lib/basic/st.ath

```
module ST {

define (renaming m) :=
let {m := (Map.apply-to-both m get-symbol)}
  lambda (x)
      match x {
      | (some-symbol c) => (Map.apply-or-same m c)
      | ((some-symbol f) (some-list terms)) => (make-term (Map.apply-or-same m f) (map (renaming m) terms))
      | (some-var _) => x
      | ((some-sent-con sc) (some-list props)) => (sc (map (renaming m) props))
      | ((some-quant q) (some-var x) body) => (q x ((renaming m) body))
      | (some-list L) => (map (renaming m) L)
      | _ => x
      }
define (no-renaming x) := x
define theory-index := (HashTable.table 100)

# so we can pick out a theory either by name or as a value:
define (metaid->string x) := check {(meta-id? x) => (id->string x) | else => x}
define (get-theory th) := try {(HashTable.lookup theory-index (metaid->string th)) | th}

define (make-theory superiors axioms) :=
  let {name := (separate (mod-path) ".");
       th := |{'superiors := (map get-theory superiors),
               # Hash table mapping each axiom p to 'AXIOM:
               'axioms := (pairs->table (map lambda (p) [p 'AXIOM] axioms)),
               # Hash table mapping each theorem p to a method that derives it:
               'theorems := (table 50),
               'adapted := |{}|,
               'name := name}|;
       _ := (HashTable.add theory-index [name --> th])}
    th

private define name        := lambda (th) (th 'name)
define (superiors th)      := (th 'superiors)
private define axiom-table := lambda (th) (th 'axioms)
define (top-axioms th)     := (HashTable.keys (axiom-table (get-theory th)))
define (theorem-table th)  := (th 'theorems)
define (top-theorems th)   := (HashTable.keys (theorem-table (get-theory th)))
define (adapted? th)       := (negate (Map.empty? ((get-theory th) 'adapted)))
define (get-symbol-map th) := (((get-theory th) 'adapted) 'symbol-map)
define (get-renaming th)   := (renaming (get-symbol-map (get-theory th)))
define (original-name th)  := check {(adapted? th) => ((th 'adapted) 'original-name)
                                    | else => (name th)}

define (theory-name th) := (name (get-theory th))
define get-adapter := get-renaming

private define all-axioms :=
  lambda (th)
    let {all := (join (top-axioms th)
                      (flatten (map all-axioms (superiors th))))}
      check {(adapted? th) => ((get-renaming th) all)
           | else => all}

define (theory-axioms th) :=
  (all-axioms (get-theory th))

private define all-theorems :=
  lambda (th)
    let {all := (join (top-theorems th)
                      (flatten (map all-theorems (superiors th))))}
      check {(adapted? th) => ((get-renaming th) all)
           | else => all}

define (theory-theorems th) :=
  (all-theorems (get-theory th))
```

```
define (make-adapted-theory th sym-map) :=
  let {[th new-name] := [(get-theory th) (separate (mod-path) ".")];
       res := |{'superiors := (superiors th),
                'axioms    := (axiom-table th),
                'theorems  := (theorem-table th),
                'adapted   := |{'original-name := (name th), 'symbol-map := sym-map}|,
                'name      := new-name }|;
       _ := (HashTable.add theory-index [new-name --> res])}
    res

define adapt-theory := make-adapted-theory

define add-edge :=
  let {mem := (HashTable.table 100)}
    lambda (G name1 name2 i)
      check {([name1 name2] HashTable.in mem) => ()
           | else => let {_ := (HashTable.add mem [[name1 name2] --> true])}
                          (Graph-Draw.add-edge G name1 name2 i)}

define (make-theory-graph G counter) :=
  lambda (th)
    let {th := (get-theory th);
         T := (name th);
         _ := (Graph-Draw.add-node G T);
         _ := (map-proc (make-theory-graph G counter) (superiors th));
         _ := check {(adapted? th) => (add-edge G (original-name th) T (inc counter)) | else => ()}}
      (map-proc lambda (sup) (add-edge G (name sup) T (inc counter))
                (superiors th))

define (draw-theory th) :=
  let {G := (Graph-Draw.make-graph 0);
       counter := (cell 0);
       _ := ((make-theory-graph G counter) th)}
    (Graph-Draw.draw-and-show G Graph-Draw.viewer)

define (draw-all-theories) :=
  let {G := (Graph-Draw.make-graph 0);
       counter := (cell 0);
       _ := (map-proc (make-theory-graph G counter)
                      (rev (HashTable.keys theory-index)))}
    (Graph-Draw.draw-and-show G Graph-Draw.viewer)

define (add-axiom th) :=
  lambda (p) (HashTable.add (axiom-table (get-theory th)) [p --> 'AXIOM])

define (add-axioms th new-axioms) := (map-proc (add-axiom th) new-axioms)

define (find-in-theory p) :=
  lambda (th)
    try {(HashTable.lookup (axiom-table th) p)
       | (HashTable.lookup (theorem-table th) p)
       | (first-image (superiors th) (find-in-theory p))}

define (get-from-theory th p) :=
   let {th := (get-theory th)}
    ((find-in-theory p) th)

define (get-property p adapter th) :=
  let {_ := (get-from-theory th p);
       p := check {(adapted? th) => ((get-renaming th) p) | else => p}}
    (adapter p)

define (test-proof th) :=
 let {th := (get-theory th)}
 lambda (p)
   let {_ := (print "\nTesting proof of:\n" p "...\n")}
   match (get-from-theory th p) {
     'AXIOM => (print "\nThis is an axiom:\n" p)
   | (some-method M) =>
      let {error-msg := (cell ());
```

```
           _ := (!dcatch method ()
                           assume (and* (theory-axioms th))
                             conclude p (!M p no-renaming)
                          method (str)
                            let {_ := (set! error-msg str)}
                              (!true-intro))}
        check {(equal? (ref error-msg) ()) => (print "\nProof worked.\n")
             | else => (print "\nProof failed: " (ref error-msg) "\n")}
   }

define (test-proofs props th) := (map-proc (test-proof th) props)

define (test-all-proofs th) :=
  let {th := (get-theory th)}
    (test-proofs (top-theorems th) th)

define (proof-method-works? p M th) := true

define (add-if-proof-method-works M th) :=
  lambda (p)
     check {(proof-method-works? p M th) => (HashTable.add (theorem-table th) [p --> M])}

define (add-theorems th m) :=
  let {th := (get-theory th)}
    (map-proc lambda (pair)
                match pair {
                  [(some-sent p) M] => ((add-if-proof-method-works M th) p)
                | [(some-list L) M] => (map-proc (add-if-proof-method-works M th) L)
                }
              (Map.key-values m))

define (theory-axiom? th p) := (p HashTable.in (axiom-table (get-theory th)))

define chain-help := chain-transformer

define (prove-property p adapt th) :=
  let {th := (get-theory th);
       M :=  (get-from-theory th p);
       adapt := check {(adapted? th) => (o adapt (get-renaming th)) | else => adapt};
       q := (adapt p)}
    check {((holds? q) || (equal? M 'AXIOM)) => (!claim q)
         | else => (!M p adapt)}


define (proof-tools adapter th) :=
   let {th := (get-theory th);
        get := lambda (p) (get-property p adapter th);
        prove := method (p) (!prove-property p adapter th);
        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<-]

define (print-instance-check renamer th) :=
  (map-proc lambda (p)
              let {p := (renamer p);
                   _ := (print "\nChecking\n" (val->string p) "\n")}
                check {(holds? p) => ()
                      | else => (print "\nError: This has not been proved!\n\n")}
            (theory-axioms th))

define (print-theory th) :=
  let {_ := (print "\n");
       _ := (print (theory-name th));
       _ := (print ".theory:\n\nAxioms:\n");
       _ := (map-proc write (theory-axioms th));
       _ := (print "\nTheorems:\n")}
    (map-proc write (theory-theorems th))

} # module ST
```

```
open ST

EOF
(load "st")
```