p) ())
              ([_basetheory _prefix _newnames]
               (loop (theory-properties _th) _th)))
            (index-adapted theory-name _rest)))
      ([] ())))))

## (theory superiors tp-list theory-name) defines a tfun with name theory-name,
## toplevel pairs [f_i p_i] as given in tp-list, and ancestor theories as
## given in superiors.

define theory :=
  lambda (superiors tp-list theory-name)
  let {new-plist := (cell (add 'superiors superiors));
       _ := (set! plist-list (add theory-name (add new-plist (ref plist-list))));
       _ := match tp-list {
              (some-list p) => (set! new-plist (add [p Axiom] (ref new-plist)))
            | (some-proc p) => (set! new-plist (add p (ref new-plist)))
            };
#       _ : = (index-adapted theory-name superiors);
       pfun := lambda (prop)
                 match prop {
                   'selfname => theory-name
                 | 'selfcell => new-plist
                 | _ => try {let {_ := (set! search-stack (add [theory-name prop]
                                                               (ref search-stack)))}
                               (get-from-theory new-plist prop)
                            |
                              ()
                            }
                 }
      }
  pfun

## (theory-axioms theory) returns a list of the properties (without
## their names) that are labelled Axiom in theory, which may be either
## a tfun or an adapted theory.

(define (theory-axioms theory)
  (letrec ((helper (lambda (alist)
                     (match alist
                       ((list-of _x _rest)
                        (match _x
                          ([_t Axiom] (join _t (helper _rest)))
                          ([_ _] (helper _rest))
                          ('superiors (fold join
                                            (map theory-axioms _rest)
                                            []))))
                       ([] [])))))
    (match theory
      ((some-proc tfun)
       (remove-duplicates (helper (ref (tfun 'selfcell)))))
      ([_basetheory _prefix _newnames]
       (let ((adapt (renaming _newnames)))
         (map (lambda (x)
                (match x
                  ((some-sent p) (adapt p))
                  (_ (add _prefix [x]))))
              (theory-axioms _basetheory)))))))
```

```
## (theory-axiom-names theory) returns a list of the names of
## properties that are labelled Axiom in theory, which may be either
## a tfun or an adapted theory.

(define (theory-axiom-names theory)
  (letrec ((helper (lambda (alist)
                     (match alist
                       ((list-of _x _rest)
                        (match _x
                          ([_t Axiom] (join [_t] (helper _rest)))
                          ([_ _] (helper _rest))
                          ('superiors (fold join
                                            (map theory-axiom-names _rest)
                                            []))))
                       ([] [])))))
    (match theory
      ((some-proc tfun)
       (remove-duplicates (helper (ref (tfun 'selfcell)))))
      ([_basetheory _prefix _newnames]
       (let ((adapt (renaming _newnames)))
         (map (lambda (x)
                (match x
                  ((some-sent p) (adapt p))
                  (_ (add _prefix [x]))))
              (theory-axiom-names _basetheory)))))))

(define (theory-theorem-names theory)
  (letrec ((helper (lambda (alist)
                     (match alist
                       ((list-of _x _rest)
                        (match _x
                          ([_ Axiom] (helper _rest))
                          ([_t _] (join [_t] (helper _rest)))
                          ('superiors (fold join
                                            (map theory-theorem-names _rest)
                                            []))))
                       ([] [])))))
    (match theory
      ((some-proc tfun)
       (remove-duplicates (helper (ref (tfun 'selfcell)))))
      ([_basetheory _prefix _newnames]
       (let ((adapt (renaming _newnames)))
         (map (lambda (x)
                (match x
                  ((some-sent p) (adapt p))
                  (_ (add _prefix [x]))))
              (theory-theorem-names _basetheory)))))))

## theory-clone, for defining a theory by adapting a single theory,
## and adding no new axioms.

(define (theory-clone adapted-theory theory-name)
  (theory [adapted-theory] [] theory-name))

## (print-theory theory) prints on the terminal a list of the sentences
## (with their names) in theory, which may be either a tfun or an adapted
## theory.

define sentence-name :=
  lambda (M s)
    (join M "." (table-lookup (table-lookup Module-Table M) s))

# Using new module indexing implemented in r1891:
define sentence-name :=
  lambda (M s)
    let {unqualified-name := match (head (rev (sentence-names s))) {
                                (split _ (list-of `. name)) => name
                              }}
    (join M "." unqualified-name)
```

```
(define (name->string pfun)
  (tail (val->string (pfun 'selfname))))

(define (print-theory theory)
  letrec {helper := lambda (alist theory-name)
                      match alist {
                        (list-of _x _rest) =>
                        match _x {
                          [_t _p] =>
                          let {_ := (map-proc
                                      lambda (s)
                                        (print "\n" (join (sentence-name theory-name s)
                                                          ":\n" (val->string s) "\n\n"))
                                      _t)}
                                     (helper _rest theory-name)
                        | 'superiors => (map-proc print-theory _rest)
                        | _ => ()
                        }
                     }}
  match theory {
    (some-proc f) => let {_ := (print "\n");
                          _ := (print  (join (name->string f) ":\n\n"))}
                     (helper (ref (f 'selfcell)) (name->string f))
  | [basetheory prefix newnames] =>
    (map-proc lambda (x)
                let {_ := (print (sentence-name (name->string basetheory) x));
                     _ := (print ":\n");
                     _ := (write-val ((renaming newnames) x))}
                (print "\n\n")
              (theory-properties basetheory))
  })

(define (theory-list? plist)
  (member? 'superiors (ref plist)))

(define (get-prop full-name)
  (match full-name
    ([_pname _name]
     (let ((plist (find (ref plist-list) _pname))
           (prop (check ((theory-list? plist)
                         (head (get-from-theory plist _name)))
                        (else (get-abstract plist _name)))))
       prop))))

(define (print-name full-name)
  (match full-name
    ([_pname _name]
     (seq
      (print (fold join ["\n(" (tail (val->string _pname)) " "]
                  []))
      (write-val _name) (print ")")))
    (_ (print (tail (symbol->string full-name))))))

## (get-property P adapt tfun) finds the property P in the theory
## described by tfun, and returns the adapted property (adapt P).  An
## error is reported on the terminal if P is not found.  All
## properties added to the theory when it is defined or evolved are
## searched, as well all properties added to any of the theory's
## ancestor theories when they are defined or evolved.  If P is of the
## form [id P'], where id is a meta-id, id must match a prefix of a
## theory of the form [basetheory id r], where r is a renaming, and
## basetheory will be searched for (r P').

define get-property :=
  lambda (prop adapt tfun)
#    let {_ := (debug "prop" prop)}
    (adapt (head (tfun prop)))

## (!property name adapt theory) finds the pair [t p] named by name
## in theory, and if t is not already in the assumption base it
```

```
## attempts to prove it using p. An error is reported
## on the terminal if n is not found or if t is not in the
## assumption base and p is the symbol Axiom rather than a proof method.
## All properties added to the theory when it is defined or
## evolved are searched, as well all properties added to any of the
## theory's ancestor theories when they are defined or evolved.
## test-proof is the same method except that it proceeds with
## the proof without checking to see if the sentence is in the
## assumption base (to allow repeated execution of a proof method even if
## its goal has already been proved).

(define (error-in-axiom-use method-name name prop)
  (dlet ((_ (seq (print "\nError: ")  (print method-name)
                 (print " method applied to a sentence labeled Axiom:\n\n")
                 (write-val name) (print "\n")
                 (write-val prop) (error "\n"))))
    (!true-intro)))

(define (print-theory-name theory)
  (match theory
    ((some-proc tfun)
     (print (name->string tfun)))
    ([_basetheory _prefix _newnames]
     (seq
      (print "[") (print-theory-name _basetheory) (print " ") (write-val _prefix)
      (print "  (renaming ") (write-val _newnames) (print ")]")))))

(define (prove-property sent adapt theory)
  (dmatch theory
    ((some-proc tfun)
        (dlet (([_t _p] (tfun sent))
               (t' (adapt _t)))
          (dcheck
           ((holds? t')  # t' is already in the AB
            (!claim t'))
           (else # use its accompanying proof p' to prove it
            (dlet ((_ (seq (print "Subsidiary proof:\n")
                           (write-val sent) (print " in ")
                           (print-theory-name tfun) (print "\n")
                           (write t'))))
              (dseq
               (dmatch _p
                 (Axiom (!error-in-axiom-use "property" sent t'))
                 (_ (conclude t' (!_p sent adapt))))
               (dlet ((_ (seq (print "Done with subsidiary proof of\n")
                              (write-val sent) (print " in ")
                              (print-theory-name tfun) (print "\n"))))
                 (!claim t'))))))))
    ([_basetheory _prefix _newnames]
     (!prove-property (resolve sent _prefix) (change-ops adapt _newnames)
                      _basetheory))))

(define property prove-property)  # temporarily, for backward compatibility

(define (test-proof sent adapt theory)
 let {_ := (print "\nTesting proof of ");
      _ := match sent {
             [_prefix _sent] =>
             (print (join "[" (val->string _prefix) " "
                          (head (rev (sentence-names _sent))) "]"))
           | _ => (print (head (rev (sentence-names sent))))
           };
      _ := (print ":\n")}
   match theory {
     (some-proc tfun) =>
       let {[_t _p] := (tfun sent);
            t' := (adapt _t)}
       match _p {
         Axiom => (!error-in-axiom-use "test-proof" sent t')
       | _ => conclude t'
```

```
                  (!_p sent adapt)
        }
    | [_basetheory _prefix _newnames] =>
      (!test-proof (resolve sent _prefix) (change-ops adapt _newnames)
                   _basetheory)
})

(define property-test test-proof )  # temporarily, for backward compatibility

# Tests all proofs in a theory:

define test-proofs :=
  lambda (theory)
    (map-proc
      lambda (name)
        let {s := (!test-proof name no-renaming theory);
             _ := (print "\nTheorem:\n");
             _ := (write-val s);
             _ := (print "\n")}
        ()
      (theory-theorem-names theory))

# (using lemma name) produces a binary method for proving a lemma within an
# implication chain. N.B.: doesn't work in an equality chain.

define using :=
  lambda (lemma name)
    method (p q)
      let {L := (!lemma name)}
       (!chain-last [p ==> q [L]])

# Instance checking procedures and a higher-order proof method.

(define (higher-order-checking)
  (let ((checked-instances (cell []))
        (instance-check
         (lambda (instance theory)
           (let ((result-list
                  (map (lambda (prop)
                         (let ((prop' (instance prop)))
                           (let ((result (holds? prop')))
                             result)))
                       (theory-axioms theory))))
             (negate (member? false result-list)))))
        (checked-instance?
         (lambda (instance theory)
           (member? [(instance 'selflist) (theory 'selfcell)]
                    (ref checked-instances))))
        (record-instance-check
         (lambda (instance theory)
           (check ((checked-instance? instance theory) ())
                  ((instance-check instance theory)
                   (set! checked-instances (add [(instance 'selflist)
                                                 (theory 'selfcell)]
                                                (ref checked-instances))))
                  (else ())))))
    [instance-check checked-instance? record-instance-check]))

(define hoc (higher-order-checking))

# (instance-check instance theory) returns true if for every axiom p
# in (theory-axioms theory), (instance p) is in the AB.
(define instance-check (first hoc))

# (checked-instance? instance theory) returns true if [i t], where
# i is (instance 'selflist) and t is (theory 'selfname), is in the
# recorded instances table.
(define checked-instance? (second hoc))

# (record-instance-check instance theory) records the pair [i t], where
```

```
# i is (instance 'selflist) and t is (theory 'selfname),
# if (instance-check instance theory) succeeds. Otherwise, does nothing.
(define record-instance-check (third hoc))

# print-instance-check is a version of instance-check that prints
# the results of the checking.

(define (print-instance-check instance theory)
  (let ((result-list
         (map (lambda (prop)
                (let ((prop' (instance prop)))
                  (seq (print "Checking\n") (write-val prop') (print "\n\n")
                       (let ((result (holds? prop')))
                         (seq
                          (check ((equal? result false)
                                  (print "Error: this hasn't been proved!\n\n"))
                                 (else ()))
                          result)))))
              (theory-axioms theory))))
    (negate (member? false result-list))))

# by-instance-check is a primitive method that provides a limited
# form of higher order proof based on theory instance checking.
#
# Suppose [T P] = (theory name).  Then if instance is a checked
# instance of theory, (!by-instance-check name instance theory)
# puts T into the assumption base.  If instance is not already
# a checked instance, it is checked, and if it passes the check
# it is recorded as a checked instance, and T is put into the
# assumption base.  Otherwise, the proof attempt fails.
#
# This version narrates its operation since it is still experimental.

(primitive-method (by-instance-check name instance theory)
  (seq (print "\nProving ") (print-prop-name name)
       (print " by checking instance") (write (instance 'selflist))
       (print "of theory ") (print-theory-name theory) (print "\n")
       (check ((checked-instance? instance theory)
               (get-property name instance theory))
              ((instance-check instance theory)
               (seq (print "...instance check succeeded - recording it for future use.\n")
                    (record-instance-check instance theory)
                    (get-property name instance theory)))
              (else (seq (print "Failed.\n")
                         true)))))

# (!prove-assuming M assumptions) executes nullary method M in the context of
# assuming every property in assumptions.
(define (prove-assuming M assumptions)
  (dmatch assumptions
    ((list-of _prop _more)
     (assume _prop
       (!prove-assuming M _more)))
    ([] (!M))))

# (!check-proofs prop-names proof-method theory) executes (!proof-method name no-renaming))
# for each name in prop-names, in the context of assuming every property in
# (theory-axioms theory).
(define (check-proofs prop-names proof-method theory)
  (dmatch prop-names
    ((list-of _name _more)
     (dlet ((_ (seq (print "\nChecking proof of ") (print-prop-name _name) (print "\n"))))
       (dseq (!prove-assuming (method () (conclude (get-property _name no-renaming theory)
                                            (!proof-method _name no-renaming)))
                              (theory-axioms theory))
             (!check-proofs _more proof-method theory))))
    ([] (!true-intro))))

# This version of evolve checks the given proof method, and doesn't put theorem&proof
# in the theory if any of the proofs in it fails.  This checking is necessary for
```

```
# soundness of by-instance-check.  (But there is still a hole, since there is nothing
# to prevent the theory from being updated without checking, as in the former
# version of evolve.  Need to plug this....)
(define (evolve tfun theorem&proof)
  (let ((plist (tfun 'selfcell))
        (theory-name (tfun 'selfname))
        (sentences (first theorem&proof)))
    (seq
     (check ((not-equal? (second theorem&proof) Axiom)
             (seq
              (set! plist (add theorem&proof (ref plist)))
              (dtry (!check-proofs sentences (second theorem&proof) tfun)
                    (dlet ((_ (seq (set! plist (tail (ref plist)))
                                   (print "\nChecking failed; evolve not done.\n"))))
                      (!claim false)))))
            (else
             (set! plist (add theorem&proof (ref plist)))))
     (map-proc (lambda (s) (index-syms [theory-name s] s))
               sentences)
     ())))

## Temporarily use the old version while converting to subsorted Athena, as
## it makes it somewhat easier to debug proofs.
## (evolve tfun theorem&proof) adds theorem&proof to the plist in tfun and
## indexes the sentences in its theorem part.

(define (evolve tfun theorem&proof)
  (let ((plist (tfun 'selfcell))
        (theory-name (tfun 'selfname))
        (sentences (head theorem&proof)))
    (seq
     (set! plist (add theorem&proof (ref plist)))
     (map-proc (lambda (s) (index-syms [theory-name s] s))
               sentences)
     tfun)))

define chain-help :=
  method (given L modifier)
    letrec {insert-given := lambda (P)
                              match P {
                                (some-sent _) => try {(given P) | P}
                              | (some-method _) => P
                              | (some-list L) => (map insert-given L)
                              | _ => (given P)
                              };
            insert-givens :=
              lambda (L)
                match L {
                  (split [_direction _y _P] _rest) =>
                    (add _direction (add _y (add (insert-given _P)
                                                 (insert-givens _rest))))
                | [] => []
                }
            }
    match L {
      (list-of _t _rest) => (!generic-chain (add _t (insert-givens _rest))
                                            modifier)
    }

define process-fun-def :=
  lambda (F)
    let {_ := (map-proc
                lambda (D)
                  match D {
                    [name axiom] =>
                    (process-input-from-string
                      (join "(define " (tail (val->string name))
                            (val->string axiom) ")"))
                  }
                (second F))}
```

```
     (map second (second F))

define process-fun-def :=
  lambda (F)
    let {axioms := (map second (second F))}
#        _ := (process-input-from-string
#              (join "define ["
#                    (fold join
#                          (map lambda (x) (join (tail (val->string x)) " ")
#                               (map first (second F)))
#                          [])
#                    "] := "
#                    (val->string axioms)))}
    axioms

#define abstract-fun-def :=  lambda (L) (process-fun-def (fun-def L))

define (proof-tools adapt Theory) :=
   let {get := lambda (P) (get-property P adapt Theory);
        prove := method (P) (!prove-property P adapt Theory);
        chain := method (L) (!chain-help get L 'none);
        chain-> := method (L) (!chain-help get L 'last);
        chain<- := method (L) (!chain-help get L 'first)}
     [get prove chain chain-> chain<-]

} # module propman

# export names of user-level procedures:

define [print-names print-indexed-properties pip close-props renaming
no-renaming Axiom External Definition resolve change-ops
using higher-order-checking hoc instance-check checked-instance?
print-instance-check by-instance-check prove-assuming check-proofs
chain-help] :=
[propman.print-names propman.print-indexed-properties propman.pip
 propman.close-props propman.renaming propman.no-renaming propman.Axiom
 propman.External propman.Definition propman.resolve propman.change-ops
 propman.using propman.higher-order-checking
 propman.hoc propman.instance-check propman.checked-instance?
 propman.print-instance-check propman.by-instance-check
 propman.prove-assuming propman.check-proofs
 propman.chain-help]
```