lib/basic/smt.ath

```
#;;;;;;;;; Athena code for using Yices ;;;;;;;;;

## Hash-table management for the mappings between the
## Athena and the Yices worlds:

(define (make-empty-map) [(make-term-hash-table) (make-hash-table)])

(define (add-binding x y map)
  (match map
    ([term-ht string-ht] (seq (match x
                                ((some-term _) (term-enter term-ht x y))
                                (_ (enter string-ht x y)))
                              map))))

(define (remove-binding key map)
  (match map
    ([term-ht string-ht] (match key
                           ((some-term _) (seq (term-table-remove term-ht key)
                                               map))
                           (_ (seq (remove string-ht key)
                                   map))))))

(define (many-strings->one-string string-list)
  (foldl (lambda (x y) (join x "\n" y)) [] string-list))

(define (apply-map map x)
  (match [map x]
    ([[term-ht _] (some-term _)] (term-look-up term-ht x))
    ([[_ string-ht] _] (look-up string-ht x))))


(define (dom m)
  (match m
    ([term-ht string-ht] (let ((string-entries (show-table string-ht))
                               (term-entries   (show-table term-ht)))
                           (join (map first term-entries)
                                 (map first string-entries))))))

(define (map-range m)
  (match m
    ([term-ht string-ht] (let ((string-entries (show-table string-ht))
                               (term-entries   (show-table term-ht)))
                           (join (map second term-entries)
                                 (map second string-entries))))))

(define (dom-range-list m)
  (match m
    ([term-ht string-ht] (let ((term-entries   (show-table term-ht))
                               (string-entries (show-table string-ht)))
                           (join term-entries string-entries)))))


(define (in-dom? a m)
  (match (apply-map m a)
    (() false)
    (_ true)))

#### Pre-defined numeric relations and SMT code:

#(declare (<= < >= >) (-> (Real Real) Boolean))

#;;;;;;;;;;;;;;;;;;;; SOME GLOBAL OPTIONS ;;;;;;;;;;;;;;;;;;;

#;;;;;;;;; The default time limit for any single call to Yices
#;;;;;;;;; is currently 60 seconds. This can be changed as desired.
#;;;;;;;;; Simply do (set! yices-time-limit "n"), where n is the
#;;;;;;;;; desired number of seconds:

```

```
(define yices-time-limit (cell "60"))

# By default, Athena removes excess/superfluous information from the output model
# in order to simplify its presentation. To get the full output model, set the
# global variable show-whole-model? to true:

(define show-whole-model? (cell false))

#;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
#;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


#;;;;;;;; Uncomment the following definition of mprint in order
#;;;;;;;; to get comments and statistics from the various procedures:

# (define (mprint str) (print str))

(define (mprint str) ())

#;;;;;;;; Setting up the translation from Athena to Yices:

(define [bar comma lparen rparen lbrack rbrack blank colon scolon quot-mark]
        [" | " "," "(" ")" "[" "]" " " ":" ";" "\""])

(define [c-comma c-lparen c-rparen c-blank c-newline] [`, `( `) `\blank `\n])

(define (all-distinct terms)
  (letrec ((try-all (lambda (t L res)
                      (match L
                        ([] res)
                        ((list-of s more) (match (try (not (= t s)) ())
                                            (() (try-all t more res))
                                            (inequality (try-all t more (add inequality res))))))))))
           (loop (lambda (terms res)
                   (match terms
                     ([] res)
                     ((list-of t more)
                         (loop more (join (try-all t more []) res)))))))
    (loop terms [])))

(define (in x range)
 (match range
   ([l h] (and (<= l x) (<= x h)))))

(define (outside x range)
 (match range
   ([l h] (or (< x l) (< h x)))))

(define (ite x y z)
  (and (or (not x) y)
       (or x z)))

(define (midpoint l h)
  (l plus ((h minus l) div 2)))

(define yices-numeral-prefix "fresh-integer-numeral-")

(define (get-signature' f)
  (let ((rename (lambda (sort)
                  (match sort
                    ("Int" "int")
                    ("Real" "real")
                    ("Boolean" "bool")
                    (_ sort)))))
   (map rename (get-signature f))))

(define (separate L token)
  (match L
    ([] "")
    ([s] s)
```

```
    ((list-of s1 (bind rest (list-of _ _)))
      (join s1 token (separate rest token)))))

(define (separate-all-but-last L token)
  (match L
    ([] "")
    ([_] "")
    ((list-of s1 (bind rest (list-of _ _)))
      (let ((str (separate-all-but-last rest token)))
        (check ((null? str) s1)
               (else (join s1 token str)))))))

(define (integer? n)
  (equal? (sort-of n) "Int"))

(define (real? n)
  (match n
    (x:Real true)
    (_ false)))

(define (proper-real? n)
  (&& (real? n) (negate (integer? n))))

(define (real->rational x)
  (let ((x-str (val->string x))
        (f (lambda (str)
             (match str
               ((split integral (list-of `. decimal))
                  (let ((d (raise 10 (length decimal))))
                    [(string->num (join integral decimal)) d])))))
    (f x-str)))


(define (rational->real n-str d-str)
  (let ((n (string->num (join n-str ".0")))
        (d (string->num d-str)))
   (div n d)))

(define (integer-numeral? n)
  (&& (numeral? n) (integer? n)))

(define fresh-var-prefix "var")

(define (real-numeral? n)
  (&& (numeral? n) (real? n)))

(define (make-fresh prefix counter vmap-range)
  (let ((first-attempt (join prefix (val->string (inc counter)))))
    (check ((for-some vmap-range
                      (lambda (v)
                        (match v
                          ((list-of vname _) (equal? vname first-attempt))
                          (_ false))))
             (make-fresh prefix counter vmap-range))
           (else first-attempt))))

(define built-in-symbols
   (map string->symbol ["<" ">" "<=" ">=" "=" "+" "-" "*" "/"]))

(define (built-in? f)
  (member? f built-in-symbols))

(define (binary-infix? f args)
  (&& (equal? (length args) 2)
      (built-in? f)))

(define (make-generic-term-string f arg-strings)
   (join lparen (symbol->string f) blank (separate arg-strings blank) rparen))

(define (translate-relation-symbol R)
```

```
209   (symbol->string R))
210
211 (define (sc->string sc)
212   (match sc
213     (and "and")
214     (or  "or")))
215
216 (define (make-constraint sc strings)
217    (let ((sc-string (sc->string sc)))
218      (match strings
219        ([s] s)
220        (_ (join lparen sc-string blank strings)))))
221
222 (define (ok-string? name)
223   true)
224
225 (define (translate-sort s)
226   (match s
227     ("Int"  "int")
228     ("Real" "real")
229     ("Boolean" "bool")
230     (_ s)))
231
232
233 (define (translate-constraint c counter vmap bound-var?)
234  (let ((vm (match vmap
235              (() (make-empty-map))
236              (_ vmap)))
237        (debug? false))
238   (letrec ((translate-var
239             (lambda (x)
240               (match (apply-map vm x)
241                 (() (let ((var-name (check ((var? x) (var->string x))
242                                            (else (val->string x))))
243                           (c (first var-name))
244                           (var-name' (check ((ok-string? var-name) var-name)
245                                             (else (make-fresh fresh-var-prefix counter (map-range vm)))))
246                           (var-sort (sort-of x))
247                           (res (match var-sort
248                                  ("Int"  [var-name' "int"])
249                                  ("Real" [var-name' "real"])
250                                  ("Boolean"  [var-name' "bool"])
251                                  (_      [var-name' var-sort])))
252                           (_ (check ((bound-var? x) ())
253                                     (else (add-binding x res vm)))))
254                       var-name'))
255                 ((list-of var-name _) var-name))))
256            (translate-term
257             (lambda (t)
258               (match t
259                (((some-symbol f) (bind args (list-of _ _)))
260                  (let ((arg-strings (translate-terms args))
261                        (res-string (check ((binary-infix? f args)
262                                            (join lparen (symbol->string f) blank (first arg-strings)
    blank
263                                                  (second arg-strings) rparen))
264                                           (else (make-generic-term-string f arg-strings))))
265                        (res-string' (check ((&& (equal? f -) (equal? (length arg-strings) 1))
266                                             (join lparen "- 0 " (first arg-strings) rparen))
267                                            (else res-string)))
268                        (f-string (symbol->string f))
269                        (_ (match (apply-map vm f-string)
270                             (() (check ((binary-infix? f [1 2]) ())
271                                        ((constructor? f) ())
272                                        ((selector? f) ())
273                                        (else (add-binding f-string (add 'function (get-signature' f)) vm))))
274                             (_ ()))))
275                    res-string'))
276                ((some-var x) (translate-var x))
277                (_ (let ((_ (match debug?
```

```
278                          (true (print ("\nAbout to convert the following term t: " t)))
279                          (_ ())))))
280                 (check ((&& (real-numeral? t) (negate (integer? t)))
281                         (let (([n d] (real->rational t))
282                               (str (join (val->string n) "/" (val->string d))))
283                          str))
284                        ((integer-numeral? t) (val->string t))
285                        (else (translate-var t))))))))
286             (translate-terms
287               (lambda (t's)
288                  (letrec ((loop (lambda (terms strings)
289                                   (match terms
290                                     ([] (rev strings))
291                                     ((list-of t rest)
292                                         (let ((t-string (translate-term t)))
293                                           (loop rest (add t-string strings))))))))
294                    (loop t's []))))
295             (translate-atomic-constraint
296               (lambda (c)
297                 (match c
298                    (((some-symbol R) (some-term t1) (some-term t2))
299                       (let ((t1-string (translate-term t1))
300                             (t2-string (translate-term t2))
301                             (R-sign (translate-relation-symbol R))
302                             (_ (check ((built-in? R) ())
303                                       (else (add-binding (symbol->string R) (add 'function (get-signature' R)) vm))))
304                             (res-string (join lparen R-sign blank t1-string blank t2-string rparen)))
305                         res-string))
306                    (((some-symbol R) (some-term t))
307                       (let ((t-string (translate-term t))
308                             (R-sign (translate-relation-symbol R))
309                             (_ (add-binding (symbol->string R) (add 'function (get-signature' R)) vm))
310                             (res-string (join lparen R-sign blank t-string rparen)))
311                         res-string))
312                    ((some-symbol b) (translate-term b))
313                    (((some-symbol R) (some-list terms))
314                       (let ((term-strings (translate-terms terms))
315                             (R-sign (translate-relation-symbol R))
316                             (_ (check ((built-in? R) (add-binding (symbol->string R) (add 'function (get-signature' R))
317                                       (else (add-binding (symbol->string R) (add 'function (get-signature' R)) vm))))
318                             (res-string (make-generic-term-string R term-strings)))
319                         res-string))
320                    ((some-var b) (translate-term b)))))
321             (tran (lambda (c)
322                     (match c
323                       ((some-atom _) (translate-atomic-constraint c))
324                       ((not c') (let ((string (tran c')))
325                                   (join lparen "not " string rparen)))
326                       (((some-sent-con sc) (some-list constraints))
327                          (match c
328                            ((if c1 c2) (let ((strings [(tran c1) (tran c2)]))
329                                          (join lparen "=>" blank (separate strings blank) blank rparen)))
330                            ((iff c1 c2) (tran (and (if c1 c2) (if c2 c1))))
331                            (_ (let ((strings (tran* constraints rparen)))
332                                 (make-constraint sc strings)))))
333                       (((bind Q (|| forall exists)) (some-var x) body)
334                          (let ((body-string (tran body))
335                                ([var-name var-sort] (match (apply-map vm x)
336                                                       (() [(var->string (fresh-var)) "INT"])
337                                                       (res [(first res) (second res)])))
338                                (quant-str (match Q
339                                             (forall "forall ") (_ "exists ")))
340                                (str (join lparen quant-str lparen var-name colon colon var-sort rparen
341                                           blank body-string rparen))
342                                (_ (remove-binding x vm))
343                                (_ (add-binding x [(make-fresh "a_a_a" counter
344                                                               (map-range vm)) var-sort] vm)))
345                            str))
346                       ((some-term _) (translate-term c)))))
347             (tran* (lambda (constraints strings)
```

```
348                     (match constraints
349                       ((list-of c more) (tran* more (join " " (tran c) strings)))
350                       ([] strings)))))
351     (let ((res (tran c)))
352       [res vm])))
353
354 (define (tc c)
355   (let ((counter (cell 0)))
356     (translate-constraint c counter () (lambda (_) false))))
357
358 (define (translate-constraints constraints bound-var?)
359   (let ((counter (cell 0))
360         (vm (make-empty-map)))
361     (letrec ((loop (lambda (constraints vm results)
362                      (match constraints
363                        ([] [(rev results) vm])
364                        ((list-of c more) (let (([res vm'] (translate-constraint c counter vm bound-var?)))
365                                            (loop more vm' (add res results))))))))
366       (loop constraints vm [])))))
367
368 (define (tc-all constraints)
369   (translate-constraints constraints (lambda (_) false)))
370
371 (define (constructor-name? str)
372   (try (constructor? (string->symbol str))
373        false))
374
375 (define (get-constructor-decs var-sort)
376   (let ((constructors (constructors-of var-sort))
377         (get-con-string (lambda (c)
378                           (check ((equal? (arity-of c) 0)
379                                   (symbol->string c))
380                                  (else (let ((arg-sorts (all-but-last (get-signature' c)))
381                                              (sel-names (selector-names c))
382                                              (sel-strings (map (lambda (arg-sort)
383                                                                  (let ((sel-name (join "sel" (var->string (fresh-var)))
384                                                                    (join sel-name colon colon arg-sort)))
385                                                                arg-sorts))
386                                              (sel-strings (map (lambda (arg-sort-and-sel-name)
387                                                                  (match arg-sort-and-sel-name
388                                                                    ([arg-sort []]
389                                                                       (let ((sel-name (join "sel" (var->string (fresh-v
390                                                                         (join sel-name colon colon arg-sort)))
391                                                                    ([arg-sort sel-name] (join sel-name colon colon arg-
392                                                                      (zip arg-sorts sel-names)))
393                                              (sel-string (foldl (lambda (x y) (join x " " y)) [] sel-strings)))
394                                          (join lparen (symbol->string c) sel-string rparen)))))))
395     (map get-con-string constructors)))
396
397 (define (predefined-type? vtype)
398   (|| (equal? vtype "int") (equal? vtype "real") (equal? vtype "bool")))
399
400 (define (make-domain-dec var-type)
401   (check ((datatype-sort? var-type)
402           (let ((constructor-decs (get-constructor-decs var-type)))
403                   (join "\n(define-type " var-type
404                         " (datatype "  (separate constructor-decs blank) "))\n")))
405   (else (join "\n(define-type " var-type ")\n"))))
406
407 (define (make-new-domain-decs remaining-domain-names domain-names-so-far domain-decs)
408   (match remaining-domain-names
409     ([] [domain-names-so-far domain-decs])
410     ((list-of domain-name rest)
411        (check ((member? domain-name domain-names-so-far)
412                  (make-new-domain-decs rest domain-names-so-far domain-decs))
413               ((predefined-type? domain-name)  [domain-names-so-far domain-decs])
414               (else (let ((new-domain-dec (make-domain-dec domain-name)))
415                       (make-new-domain-decs rest (add domain-name domain-names-so-far)
416                                             (join new-domain-dec domain-decs))))))))
417
```

```
418
419 (define (defined-symbol? str symbol-definitions)
420   (member? str (map (lambda (x) (symbol->string (root (first x)))) symbol-definitions)))
421
422 (define (get-decs-from-vmap dom-range symbol-definitions)
423   (letrec ((loop (lambda (dom-range domains-so-far domain-decs var-decs reverse-vmap)
424                    (match dom-range
425                      ([] [domain-decs var-decs reverse-vmap])
426                      ((list-of [var (list-of 'function rest)] more)
427                        (check
428                         ((defined-symbol? var symbol-definitions)
429                             (loop more domains-so-far domain-decs var-decs reverse-vmap))
430                         (else
431                           (let ((new-vdec (join newline lparen "define" blank var colon colon lparen "-> " (separate
432                                 ([domains-so-far' domain-decs'] (make-new-domain-decs rest domains-so-far domain-decs
433                                 # (_ (mprint (join "\nfunction declaration encountered, with rest signature: " (val->
434                                 # (_ (mprint (join "\nDomains-so-far: " (val->string domains-so-far) "\nand new domai
435                                 #                  "\nand domain-decs: " (val->string domain-decs)
436                                 #                  "\nand new domain-decs': " (val->string domain-decs'))))
437                                 (loop more domains-so-far' domain-decs' (join new-vdec var-decs) reverse-vmap)))))
438                      ((list-of [var var-value] more)
439                        (let ((var-name (first var-value))
440                              (var-type (second var-value))
441                              (new-vdec (join newline lparen "define " var-name colon colon var-type rparen))
442                              ([reverse-vmap' var-decs'] (check ((constructor-name? var-name) [reverse-vmap var-decs]
443                                                                 (else [(add-binding var-name var reverse-vmap)
444                                                                        (join new-vdec var-decs)])))))
445                          (check ((predefined-type? var-type)
446                                    (loop more domains-so-far domain-decs var-decs' reverse-vmap'))
447                                 ((member? var-type domains-so-far)
448                                    (loop more domains-so-far domain-decs var-decs' reverse-vmap'))
449                                 (else (let ((new-domain-dec (make-domain-dec var-type)))
450                                         (loop more (add var-type domains-so-far)
451                                                    (join new-domain-dec domain-decs)
452                                                    var-decs' reverse-vmap')))))))))))
453     (loop dom-range  [] [] [] (make-empty-map))))
454
455 (define (get-declarations c)
456   (let (([char-vec vmap] (tc c))
457         (dom-range (dom-range-list vmap)))
458     (join [char-vec vmap] (get-decs-from-vmap dom-range []))))
459
460 (define (get-declarations* constraint-list)
461   (let (([constraint-strings vmap] (tc-all constraint-list))
462         (dom-range (dom-range-list vmap)))
463     (letrec ((loop (lambda (dom-range domains-so-far domain-decs var-decs reverse-vmap)
464                      (match dom-range
465                        ([] [domain-decs var-decs reverse-vmap])
466                        ((list-of [var (list-of 'function rest)] more)
467                            (let ((new-vdec (join newline lparen "define" blank var colon colon lparen "-> " (separate
468                                  ([domains-so-far' domain-decs'] (make-new-domain-decs rest domains-so-far domain-decs
469                                  (loop more domains-so-far' domain-decs' (join new-vdec var-decs) reverse-vmap)))
470                        ((list-of [var var-value] more)
471                          (let ((var-name (first var-value))
472                                (var-type (second var-value))
473                                (new-vdec (join newline lparen "define " var-name colon colon var-type rparen))
474                                ([reverse-vmap' var-decs'] (check ((constructor-name? var-name) [reverse-vmap var-decs]
475                                                                   (else [(add-binding var-name var reverse-vmap)
476                                                                          (join new-vdec var-decs)])))))
477                            (check ((predefined-type? var-type)
478                                      (loop more domains-so-far domain-decs var-decs' reverse-vmap'))
479                                   ((member? var-type domains-so-far)
480                                      (loop more domains-so-far domain-decs var-decs' reverse-vmap'))
481                                   (else (let ((new-domain-dec (make-domain-dec var-type)))
482                                           (loop more (add var-type domains-so-far)
483                                                      (join new-domain-dec domain-decs)
484                                                      var-decs' reverse-vmap')))))))))))
485       (join [constraint-strings vmap] (loop dom-range [] [] [] (make-empty-map))))))
486
487
```

```
(define (gd c symbol-definitions)
  (match symbol-definitions
    ([] (get-declarations c))
    (_ (let ((vmap (make-empty-map))
             (counter (cell 0))
             (def-strings (map (lambda (sd-pair)
                                 (let ((sym  (root (first sd-pair)))
                                       (sym-name (symbol->string sym))
                                       (sig (separate (get-signature' sym) " "))
                                       (sym-name-and-sig (join sym-name colon colon lp "-> " sig rp))
                                       (params (children (first sd-pair)))
                                       (bound-var? (lambda (x) (member? x params)))
                                       (parameter-list (separate (map (lambda (x)
                                                                        (let ((var-name (first (translate-constraint
                                                                              (sort (translate-sort (sort-of x))))
                                                                          (join var-name colon colon sort)))
                                                                      params) " "))
                                       (body (first (translate-constraint (second sd-pair) counter vmap bound-var?)))
                                   (join "\n(define " sym-name-and-sig blank
                                    "(lambda " lp parameter-list rp blank body "))\n")))
                               symbol-definitions))
             ([c-res _] (translate-constraint c counter vmap (lambda (_) false)))
             (dom-range (dom-range-list vmap)))
        (join [def-strings c-res vmap] (get-decs-from-vmap dom-range symbol-definitions))))))

(define (get-declarations-faster constraints)
 (let
  ((v-ht (make-hash-table))
   (d-ht (make-hash-table))
   (get-all-v-decs (lambda ()
                     (map first (show-table v-ht))))
   (get-all-d-decs (lambda ()
                     (map first (show-table d-ht))))
   (process-constraint
    (lambda (c rvmap)
      (let (([str vmap] (tc c))
            (dom-range (dom-range-list vmap)))
        (letrec ((loop (lambda (dom-range domains-so-far domain-decs var-decs reverse-vmap)
                         (match dom-range
                           ([] [domain-decs var-decs reverse-vmap])
                           ((list-of [var (list-of 'function rest)] more)
                               (let ((new-vdec (join newline lparen "define" blank var colon colon lparen "-> " (separ
                                  (loop more domains-so-far domain-decs (add new-vdec var-decs) reverse-vmap)))
                           ((list-of [var var-value] more)
                             (let ((var-name (first var-value))
                                   (var-type (second var-value))
                                   (new-vdec (join newline lparen "define " var-name colon colon var-type rparen))
                                   ([reverse-vmap' var-decs'] (check ((constructor-name? var-name) [reverse-vmap var-d
                                                                     (else [(add-binding var-name var reverse-vmap)
                                                                            (add new-vdec var-decs)])))))
                               (check ((|| (equal? var-type "int") (equal? var-type "real") (equal? var-type "bool"))
                                       (loop more domains-so-far domain-decs var-decs' reverse-vmap'))
                                  ((member? var-type domains-so-far)
                                    (loop more domains-so-far domain-decs var-decs' reverse-vmap'))
                                   (else (let ((new-domain-dec (check ((datatype-sort? var-type)
                                                                       (let ((constructor-decs (check ((poly? var
                                                                                                       (else (get-
                                                                         (join "\n(define-type " var-type
                                                                               " (datatype "
(separate constructor-decs blank) "))\n")))
                                                                     (else (join "\n(define-type " var-type ")\n")
                                                 (loop more (add var-type domains-so-far)
                                                            (add new-domain-dec domain-decs)
                                                            var-decs' reverse-vmap')))))))))))
                 (join [str vmap] (loop dom-range [] [] [] rvmap)))))))
        (letrec ((do-all (lambda (c-list all-strings rev-vmap)
                           (match c-list
                             ([] [(rev all-strings) (get-all-v-decs) (get-all-d-decs) rev-vmap])
                             ((list-of c more) (let (([str vmap d-decs v-decs reverse-vmap] (process-constraint c rev-vma
                                                    (_ (map (lambda (v-dec)
```

```
                                                                  (enter v-ht v-dec true))
                                                                v-decs))
                                                        (_ (map (lambda (d-dec)
                                                                  (enter d-ht d-dec true))
                                                                d-decs)))
                                                    (do-all more (add str all-strings)
                                                                reverse-vmap)))))))
        (do-all constraints [] (make-empty-map)))))


(define (get-declarations' c)
  (let (([str _] (tc c)))
    str))

(define (get-line str)
  (letrec ((loop (lambda (str chars)
                   (match str
                     ([] [(rev chars) []])
                     ((list-of '\n rest) [(rev (add '\n chars)) rest])
                     ((list-of c rest)  (loop rest (add c chars)))))))
    (loop str [])))


(define lparen-char '\040)
(define rparen-char '\041)

(define (balanced? str)
  (let ((lparens (filter str (lambda (c) (equal? c lparen-char))))
        (rparens (filter str (lambda (c) (equal? c rparen-char)))))
    (equal? (length lparens) (length rparens))))

(define (get-line str)
  (letrec ((loop (lambda (str chars)
                   (match str
                     ([] [(rev chars) []])
                     ((list-of '\n rest) (let ((res (rev (add '\n chars))))
                                           (check ((balanced? res) [res rest])
                                                  (else (loop rest chars)))))
                     ((list-of c rest)  (loop rest (add c chars)))))))
    (loop str [])))

(define (get-val str)
  (try (string->num str)
       (match str
         ((split n-str (split "/" d-str)) (rational->real n-str d-str))
         (_ (string->symbol str)))))

(define (skip? left right)
   (let ((skipable (lambda (str)
                     (match str
                       ((split "LET" _) true)
                       ((split "(LAMBDA" _) true)
                       (_ false)))))
     (|| (skipable left) (skipable right))))

(define (parseTerm str reverse-map yices-integer-numerals)
  (letrec ((get-functor (lambda (str res)
                          (match str
                            ((list-of (val-of c-blank) rest) [(rev res) rest])
                            ((list-of (val-of c-newline) rest) [(rev res) rest])
                            ((list-of (val-of c-rparen) rest) [(rev res) str])
                            ((list-of (some-char c) rest) (get-functor rest (add c res)))
                            ([] [(rev res) []])))))
           (get-term (lambda (str)
                       (match str
                         ((list-of (val-of c-lparen) rest)
                            (let (([functor rest] (get-functor rest [])))
                              (match (get-terms rest [])
                                ([args (list-of c-rparen rest')]
                                  (let ((fsym (string->symbol functor))
```

```
627                                        (term (try (make-term fsym args)
628                                                   (let ((arg-sorts (all-but-last (get-signature fsym)))
629                                                         (args' (map
630                                                                 (lambda (arg-and-sort)
631                                                                    (match arg-and-sort
632                                                                      ([arg expected-sort] (check ((&& (integer-numera
633                                                                                                       (negate (equal
634                                                                                                    (let ((ynt (stri
635
636
637                                                                                                          (_ (set! y
638                                                                                                                   (
639                                                                                                       ynt))
640                                                                                                 (else arg)))))
641                                                                 (zip args arg-sorts))))
642                                                    (make-term fsym args')))))
643                                    [term rest']))
644                                  ([args []] (let ((_ (mprint (join "\nfunctor: " functor))))
645                                                  [(make-term (string->symbol functor) args) []])))))
646                           (_ (let (([functor rest] (get-functor str [])))
647                                (match (apply-map reverse-map functor)
648                                       (() (try [(get-val functor) rest]
649                                                [1 rest]))
650                                       (x [x rest])))))))
651              (get-terms (lambda (str results)
652                           (match str
653                             ([] [(rev results) []])
654                             (_ (match (get-term str)
655                                  ([term (list-of (val-of c-blank) rest)] (get-terms rest (add term results)))
656                                  ([term (bind all-rest (list-of (val-of c-rparen) rest))] [(rev (add term results)) all-
657                                  ([term (bind all-rest (list-of c rest))] (get-terms all-rest (add term results)))
658                                  ([term []] [(rev (add term results)) []])))))))
659        (get-term str)))
660
661  (define (parse-term str reverse-map)
662    (first (parseTerm str reverse-map)))
663
664
665
666  (define (fresh-variable? t)
667    (match t
668      ((some-var _) (prefix? "fresh-" (var->string t)))
669      (_ false)))
670
671   (define (simplify identity model)
672     (match identity
673       ((= l r) (check ((&& (fresh-variable? r) (input-leaf? l))
674                        (map (lambda (p)
675                               (replace-var r l p))
676                             model))
677                       (else model)))
678       (_ model)))
679
680
681
682  (define (process-yices-output reverse-vmap file conjuncts simplify?)
683     (let ((data (read-file file))
684           ([line1 rest1] (get-line data))
685           (yices-integer-numerals (cell []))
686           (empty-table (make-empty-map))
687           (input-leaf? (lambda (x)
688                          (for-some conjuncts
689                             (lambda (c)
690                                (member? x (leaves c))))))
691           (simplify (lambda (identity model)
692                       (match identity
693                          ((= l r) (check ((&& (fresh-variable? r) (input-leaf? l))
694                                           [true (map (lambda (p)
695                                                        (replace-var r l p))
696                                                      model)])
```

```
                                        (else [false model])))
                          (_ [false model])))))
        (get-unsat-assertions (lambda (num-string)
                                (letrec ((loop (lambda (str res)
                                                 (match (skip-until str printable?)
                                                   ([] res)
                                                   (_ (match (parseTerm str empty-table yices-integer-numerals)
                                                        ([n rest] (loop rest (add n res)))))))))
                                  (loop num-string [])))))
    (letrec ((get-model (lambda (str L)
                          (let (([line rest-lines] (get-line (skip-until str printable?))))
                            (match line
                              ((split "(= " rest)
                                (let (([left rest'])   (parseTerm rest reverse-vmap yices-integer-numerals))
                                      ([right (list-of (val-of c-rparen) rest'')] (parseTerm (skip-until rest' printal
                                 (check ((skip? left right) (get-model rest-lines L))
                                        (else (let ((left-term left)
                                                    (right-term right)
                                                    (identity  (try (= left-term right-term)
                                                                    (check ((integer-numeral? right-term)
                                                                            (let ((var-sort (sort-of left-term))
                                                                                  (ynt (string->var (join "fresh-" (
                                                                                                           (val->stri
                                                                                                        var-sort))
                                                                                  (_ (set! yices-integer-numerals
                                                                                           (add ynt (ref yices-integ
                                                                                    (= left-term ynt)))))))
                                                    (check ((equal? left-term right-term) (get-model rest-lines L))
                                                           ((&& (ground? identity) (member? identity conjuncts)) (get-mod
                                                           (else (get-model rest-lines (add identity L)))))))))
                              ([] L)
                              (_ (get-model rest-lines L))))))
             (simplify-model (lambda (L)
                               (letrec ((loop (lambda (remaining-equations latest-model)
                                                (match remaining-equations
                                                  ([] latest-model)
                                                  ((list-of eqn more) (let (([change? model] (simplify eqn latest-
                                                                                (check (change? (loop more model))
                                                                                       (else (loop more latest-model))))))
                                                                        (remove-redundancies (lambda (L)
                                                                                               (filter-out L (lambda (id)
                                                                                                               (|| (&& (member? id conjuncts)
                                                                                                                       (negate (var? (lhs id))))
                                                                                                                   (equal? (lhs id) (rhs id))))))
                                                                        (check (simplify? (remove-redundancies (loop L L)))
                                                                               (else (loop L L)))))))))
                                 (let ((get-model' (lambda (str L)
                                                     (match (get-model str L)
                                                       ((bind answer [['satisfying-assignment res] ['unsatisfied-assertions unsat-assertions]]
                                                          (match (remove-duplicates (ref yices-integer-numerals))
                                                            ([] answer)
                                                            (ynterms [['satisfying-assignment (join (simplify-model res) (all-distinct ynterm
                                                       ((some-list bindings)
                                                          (match (remove-duplicates (ref yices-integer-numerals))
                                                            ([] (simplify-model bindings))
                                                            (ynterms (join (simplify-model bindings) (all-distinct ynterms)))))))))
                                   (match line1
                                     ((split "unsat" _) 'Unsatisfiable)
                                     ((split "Error" _) 'Unknown-error)
                                     ((split "unknown" _) (try (let ((L (get-model' rest1 [])))
                                                                 ['Unknown L])
                                                               'Unknown))
                                     ((split "sat" _) (let (([line2 rest2] (get-line rest1))
#                                                            (_ (print "\nline2: " line2))
#                                                            (_ (print "\nrest2: " rest2))
                                                           )
                                                        (match line2
                                                          ((split "unsatisfied assertion ids: " more)
                                                            (let ((unsat-assertions (get-unsat-assertions more))
                                                                  (res (get-model' rest2 [])))
```

```
                                        [['satisfying-assignment res] ['unsatisfied-assertions unsat-assertions]]))
                                  (_ (get-model' rest1 [])))))))
             (_ 'Unknown))))))

(define (smt-solve-core c simplify?)
  (let ((([input-file output-file error-file] ["input1.ys" "output1.ys" "error.ys"])
         (_ (delete-files [input-file output-file error-file]))
         ([char-vec vmap d-decs v-decs reverse-vmap] (get-declarations c))
#         (RL (dom-range-list reverse-vmap))
#         (rv (second (first RL)))
#         (_ (print "\nReverse vmap: " RL))
#         (_ (print "\nAnd here is the variable corresponding to a: " rv " and its sort: " (sort-of rv)))
         (_ (mprint "\nDone with translation...\n"))
         (_ (write-file input-file ";; Type declarations:\n"))
         (_ (write-file input-file d-decs))
         (_ (write-file input-file "\n;; Variable declarations:\n\n"))
         (_ (write-file input-file v-decs))
         (_ (write-file input-file "\n\n;; Query: \n\n"))
         (_ (write-file input-file "(assert "))
         (_ (write-file input-file char-vec))
         (_ (write-file input-file ")\n"))
         (_ (write-file input-file (join "\n(check)\n")))
         (_ (mprint "\nSending OS command...\n"))
         (time1 (time))
         (_ (exec-command (join "yices --timeout=" (ref yices-time-limit) " -e " input-file " > " output-file " 2> " er
         (time2 (time))
         (_ (mprint (join "\nDone. Total solving time: " (val->string (minus time2 time1)))))
         (res (process-yices-output reverse-vmap output-file (get-conjuncts c) simplify?)))
    res))


(define (smt-solve-core-with-defs c symbol-definitions simplify?)
 (match symbol-definitions
   ([] (smt-solve-core c simplify?))
   (_
     (let ((([input-file output-file error-file] ["input1.ys" "output1.ys" "error.ys"])
            (_ (delete-files [input-file output-file error-file]))
            ([defs char-vec vmap d-decs v-decs reverse-vmap] (gd c symbol-definitions))
            (def-str (many-strings->one-string defs))
            (_ (mprint "\nDone with translation...\n"))
            (_ (write-file input-file ";; Type declarations:\n"))
            (_ (write-file input-file d-decs))
            (_ (write-file input-file "\n;; Variable declarations:\n\n"))
            (_ (write-file input-file v-decs))
            (_ (write-file input-file "\n\n;; Definitions:\n"))
            (_ (write-file input-file def-str))
            (_ (write-file input-file "\n\n;; Query: \n\n"))
            (_ (write-file input-file "(assert "))
            (_ (write-file input-file char-vec))
            (_ (write-file input-file ")\n"))
            (_ (write-file input-file (join "\n(check)\n")))
            (_ (mprint "\nSending OS command...\n"))
            (time1 (time))
            (_ (exec-command (join "yices --timeout=" (ref yices-time-limit) " -e " input-file " > " output-file " 2> "
            (time2 (time))
            (_ (mprint (join "\nDone. Total solving time: " (val->string (minus time2 time1)))))
            (res (process-yices-output reverse-vmap output-file (get-conjuncts c) simplify?)))
       res))))

(define (translate-and-write-to-file c file file-to-include)
  (let ((([str vmap d-decs v-decs reverse-vmap] (get-declarations c))
         (_ (write-file file ";; Type declarations:\n"))
         (_ (write-file file d-decs))
         (_ (write-file file "\n;; Variable declarations:\n\n"))
         (_ (write-file file v-decs))
         (_ (write-file file "\n\n;; Query: \n\n"))
         (_ (write-file file (join "(include " quot-mark file-to-include quot-mark rparen newline)))
         (_ (write-file file "(assert "))
         (_ (write-file file str))
         (_ (write-file file ")\n"))
```

```
837         (_ (write-file file (join "\n(check)\n"))))
838     reverse-vmap))
839
840 (define (translate-and-write-to-file-simple c file)
841   (let (([str _] (tc c))
842         (_ (write-file file "\n(assert "))
843         (_ (write-file file str))
844         (_ (write-file file ")\n")))
845     ()))
846
847 (define (smt-solve-core-repeat c n simplify?)
848   (let (([input-file output-file error-file] ["input1.ys" "output1.ys" "error.ys"])
849         (_ (delete-files [input-file output-file error-file]))
850         ([str vmap d-decs v-decs reverse-vmap] (get-declarations c))
851         (_ (mprint "\nDone with translation...\n"))
852         (_ (write-file input-file ";; Type declarations:\n"))
853         (_ (write-file input-file (separate d-decs "\n")))
854         (_ (write-file input-file "\n;; Variable declarations:\n\n"))
855         (_ (write-file input-file (separate v-decs "\n")))
856         (_ (write-file input-file "\n\n;; Query: \n\n"))
857         (_ (write-file input-file (join "(assert " str ")\n")))
858         (_ (write-file input-file (join "\n(check)\n")))
859         (_ (mprint "\nSending OS command...\n"))
860         (t (running-time (lambda () (exec-command (join "yices --timeout=" (ref yices-time-limit)
861                                                         " -e " input-file " > " output-file " 2> " error-file))) n))
862         (_ (mprint "\nDONE! Performed " n " calls in " t " seconds...\n"))
863         (res (process-yices-output reverse-vmap output-file (get-conjuncts c) simplify?)))
864     res))
865
866 # Interface for solving MaxSat problems w/ Yices:
867
868 (define (max-sat-smt-solve-core L simplify?)
869   (let ((assertions (map first L))
870         (weights (map second L))
871         ([input-file output-file error-file] ["minput1.ys" "moutput1.ys" "merror.ys"])
872         (_ (delete-files [input-file output-file error-file]))
873         ([assertion-strings v-decs d-decs reverse-vmap] (get-declarations-faster assertions))
874         (assertion-strings' (letrec ((loop (lambda (data weights res)
875                                              (match [data weights]
876                                                ([[] []] (rev res))
877                                                ([(list-of str rest-data) (list-of weight rest-weights)]
878                                                   (let ((new-assertion-string
879                                                           (match weight
880                                                             ('inf (join "\n(assert+ " str rparen newline))
881                                                             (_ (join "\n(assert+ " str blank (val->string weight) rpa
882                                                     (loop rest-data rest-weights (add new-assertion-string res))))))))))
883                                (loop assertion-strings weights []))))
884         (_ (mprint "\nDone with translation...\n"))
885         (_ (write-file input-file ";; Type declarations:\n"))
886         (_ (write-file input-file (separate d-decs "\n")))
887         (_ (write-file input-file "\n;; Variable declarations:\n\n"))
888         (_ (write-file input-file (separate v-decs "\n")))
889         (_ (write-file input-file "\n\n;; Assertions: \n"))
890         (_ (map (lambda (assertion-string)
891                   (write-file input-file assertion-string))
892                 assertion-strings'))
893         (_ (write-file input-file (join "\n(max-sat)\n")))
894         (_ (mprint "\nSending OS command...\n"))
895         (time1 (time))
896         (_ (exec-command (join "yices --timeout=" (ref yices-time-limit) " -e " input-file " > " output-file " 2> " er
897         (time2 (time))
898         (_ (mprint (join "\nDone. Total solving time: " (val->string (minus time2 time1)))))
899         (res (process-yices-output reverse-vmap output-file assertions simplify?)))
900     res))
901
902 (define (solve-smt-constraint c simplify?)
903   (check ((poly? c) (error "Polymorphic constraints are not supported presently."))
904          (else (smt-solve-core (rename c) simplify?))))
905
906 (define (solve-smt-constraint-with-defs c symbol-definitions simplify?)
```

```
907   (check ((poly? c) (error "Polymorphic constraints are not supported presently."))
908          (else (smt-solve-core-with-defs (rename c) symbol-definitions simplify?))))
909
910 (define (smt-solve c)
911   (check ((ref show-whole-model?) (solve-smt-constraint c false))
912          (else (solve-smt-constraint c true))))
913
914 (define (smt-solve-with-defs c symbol-defs)
915   (check ((ref show-whole-model?) (solve-smt-constraint-with-defs c symbol-defs false))
916          (else (solve-smt-constraint-with-defs c symbol-defs true))))
917
918 (define (smt-solve-list constraints)
919   (let (([input-file output-file error-file] ["input1.ys" "output1.ys" "error.ys"])
920         (_ (delete-files [input-file output-file error-file]))
921         ([constraint-strings vmap d-decs v-decs reverse-vmap] (get-declarations* constraints))
922 #        (_ (mprint "\nDone with translation...\n"))
923         (_ (write-file input-file ";; Type declarations:\n"))
924         (_ (write-file input-file d-decs))
925         (_ (write-file input-file "\n;; Variable declarations:\n\n"))
926         (_ (write-file input-file v-decs))
927 #        (_ (mprint "\nDone writing the type and variable declarations, about to write the main assertions...\n"))
928 #         (big-string (join "\n(assert (and "
929 #                           (separate constraint-strings " ")
930 #                           "))\n"))
931 #        (_ (mprint "\nDone computing big string...\n"))
932         (_ (write-file input-file "\n\n;; Query: \n\n"))
933          (_ (map-proc (lambda (constraint-string)
934                         (seq (write-file input-file "(assert ")
935                              (write-file input-file constraint-string)
936                              (write-file input-file ")\n")))
937                       constraint-strings))
938         (_ (write-file input-file (join "\n(check)\n")))
939         (_ (mprint "\nSending OS command...\n"))
940         (time1 (time))
941         (_ (exec-command (join "yices --timeout=" (ref yices-time-limit) " -e " input-file " > " output-file " 2> " er
942         (time2 (time))
943         (_ (mprint (join "\nDone. Total solving time: " (val->string (minus time2 time1)))))
944         (res (check ((ref show-whole-model?)  (process-yices-output reverse-vmap output-file (flatten (map get-conjunc
945                     (else (process-yices-output reverse-vmap output-file (flatten (map get-conjuncts constraints)) tru
946   res))
947
948 (define (max-smt-solve L)
949   (check ((ref show-whole-model?)   (max-sat-smt-solve-core L false))
950          (else (max-sat-smt-solve-core L true))))
951
952 (define (solve p)
953   (smt-solve (and* (add p (ab)))))
954
955 (set-precedence (smt-solve solve max-smt-solve) 5)
956
957 # Code for obtaining multiple models of a given constraint c:
958
959 (define (smt-multiple-models c max)
960   (let ((negate-model (lambda (model)
961                         (and* (map not model)))))
962     (letrec ((loop (lambda (c i models)
963                      (check ((less? i max)
964                              (match (smt-solve c)
965                                ((some-list model) (loop (and c (negate-model model))
966                                                         (plus i 1)
967                                                         (add model models)))
968                                (_ models)))
969                             (else models)))))
970       (loop c 0 []))))
971
972 (define (smt-satisfiable? c)
973   (try (match (smt-solve c)
974          ('Unknown 'Unknown)
975          ('Unsatisfiable false)
976          (_ true))
```

```
977         'Unknown))
978
979   (define (smt-valid? c)
980     (try (match (smt-solve (not c))
981            ('Unknown 'Unknown)
982            ('Unsatisfiable true)
983            (_ false))
984          'Unknown))
985
986   (define (smt-unsatisfiable? c)
987     (negate (smt-satisfiable? c)))
988
989   (define (smt-implies? c1 c2)
990     (smt-unsatisfiable? (and c1 (not c2))))
991
992   (define (smt-check c expected)
993     (let ((my-assert
994            (lambda (result)
995              (check ((equal? result expected)
996                      (println (join "As expected: " (val->string expected))))
997                     (else
998                      (println
999                       (join "Error: Expected " (val->string expected)
1000                             ", but returned " (val->string result))))))))
1001      (try (let ((result (smt-solve c)))
1002             (match result
1003               ((|| 'Unknown 'Unsatisfiable) (my-assert result))
1004               (_ (my-assert 'Satisfiable))))
1005           (println (join "SMT unit test '" (val->string c) "' timed out.")))))
1006
1007   (define (size p)
1008     (match p
1009       ((some-atom t) (term-size t))
1010       ((((|| not and or if iff) (some-list args))
1011          (foldl plus 1 (map size args)))
1012       (((some-quant _) (some-var _) (some-sent q))
1013          (plus 2 (size q)))))
1014
1015
1016   (define (apply-solution L s)
1017     (match (find-first' L
1018              (lambda (id)
1019                (match id
1020                  ((= (val-of s) (some-symbol t)) t)
1021                  ((= (some-symbol t) (val-of s)) t)
1022                  (_ false)))
1023              (lambda () ()))
1024      (() 'none)
1025      (res res)))
1026
1027   (define (get-cost solution cost-terms)
1028     (let ((costs (map (lambda (cost-term) (apply-solution solution cost-term)) cost-terms)))
1029       (eval (foldl + 0 costs))))
1030
1031   (define (make-cost-term t)
1032     (match t
1033       ((some-var x) (string->var (join "cost" (var->string x) ":Int")))
1034       (((some-symbol f) (some-list _)) (string->var (join "cost" (symbol->string f) ":Int")))))
1035
1036   (define (parameter-value-and-cost p desired p-change-cost)
1037     (let ((p-cost-term (make-cost-term p)))
1038       (ite (= p desired) (= p-cost-term 0) (= p-cost-term p-change-cost))))
1039
1040   (define (sum-all terms)
1041     (match terms
1042       ([x] x)
1043       ((list-of x (bind rest (list-of _ _))) (+ x (sum-all rest)))))
1044
1045   (define (sum n)
1046     (check ((less? n 1) 0)
```

```
1047          (else (plus n (sum (minus n 1))))))
1048
1049  (define (cost-term-leaves cost-term)
1050    (match cost-term
1051      (((some-symbol f) (some-list args))
1052          (check ((member? f built-in-symbols) (flatten (map cost-term-leaves args)))
1053                 (else [cost-term])))
1054      (_ [cost-term])))
1055
1056  (define (smt-solve-and-minimize constraint cost-term max-cost)
1057   (let ((counter (cell 0))
1058         (cost-terms (cost-term-leaves cost-term))
1059         (main-input-file "mininput.ys")
1060         ([output-file error-file] ["minoutput.ys" "minerror.ys"])
1061         (cost-constraint-file "cc.ys")
1062         (_ (delete-files [main-input-file output-file error-file cost-constraint-file]))
1063         (reverse-vmap (translate-and-write-to-file constraint main-input-file cost-constraint-file))
1064         (main-conjuncts (get-conjuncts constraint))
1065         (solve (lambda (cost-constraint)
1066                  (seq (inc counter)
1067                       (delete-files [cost-constraint-file output-file error-file])
1068                       (translate-and-write-to-file-simple cost-constraint cost-constraint-file)
1069                       (mprint "\nSolver kicking in...\n")
1070                       (let ((t1 (time))
1071                             (_ (exec-command (join "yices --timeout=" (ref yices-time-limit) " -e " main-input-file " 
1072                             (t2 (time)))
1073                         (mprint (join "\nSolver done, total solving time " (val->string (minus t2 t1)) " seconds; abo
1074                       (let ((res (process-yices-output reverse-vmap output-file (join (get-conjuncts cost-constraint) 
1075                             (_ (mprint "\nOutput processing done...\n")))
1076                         res)))))
1077    (letrec ((loop (lambda (l h)
1078                     (let ((_ (mprint (join "\nlo : " (val->string l) ", hi: " (val->string h) ", and mid: " (val->string
1079                      (check ((less? h l) (seq (mprint "\nNothing found...\n") 'Unsatisfiable))
1080                             ((equal? h l) (solve (= cost-term h)))
1081                             (else (let ((midpoint (midpoint l h))
1082                                         (cost-constraint (and (>= cost-term l) (<= cost-term midpoint))))
1083                                     (match (solve cost-constraint)
1084                                       ((some-list L) (check ((less? l midpoint)
1085                                                              (let ((total-cost (get-cost L cost-terms))
1086                                                                    (_ (mprint (join "\n   Total cost: " (val->string t
1087                                                                (loop l total-cost)))
1088                                                            (else L)))
1089                                       ('Unknown 'Unknown)
1090                                       (_ (loop (plus midpoint 1) h)))))))))))
1091       (let ((res (loop 0 max-cost))
1092             (_ (mprint (join "\n\nTotal calls: " (val->string (ref counter)) "\n\n"))))
1093         res))))
1094
1095  (define (smt-solve-and-minimize-afresh constraint cost-term max-cost)
1096   (let ((counter (cell 0))
1097         (cost-terms (cost-term-leaves cost-term))
1098         (solve (lambda (c) (seq (inc counter)
1099                                 (smt-solve c)))))
1100    (letrec ((loop (lambda (l h)
1101                     (let ((_ (mprint (join "\nlo : " (val->string l) ", hi: " (val->string h) ", and mid: " (val->string
1102                      (check ((less? h l) (seq (mprint "\nNothing found...\n") 'Unsatisfiable))
1103                             ((equal? h l) (solve (and constraint (= cost-term h))))
1104                             (else (let ((midpoint (midpoint l h))
1105                                         (_ (print "\nmidpoint: " midpoint))
1106                                         (cost-constraint (and (>= cost-term l) (<= cost-term midpoint))))
1107                                     (match (solve (and constraint cost-constraint))
1108                                       ((some-list L) (check ((less? l midpoint)
1109                                                              (let ((total-cost (get-cost L cost-terms))
1110                                                                    (_ (mprint (join "\n   Total cost: " (val->string t
1111                                                                (loop l total-cost)))
1112                                                            (else L)))
1113                                       (_ (loop (plus midpoint 1) h)))))))))))
1114       (let ((_ ())
1115             (res (loop 0 max-cost))
1116             (_ (mprint (join "\n\nTotal calls: " (val->string (ref counter)) "\n\n"))))
```

```
        res))))

#;;;;;;;;; Some useful procedures for testing the above code:

(define (apply-solution L s)
  (match (find-first' L
           (lambda (id)
             (match id
               ((= (val-of s) t) t)
               (_ false)))
           (lambda () ()))
    (() 'none)
    (res res)))

(define (make-constraint n)
  (let ((span (from-to 1 n))
        (vars (map (lambda (_) (fresh-var "Int")) span))
        (counter (cell 1))
        (cost (cell 1))
        (range-sentences-and-var-values-1
            (map (lambda (v)
                   (let ((low ((inc counter) times 10))
                         (hi  (plus low 5)))
                     [(in v [low hi])
                      (= v (plus low 1))]))
                 vars))
        (range-sentences-and-var-values-2
           (map (lambda (v)
                  (let ((low ((inc counter) times 100))
                        (hi  (plus low 10)))
                    [(in v [low hi])
                     (= v (plus low 2))]))
                vars))
        ([range-sentences-1 var-values-1] (unzip range-sentences-and-var-values-1))
        ([range-sentences-2 var-values-2] (unzip range-sentences-and-var-values-2))
        (constraint (or (and* range-sentences-1) (and* range-sentences-2)))
        (mid (midpoint 1 n))
        (values-1 (take var-values-1 mid))
        (values-2 (second (split-list var-values-2 mid)))
        (values (join values-1 values-2))
        (cost-constraints (map (lambda (var-val)
                                 (match var-val
                                   ((= v val) (let ((v-cost-term (make-cost-term v)))
                                                (ite (= v val) (= v-cost-term 0) (= v-cost-term (inc cost)))))))
                               values))
       (cost-variables (map make-cost-term vars))
       (cost-term (sum-all cost-variables))
       (cost-constraint (and* cost-constraints))
       (max-cost (sum (length vars))))
    [constraint vars cost-constraint cost-term max-cost]))

(define (make-max-constraint n)
  (let ((span (from-to 1 n))
        (vars (map (lambda (_) (fresh-var "Int")) span))
        (counter (cell 1))
        (cost (cell 1))
        (range-sentences-and-var-values-1
            (map (lambda (v)
                   (let ((low ((inc counter) times 10))
                         (hi  (plus low 5)))
                     [(in v [low hi])
                      (= v (plus low 1))]))
                 vars))
        (range-sentences-and-var-values-2
           (map (lambda (v)
                  (let ((low ((inc counter) times 100))
                        (hi  (plus low 10)))
                    [(in v [low hi])
                     (= v (plus low 2))]))
                vars))
```

```
1187          ([range-sentences-1 var-values-1] (unzip range-sentences-and-var-values-1))
1188          ([range-sentences-2 var-values-2] (unzip range-sentences-and-var-values-2))
1189          (main-constraint [(or (and* range-sentences-1) (and* range-sentences-2)) 'inf])
1190          (mid (midpoint 1 n))
1191          (values-1 (take var-values-1 mid))
1192          (values-2 (second (split-list var-values-2 mid)))
1193          (values (join values-1 values-2))
1194          (cost-constraints (map (lambda (var-val)
1195                                   (match var-val
1196                                     ((= v val) [var-val (inc cost)])))
1197                                 values)))
1198      [main-constraint vars cost-constraints]))
1199
1200  (define (test-max n)
1201    (let (([constraint vars cost-constraints] (make-max-constraint n))
1202          (L (add constraint cost-constraints)))
1203      (max-smt-solve L)))
1204
1205  (define (testtc n)
1206    (let ((c (first (make-constraint n))))
1207    (running-time (lambda () (tc c)) 0)))
1208
1209  # MLton results:
1210  # (testtc 300) -> 0.15
1211  # (testtc 500) -> 0.33
1212  # (testtc 700) -> 0.66
1213
1214  # SML-NJ results:
1215  # (testtc 300) -> 0.32 seconds
1216  # (testtc 500) -> 0.92 seconds
1217  # (testtc 700) -> 1.90
1218
1219  EOF
1220
1221  # (load-file "smt.ath")
1222
1223  (domains D1 D2)
1224
1225  (declare f1 (-> (D1) D1))
1226  (declare g2 (-> (D1 D2) D1))
1227
1228  (declare d1 D1)
1229  (declare d2 D2)
1230
1231  (datatype Day
1232    Mon Tue Wed Thu Fri Sat Sun)
1233
1234  (declare nextDay (-> (Day) Day))
1235
1236  (define nextDay-axioms
1237     [(nextDay Mon = Tue)
1238      (nextDay Tue = Wed)
1239      (nextDay Wed = Thu)
1240      (nextDay Thu = Fri)
1241      (nextDay Fri = Sat)
1242      (nextDay Sat = Sun)
1243      (nextDay Sun = Mon)])
1244
1245  (datatype Color
1246    red blue green)
1247
1248  (datatype IntList
1249    nil
1250    (cons HEAD:Int IntList))
1251
1252  (datatype IntList2
1253    nil2
1254    (cons2 HEAD2:Int TAIL2:IntList2))
1255
1256  (define (test n x) (running-time (lambda () (test-max n)) x))
```

```
(define large-constraint
  (let (([A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14 A15 A16 A17 A18 A19 A20]
         [?A1:Int ?A2:Int ?A3:Int ?A4:Int ?A5:Int ?A6:Int ?A7:Int ?A8:Int ?A9:Int ?A10:Int
          ?A11:Int ?A12:Int ?A13:Int ?A14:Int ?A15:Int ?A16:Int ?A17:Int ?A18:Int ?A19:Int ?A20:Int]))
    (or (and (A1 in [10 20])
             (A2 in [30 40])
             (A3 in [40 50])
             (A4 in [50 60])
             (A5 in [60 70])
             (A6 in [70 80])
             (A7 in [80 90])
             (A8 in [90 100])
             (A9 in [100 110])
             (A10 in [110 120])
             (A11 in [120 130])
             (A12 in [130 140])
             (A13 in [140 150])
             (A14 in [150 160])
             (A15 in [160 170])
             (A16 in [170 180])
             (A17 in [180 190])
             (A18 in [190 200])
             (A19 in [200 210])
             (A20 in [210 220]))
        (and (A1 in [210 220])
             (A2 in [230 240])
             (A3 in [240 250])
             (A4 in [250 260])
             (A5 in [260 270])
             (A6 in [270 280])
             (A7 in [280 290])
             (A8 in [290 2100])
             (A9 in [2100 2110])
             (A10 in [2110 2120])
             (A11 in [2120 2130])
             (A12 in [2130 2140])
             (A13 in [2140 2150])
             (A14 in [2150 2160])
             (A15 in [2160 2170])
             (A16 in [2170 2180])
             (A17 in [2180 2190])
             (A18 in [2190 2200])
             (A19 in [2200 2210])
             (A20 in [2210 2220])))))

(define cost-function
  (and (ite (= ?A1:Int 13) (= ?costA1:Int 0)
                           (= ?costA1:Int 3))
       (ite (= ?A2:Int 35) (= ?costA2:Int 0)
                           (= ?costA2:Int 2))
       (ite (= ?A19:Int 2207) (= ?costA19:Int 0)
                              (= ?costA19:Int 6))))

(define cost-term (?costA1:Int + ?costA2:Int + ?costA19:Int))

(smt-solve-and-minimize (and large-constraint cost-function)
                        cost-term 11)

(define [constraint-2 vars-2 cost-constraint-2 cost-term-2 max-cost-2] (make-constraint 2))

(smt-solve-and-minimize (and constraint-2 cost-constraint-2) cost-term-2 max-cost-2)

(define [constraint-30 vars-30 cost-constraint-30 cost-term-30 max-cost-30] (make-constraint 30))

(smt-solve constraint-30)

(running-time (lambda () (smt-solve-and-minimize (and constraint-30 cost-constraint-30) cost-term-30 max-cost-30)) 0)

(define [constraint-50 vars-50 cost-constraint-50 cost-term-50 max-cost-50] (make-constraint 50))
```

```
1327
1328 (smt-solve constraint-50)
1329
1330 (running-time (lambda () (smt-solve constraint-50)) 0)
1331
1332 (running-time (lambda () (smt-solve-and-minimize (and constraint-50 cost-constraint-50) cost-term-50 max-cost-50)) 0)
1333
1334 (smt-solve-and-minimize (and constraint-50 cost-constraint-50) cost-term-50 max-cost-50)
1335
1336 (define [constraint-70 vars-70 cost-constraint-70 cost-term-70 max-cost-70] (make-constraint 70))
1337
1338 (running-time (lambda () (smt-solve-and-minimize (and constraint-70 cost-constraint-70) cost-term-70 max-cost-70)) 0)
1339
1340 (define [constraint-100 vars-100 cost-constraint-100 cost-term-100 max-cost-100] (make-constraint 100))
1341
1342 (running-time (lambda () (smt-solve constraint-100)) 0)
1343
1344 (define [constraint-200 vars-200 cost-constraint-200 cost-term-200 max-cost-200] (make-constraint 200))
1345
1346 (smt-solve constraint-200)
1347
1348 (define [constraint-300 vars-300 cost-constraint-300 cost-term-300 max-cost-300] (make-constraint 300))
1349
1350 (smt-solve constraint-300)
1351
1352 (define [constraint-400 vars-400 cost-constraint-400 cost-term-400 max-cost-400] (make-constraint 400))
1353
1354 (smt-solve constraint-400)
1355
1356 (define [constraint-500 vars-500 cost-constraint-500 cost-term-500 max-cost-500] (make-constraint 500))
1357
1358 (smt-solve constraint-500)
1359
1360 (running-time (lambda () (smt-solve constraint-500)) 0)
1361
1362 # MLton time -> 0.56 seconds
1363 # SMLNJ time -> 1.14 seconds
1364
1365 #============================================================================
1366
1367 (define large-max-constraint
1368   [[large-constraint 'inf]
1369    [(= ?A1:Int 13) 2]
1370    [(= ?A2:Int 35) 3]
1371    [(= ?A3:Int 45) 4]
1372    [(= ?A4:Int 55) 5]
1373    [(= ?A5:Int 55) 6]
1374    [(= ?A6:Int 1999) 5]
1375    [(= ?A7:Int 82) 7]
1376    [(= ?A8:Int 93) 8]
1377    [(= ?A9:Int 105) 9]
1378    [(= ?A10:Int 114) 10]
1379    [(= ?A11:Int 123) 11]
1380    [(= ?A12:Int 133) 30]
1381    [(= ?A13:Int 145) 12]
1382    [(= ?A14:Int 155) 13]
1383    [(= ?A15:Int 0) 13]
1384    [(= ?A16:Int 88888) 13]
1385    [(= ?A17:Int 88888) 20]
1386    [(= ?A18:Int 192) 20]
1387    [(= ?A19:Int 200) 20]
1388    [(= ?A20:Int 200) 20]])
1389
1390 (max-smt-solve large-max-constraint)
1391
1392 #============================================================================
1393
1394 (test-max 10)
1395
1396 (test-max 20)
```

```
(test-max 50)

(test-max 70)

(test-max 100)

(test-max 140)

(test-max 200)

(test-max 300)

#=============================================================================

(define [min-diff-constraint total-diff]
  (let (([A B C] [?A:Int ?B:Int ?C:Int])
        ([minDiffA minDiffB minDiffC] [?minDiffA:Int ?minDiffB:Int ?minDiffC:Int])
        (constraint (or (and (A in [10 20])
                             (B in [1 20])
                             (C in [720 800]))
                        (and (A in [500 600])
                             (B in [30 40])
                             (C in [920 925])))))
        (minDiffA-def (ite (> A 13) (= minDiffA (- A 13))
                                    (= minDiffA (- 13 A))))
        (minDiffB-def (ite (> B 15) (= minDiffB (- B 15))
                                    (= minDiffB (- 15 B))))
        (minDiffC-def (ite (> C 922) (= minDiffC (- C 922))
                                     (= minDiffC (- 922 C)))))
   [(and constraint minDiffA-def minDiffB-def minDiffC-def)
    (sum-all [minDiffA minDiffB minDiffC])]))

(max-smt-solve [[min-diff-constraint 'inf]
                [(= ?A:Int 13) 10]
                [(= ?B:Int 15) 20]
                [(= ?C:Int 15) 25]])

(smt-solve-and-minimize min-diff-constraint total-diff 5000)


#;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define large-constraint
  (let (([A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14 A15 A16 A17 A18 A19 A20]
         [?A1:Int ?A2:Int ?A3:Int ?A4:Int ?A5:Int ?A6:Int ?A7:Int ?A8:Int ?A9:Int ?A10:Int
          ?A11:Int ?A12:Int ?A13:Int ?A14:Int ?A15:Int ?A16:Int ?A17:Int ?A18:Int ?A19:Int ?A20:Int]))
    (or (and (A1 in [10 20])
             (A2 in [30 40])
             (A3 in [40 50])
             (A4 in [50 60])
             (A5 in [60 70])
             (A6 in [70 80])
             (A7 in [80 90])
             (A8 in [90 100])
             (A9 in [100 110])
             (A10 in [110 120])
             (A11 in [120 130])
             (A12 in [130 140])
             (A13 in [140 150])
             (A14 in [150 160])
             (A15 in [160 170])
             (A16 in [170 180])
             (A17 in [180 190])
             (A18 in [190 200])
             (A19 in [200 210])
             (A20 in [210 220]))
        (and (A1 in [210 220])
             (A2 in [230 240])
             (A3 in [240 250])
```

```
             (A4 in [250 260])
             (A5 in [260 270])
             (A6 in [270 280])
             (A7 in [280 290])
             (A8 in [290 2100])
             (A9 in [2100 2110])
             (A10 in [2110 2120])
             (A11 in [2120 2130])
             (A12 in [2130 2140])
             (A13 in [2140 2150])
             (A14 in [2150 2160])
             (A15 in [2160 2170])
             (A16 in [2170 2180])
             (A17 in [2180 2190])
             (A18 in [2190 2200])
             (A19 in [2200 2210])
             (A20 in [2210 2220])))))

(define cost-function
  (and (ite (= ?A1:Int 13) (= ?costA1:Int 0)
                           (= ?costA1:Int 3))
       (ite (= ?A2:Int 35) (= ?costA2:Int 0)
                           (= ?costA2:Int 2))
       (ite (= ?A19:Int 2207) (= ?costA19:Int 0)
                              (= ?costA19:Int 6))))

(define cost-term (?costA1:Int + ?costA2:Int + ?costA19:Int))

(smt-solve-and-minimize (and large-constraint cost-function)
                        cost-term 11)

(smt-solve-and-minimize (and large-constraint cost-function)
                        cost-term  788000034)

#;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
#;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(define L [[(or (and (= ?x 2) (= ?y 3.4))
                (and (= ?x 5) (= ?d Mon))) 10]
           [(or (and (= ?x 2) (= ?y 3.4))
                (and (= ?a 5) (= ?d Mon))) 20]])

(max-smt-solve L)

(define L1 [[(or (and (= ?x 2) (= ?y 3.4))
                 (and (= ?x 5) (= ?d Mon))) 10]
            [(and (= ?x 99) (= ?y 99.9) (= ?a 9999) (= ?d Sun)) 20]])

(max-smt-solve L1)

(define [constraint-2 vars-2 cost-constraint-2 cost-term-2 max-cost-2] (make-constraint 2))

(smt-solve-and-minimize (and constraint-2 cost-constraint-2) cost-term-2 max-cost-2)

(define [constraint-5 vars-5 cost-constraint-5 cost-term-5 max-cost-5] (make-constraint 5))

(smt-solve-and-minimize (and constraint-5 cost-constraint-5) cost-term-5 max-cost-5)

(define [constraint-20 vars-20 cost-constraint-20 cost-term-20 max-cost-20] (make-constraint 20))

(smt-solve-and-minimize (and constraint-20 cost-constraint-20) cost-term-20 max-cost-20)

(define [constraint-30 vars-30 cost-constraint-30 cost-term-30 max-cost-30] (make-constraint 30))

(define (thunk-30)
  (smt-solve-and-minimize (and constraint-30 cost-constraint-30) cost-term-30 max-cost-30))

(running-time thunk-30 0)
```

```
(define [constraint-50 vars-50 cost-constraint-50 cost-term-50 max-cost-50] (make-constraint 50))

(define (thunk-50)
  (smt-solve-and-minimize (and constraint-50 cost-constraint-50) cost-term-50 max-cost-50))

(smt-solve (and constraint-50 cost-constraint-50))

(running-time thunk-50 0)

(define [constraint-60 vars-60 cost-constraint-60 cost-term-60 max-cost-60] (make-constraint 60))

(running-time (lambda () (smt-solve-and-minimize (and constraint-60 cost-constraint-60) cost-term-60 max-cost-60)) 0)

(define [constraint-70 vars-70 cost-constraint-70 cost-term-70 max-cost-70] (make-constraint 70))

(define (thunk-70)
  (smt-solve-and-minimize (and constraint-70 cost-constraint-70) cost-term-70 max-cost-70))

(running-time thunk-70 0)

(define [constraint-90 vars-90 cost-constraint-90 cost-term-90 max-cost-90] (make-constraint 90))

(smt-solve constraint-90)

(smt-solve (and constraint-90 cost-constraint-90))

(define [constraint-150 vars-150 cost-constraint-150 cost-term-150 max-cost-150] (make-constraint 150))

(smt-solve constraint-150)

(smt-solve (and constraint-150 cost-constraint-150))

(define [constraint-200 vars-200 cost-constraint-200 cost-term-200 max-cost-200] (make-constraint 200))

(smt-solve constraint-200)

(smt-solve (and constraint-200 cost-constraint-200))

(define [constraint-300 vars-300 cost-constraint-300 cost-term-300 max-cost-300] (make-constraint 300))

(smt-solve constraint-300)

(smt-solve (and constraint-300 cost-constraint-300))

(define [constraint-500 vars-500 cost-constraint-500 cost-term-500 max-cost-500] (make-constraint 500))

(smt-solve constraint-500)

##;;;;;;;;;

(smt-solve (and constraint-30 cost-constraint-30))

(smt-solve (and constraint-30 cost-constraint-30 (0 <= cost-term-30) (cost-term-30 <= 232)))
(smt-solve (and constraint-30 cost-constraint-30 (232 <= cost-term-30) (cost-term-30 <= 465)))

(smt-solve (and constraint-30 cost-constraint-30 (= ?TOTALCOST:Int cost-term-30)
                                          (0 <= ?TOTALCOST:Int)
                                          (?TOTALCOST:Int <= 465)))


(define [constraint-30 vars-30 cost-constraint-30 cost-term-30] (make-constraint 30))

(define [constraint30 vars] (make-constraint 30))
(define [constraint40 vars] (make-constraint 40))
(define [constraint50 vars] (make-constraint 50))
(define [constraint100 vars] (make-constraint 100))


#;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
#;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
1607
1608
1609 # Some simple generic examples:
1610
1611
1612 (define c1  (or (and (= ?x 2) (= ?y 3.4))
1613                 (and (= ?x 5) (= ?d Mon))))
1614
1615 (smt-solve c1)
1616
1617 (define c2  (or (and (= ?x 2) (= ?y 3.4))
1618                 (and (= ?a 5) (= ?d Mon))))
1619
1620 (smt-check c2 'Satisfiable)
1621
1622 (define c3 (and (nextDay ?d = Thu)
1623                 (?x:Int =/= 2)
1624                 (?x:Int =/= 5)
1625                 (?color1 = red)
1626                 (conjoin nextDay-axioms)))
1627
1628 (smt-solve c3)
1629
1630 (smt-check c3 'Satisfiable)
1631
1632 (define c4  (or (and (= ?x 2) (= ?y 3.4))
1633                 (and (?c =/= blue) (?c =/= green))
1634                 (and (= ?x 5) (= ?d Mon))))
1635
1636 (smt-solve c4)
1637 (smt-check c4 'Satisfiable)
1638
1639 (define c5 (and (?x:Int <= 2)
1640                 (?x:Int >= 0)
1641                 (or (= ?w Mon)
1642                     (not (= ?w Fri)))))
1643
1644 (smt-solve c5)
1645 (smt-check c5 'Satisfiable)
1646
1647 (define c6 (or (and (= ?x 2) (= ?a 3))
1648                (and (= ?d Mon) (= Mon Tue))))
1649
1650
1651 (smt-solve c6)
1652
1653 (smt-check c6 'Satisfiable)
1654
1655 (define c7 (c6 & ~ c6))
1656
1657 (smt-solve c7)
1658 (smt-check c7 'Unsatisfiable)
1659
1660 (define c8 (and (f1 ?x = ?y)
1661                 (= ?y d1)
1662                 (forall ?foo . (not (= (f1 ?foo) d1)))))
1663
1664
1665 (smt-solve c8)
1666 (smt-check c8 'Unsatisfiable)
1667
1668 (define (square x) (x * x))
1669
1670 (define c9 (= (3 * (square ?x:Real)) 12))
1671
1672 (smt-solve c9)
1673 (smt-check c9 'Satisfiable)
1674
1675 (define c10 (and ((square ?x:Int) = 25)
1676                  (?x:Int + ?y:Int = (2 * ?x:Int) - 1)))
```

```
1677
1678 (smt-solve c10)
1679 (smt-check c10 'Satisfiable)
1680 (smt-check c10 'Unsatisfiable)
1681
1682
1683 (define c11
1684   (?color =/= red & ?color =/= blue & ?color =/= green))
1685
1686 (smt-solve c11)
1687
1688 (define c12
1689   (and (not (= ?l nil))
1690        (not (exists ?h (exists ?t (= ?l (cons ?h ?t)))))))
1691
1692
1693 (smt-solve c12)
1694
1695 (define c13
1696    (not (= (HEAD (cons ?x ?y)) ?x)))
1697
1698 (smt-solve c13)
1699
1700 (domain U)
1701
1702 (declare F (-> (U) U))
1703 (declare foo U)
1704
1705 (define c14 (and (= (F foo) foo)
1706                  (not (= (F ?x) ?x))))
1707
1708 (smt-solve c14)
1709
1710
1711 (define c15 (and (= (F foo) foo)
1712                  (not (= (F ?x) ?x))
1713                  (= (F ?y) ?x)))
1714
1715 (smt-solve c15)
1716
1717 (datatype IntList
1718   null (cons Int IntList))
1719
1720
1721 (declare append (-> (IntList IntList) IntList))
1722
1723 (define ap-axiom-1
1724   (close (= (append null ?l) ?l)))
1725
1726 (define ap-axiom-2
1727   (close (= (append (cons ?x ?l1) ?l2)
1728             (cons ?x (append ?l1 ?l2)))))
1729
1730 (define ap-axioms [ap-axiom-1 ap-axiom-2])
1731
1732 (assert ap-axioms)
1733
1734 (define c1 (not (= (append ?c1 ?c2) (append ?c2 ?c1))))
1735
1736 (smt-solve (and* (add c1 ap-axioms)))
```