# lib/basic/smt-cvc.ath

```
#;;;;;;;;;;;;;;;; Athena code for using CVC3 ;;;;;;;;;;;;;;;;

#### Simple maps (association lists).

(define empty-map [])

(define (add-binding x y map)
  (add [x y] map))

(define (remove-binding x map)
  (match map
    ([] [])
    ((list-of [key value] more) (check ((equal? key x) more)
                                       (else (add-binding key value (remove-binding x more)))))))

(define (apply-map map x)
  (match map
    ([] ())
    ((list-of [a b] more) (check ((equal? a x) b)
                                 (else (apply-map more x))))))

(define (dom m)
  (letrec ((loop (lambda (m res)
                   (match m
                     ([] res)
                     ((list-of [a b] more) (check ((member? a res) (loop more res))
                                                  (else (loop more (add a res)))))))))
    (loop m [])))


(define (map-range m)
  (map (lambda (x) (apply-map m x))
       (dom m)))

(define (dom-range-list m)
  (map (lambda (x) [x (apply-map m x)])
       (dom m)))

(define (in-dom? a m)
  (match m
    ([] false)
    ((list-of [x _] rest) (|| (equal? a x)
                              (in-dom? a rest)))))

############ SMT code:

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

(define [c-comma c-lparen c-rparen] [`, `( `)])

(define (get-signature' f)
  (let ((rename (lambda (sort)
                  (match sort
                    ("Int" "INT")
                    ("Real" "REAL")
                    (_ sort)))))
    (map rename (get-signature f))))

(define counter (cell 0))

(define (mprint x) ())

(define (separate L token)
  (match L
    ([] "")
    ([s] s)
    ((list-of s1 (bind rest (list-of _ _)))
```

```
69        (join s1 token (separate rest token)))))
70
71  (define (separate-all-but-last L token)
72    (match L
73      ([] "")
74      ([_] "")
75      ((list-of s1 (bind rest (list-of _ _)))
76        (let ((str (separate-all-but-last rest token)))
77          (check ((null? str) s1)
78                 (else (join s1 token str)))))))
79
80  (define (integer? n)
81    (match n
82      (x:Int true)
83      (_ false)))
84
85  (define (real? n)
86    (match n
87      (x:Real true)
88      (_ false)))
89
90  (define (proper-real? n)
91    (&& (real? n) (negate (integer? n))))
92
93  (define (real->rational x)
94    (match (val->string x)
95      ((split integral (list-of `. decimal))
96         (let ((d (raise 10 (length decimal))))
97           [(string->num (join integral decimal)) d]))))
98
99  (define (rational->real n-str d-str)
100   (let ((n (string->num (join n-str ".0")))
101         (d (string->num d-str)))
102    (div n d)))
103
104 (define (integer-numeral? n)
105   (&& (numeral? n) (integer? n)))
106
107 (define (real-numeral? n)
108   (&& (numeral? n) (real? n)))
109
110 (define (make-fresh prefix counter vmap-range)
111   (let ((first-attempt (join prefix (val->string (inc counter)))))
112     (check ((for-some vmap-range
113                       (lambda (v)
114                         (match v
115                           ((list-of vname _) (equal? vname first-attempt))
116                           (_ false))))
117               (make-fresh prefix counter vmap-range))
118             (else first-attempt))))
119
120 (define (translate-var x vmap counter)
121   (match (apply-map vmap x)
122     (() (let ((var-name (check ((var? x) (var->string x))
123                                (else (val->string x))))
124               (c (first var-name))
125               (var-name' (check ((&& (alpha-char? c) (alpha-numeric-string? var-name)) var-name)
126                                 (else (make-fresh counter (map-range vmap)))))
127               (var-sort (sort-of x))
128               (res (match var-sort
129                      ("Int"  [var-name' "INT"])
130                      ("Real" [var-name' "REAL"])
131                      (_      (match (constructors-of var-sort)
132                                ([] [var-name' var-sort])
133                                ((some-list cl) [var-name' var-sort (map val->string cl)])))))))
134           [var-name' (add-binding x res vmap)]))
135     ((list-of var-name _) [var-name vmap])))
136
137 (define (binary-infix? f args)
138   (&& (equal? (length args) 2)
```

```
      (member? f [< > <= >= = + - * /])))

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

(define (translate-term t vmap counter)
  (match t
    (((some-symbol f) (bind args (list-of _ _)))
       (let (([arg-strings vmap'] (translate-terms args vmap counter))
             (res-string (check ((binary-infix? f args)
                                 (join lparen (first arg-strings) blank (symbol->string f) blank
                                       (second arg-strings) rparen))
                                (else (make-generic-term-string f arg-strings))))
             (f-string (symbol->string f))
             (vmap'' (match (apply-map vmap' f-string)
                       (() (check ((binary-infix? f [1 2]) vmap')
                                  (else (add-binding f-string (add 'function (get-signature' f)) vmap'))))
                       (_ vmap'))))
         [res-string vmap'']))
    ((some-var x) (translate-var x vmap counter))
    (_ (check ((&& (real-numeral? t) (negate (integer? t)))
                (let (([n d] (real->rational t))
                      (str (join (val->string n) "/" (val->string d))))
                 [str vmap]))
              ((integer-numeral? t) [(val->string t) vmap])
              (else (translate-var t vmap counter))))))
   (translate-terms t's vmap counter)
      (letrec ((loop (lambda (terms strings vmap counter)
                       (match terms
                         ([] [(rev strings) vmap])
                         ((list-of t rest)
                            (let (([t-string vmap'] (translate-term t vmap counter)))
                              (loop rest (add t-string strings) vmap' counter)))))))
         (loop t's [] vmap counter)))


(define (translate-relation-symbol R)
  (join blank (symbol->string R) blank))

(define (translate-atomic-constraint c var-map counter)
  (match c
    ((R (some-term t1) (some-term t2))
         (let (([t1-string var-map']  (translate-term t1 var-map counter))
               ([t2-string var-map''] (translate-term t2 var-map' counter))
               (R-sign (translate-relation-symbol R))
               (res-string (join lparen t1-string R-sign t2-string rparen)))
           [res-string var-map'']))))

(define (sc->string sc)
  (match sc
    (not "NOT")
    (and "AND")
    (or  "OR")
    (if  "=>")
    (iff "<=>")))

(define (make-constraint sc strings)
   (let ((sc-string (sc->string sc)))
     (letrec ((loop (lambda (strings)
                      (match strings
                        ([s] s)
                        ((list-of s more) (join lparen s blank sc-string blank (loop more) rparen))))))
        (loop strings))))

(define (translate-constraint c counter)
  (letrec ((tran (lambda (c var-map)
                   (match c
                     ((some-atom _) (translate-atomic-constraint c var-map counter))
                     ((not c') (let (([string var-map'] (tran c' var-map)))
                                 [(join lparen "NOT " string rparen)
```

```
209                                  var-map']))
210                       (((some-sent-con sc) (some-list constraints))
211                          (let (([strings var-map'] (tran* constraints var-map [])))
212                            [(make-constraint sc strings) var-map']))
213                       ((forall (some-var x) body)
214                          (let (([body-string var-map'] (tran body var-map))
215                                ([var-name var-sort] (match (apply-map var-map' x)
216                                                       (() [(var->string (fresh-var)) "INT"])
217                                                       (res [(first res) (second res)])))
218                                (str (join lparen "FORALL " lparen var-name colon var-sort rparen
219                                           colon blank body-string rparen))
220                                (var-map'' (remove-binding x var-map'))
221                                (var-map''' (add-binding x [(make-fresh "a_a_a" counter
222                                                              (map-range var-map'')) var-sort] var-map'')))
223                            [str var-map''']))))))
224              (tran* (lambda (constraints var-map strings)
225                       (match constraints
226                         ([] [strings var-map])
227                         ((list-of c more) (let (([c-string var-map'] (tran c var-map)))
228                                             (tran* more var-map' (add c-string strings))))))))
229       (tran c empty-map)))
230
231   (define (tc c)
232     (translate-constraint c counter))
233
234   (define (tct c)
235     (let (([str map] (tc c))
236           (_ (print "\nString: " str))
237           (_ (print "\n\nVmap: " map "\n\n")))
238       [str map]))
239
240   (define (constructor-name? str)
241     (try (constructor? (string->symbol str))
242          false))
243
244   (define (get-declarations c)
245     (let (([str vmap] (tc c))
246           (dom-range (dom-range-list vmap)))
247       (letrec ((loop (lambda (dom-range domains-so-far domain-decs var-decs reverse-vmap)
248                        (match dom-range
249                          ([] [domain-decs var-decs reverse-vmap])
250                          ((list-of [var (list-of 'function rest)] more)
251                              (let ((new-vdec (join var colon blank lparen (separate-all-but-last rest comma) rparen " ->
252                                 (loop more domains-so-far domain-decs (add new-vdec var-decs) reverse-vmap)))
253                          ((list-of [var var-value] more)
254                            (let ((var-name (first var-value))
255                                  (var-type (second var-value))
256                                  (new-vdec (join var-name colon blank var-type scolon))
257                                  ([reverse-vmap' var-decs'] (check ((constructor-name? var-name) [reverse-vmap var-decs]
258                                                                     (else [(add-binding var-name var reverse-vmap)
259                                                                            (add new-vdec var-decs)])))))
260                              (check (((|| (equal? var-type "INT") (equal? var-type "REAL"))
261                                       (loop more domains-so-far domain-decs var-decs' reverse-vmap'))
262                                      ((member? var-type domains-so-far)
263                                        (loop more domains-so-far domain-decs var-decs' reverse-vmap'))
264                                      (else (let ((new-domain-dec (check ((datatype-sort? var-type)
265                                                                          (join "\nDATATYPE\n"  var-type " = " (separate
266                                                                        (else (join "\n" var-type colon blank "TYPE;\n"))
267                                                 (loop more (add var-type domains-so-far)
268                                                            (add new-domain-dec domain-decs)
269                                                            var-decs' reverse-vmap')))))))))))
270         (join [str vmap] (loop dom-range [] [] [] empty-map)))))
271
272
273   (define (get-line str)
274     (letrec ((loop (lambda (str chars)
275                      (match str
276                        ([] [(rev chars) []])
277                        ((list-of '\n rest) [(rev (add '\n chars)) rest])
278                        ((list-of c rest)  (loop rest (add c chars)))))))
```

```
279     (loop str [])))
280
281
282 (define (get-val str reverse-vmap)
283   (try (string->num str)
284        (match str
285          ((split n-str (split "/" d-str)) (rational->real n-str d-str))
286          (_ (try ((string->symbol str))
287                  (apply-map reverse-vmap str))))))
288
289 (define (get-val1 str)
290   (try (string->num str)
291        (match str
292          ((split n-str (split "/" d-str)) (rational->real n-str d-str))
293          (_ (string->symbol str)))))
294
295 (define (skip? left right)
296    (let ((skipable (lambda (str)
297                      (match str
298                        ((split "LET" _) true)
299                        ((split "(LAMBDA" _) true)
300                        (_ false)))))
301      (|| (skipable left) (skipable right))))
302
303
304 (define (parse-cvc-term str reverse-map)
305   (letrec ((get-functor (lambda (str res)
306                           (match str
307                             ((list-of (val-of c-lparen) rest) [(rev res) str])
308                             ((list-of (val-of c-rparen) rest) [(rev res) str])
309                             ((list-of (val-of c-comma) rest) [(rev res) str])
310                             ((list-of (some-char c) rest) (get-functor rest (add c res)))
311                             ([] [(rev res) []])))))
312            (get-term (lambda (str)
313                        (match (get-functor str [])
314                          ([functor (list-of (val-of c-lparen) rest)]
315                                         (let (([args rest'] (get-terms rest []))
316                                           [(make-term (string->symbol functor) args) rest']))
317                          ([functor rest] (match (apply-map reverse-map functor)
318                                            (() (try [(get-val1 functor) rest]
319                                                     [1 rest]))
320                                            (x [x rest])))))))
321            (get-terms (lambda (str results)
322                         (match str
323                           ([] [(rev results) []])
324                           (_ (match (get-term str)
325                                ([term (list-of (val-of c-comma) rest)] (get-terms rest (add term results)))
326                                ([term (list-of (val-of c-rparen) rest)] [(rev (add term results)) rest])))))))
327      (first (get-term str))))
328
329
330 (define (process-cvc-output reverse-vmap file conjuncts)
331    (let ((data (read-file file))
332          ([line1 rest1] (get-line data)))
333      (letrec ((get-model (lambda (str L)
334                            (let (([line rest] (get-line str)))
335                              (match line
336                                ((split "ASSERT (" (split left (split " = " (split right ");\n"))))
337                                   (check ((skip? left right) (get-model rest L))
338                                          (else (let ((left-term (parse-cvc-term left reverse-vmap))
339                                                      (right-term (parse-cvc-term right reverse-vmap))
340                                                      (identity  (= left-term right-term)))
341                                                  (check ((equal? left-term right-term) (get-model rest L))
342                                                         ((&& (ground? identity) (member? identity conjuncts)) (get-mod
343                                                         (else (get-model rest (add identity L))))))))
344                                ([] L)
345                                (_ (get-model rest L)))))))
346        (check ((equal? line1 "Valid.\n") 'Unsatisfiable)
347               ((equal? line1 "Unknown.\n") 'Unknown)
348               (else (get-model rest1 [])))))))
```

```
349
350 (define (athena->cvc c file)
351   (let (([str vmap d-decs v-decs reverse-vmap] (get-declarations c))
352         (_ (write-file file "%%% Type declarations:\n"))
353         (_ (write-file file (separate d-decs "\n")))
354         (_ (write-file file "\n%%% Variable declarations:\n\n"))
355         (_ (write-file file (separate v-decs "\n")))
356         (_ (write-file file "\n\n%%% Query the negation: \n\n"))
357         (_ (write-file file (join "QUERY NOT " str ";\n\n")))
358         (_ (write-file file (join "COUNTERMODEL;\n"))))
359     ()))
360
361
362 (define (cvc-test c)
363   (let (([input-file output-file] ["input1.cvc" "output1.cvc"])
364         ([str vmap d-decs v-decs reverse-vmap] (get-declarations c))
365         (_ (print "%%% Type declarations:\n"))
366         (_ (print (separate d-decs "\n")))
367         (_ (print "\n%%% Variable declarations:\n\n"))
368         (_ (print (separate v-decs "\n")))
369         (_ (print (join "\n\nformula to be queried: " str ";\n\n")))
370         (_ (print "\n\nvmap: " vmap))
371         (_ (print "\n\nReverse vmap: " reverse-vmap)))
372     [str vmap d-decs v-decs reverse-vmap]))
373
374 (define (cvc-solve-core c)
375   (let (([input-file output-file error-file] ["input1.cvc" "output1.cvc" "error.cvc"])
376         (_ (delete-files [input-file output-file error-file]))
377         ([str vmap d-decs v-decs reverse-vmap] (get-declarations c))
378         (_ (mprint "\nDone with translation...\n"))
379         (_ (write-file input-file "%% Type declarations:\n"))
380         (_ (write-file input-file (separate d-decs "\n")))
381         (_ (write-file input-file "\n%% Variable declarations:\n"))
382         (_ (write-file input-file (separate v-decs "\n")))
383         (_ (write-file input-file "\n%% Query the negation: \n"))
384         (_ (write-file input-file (join "QUERY NOT " str ";\n")))
385         (_ (write-file input-file (join "COUNTERMODEL;\n")))
386         (_ (mprint "\nSending OS command...\n"))
387         (_ (exec-command (join "cvc3 -timeout 60 " input-file " > " output-file " 2> " error-file))))
388     (process-cvc-output reverse-vmap output-file (get-conjuncts c))))
389
390 (define (cvc-solve c)
391   (cvc-solve-core (rename c)))
392
393 (define (cvc-multiple-models c max)
394   (let ((negate-model (lambda (model)
395                         (and* (map not model)))))
396     (letrec ((loop (lambda (c i models)
397                      (check ((less? i max)
398                              (match (cvc-solve c)
399                                ((some-list model) (loop (and c (negate-model model))
400                                                         (plus i 1)
401                                                         (add model models)))
402                                (_ models)))
403                             (else models)))))
404       (loop c 0 [])))))
```