```scheme
#| -*-Scheme-*-

$Id$

Copyright 2006 Massachusetts Institute of Technology

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or (at
your option) any later version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301,
USA.

|#

;;;; Boolean constraint types

(declare (usual-integrations))

(define cp:B=~A
  (cp:make-constraint-type 'B=~A '(b a)
    (lambda (constraint)
      (let ((b (cp:get-connector constraint 'b))
            (a (cp:get-connector constraint 'a)))
        (cond ((cp:has-value? a)
               (cp:set-value b
                             (not (cp:value-of a))
                             (list a)
                             constraint))
              ((cp:has-value? b)
               (cp:set-value a
                             (not (cp:value-of b))
                             (list b)
                             constraint)))))))
```

```
(define cp:C=A^B
  (cp:make-constraint-type 'C=A^B '(c b a)
    (lambda (constraint)
      (let ((c (cp:get-connector constraint 'c))
            (b (cp:get-connector constraint 'b))
            (a (cp:get-connector constraint 'a)))
        (cond ((and (cp:has-value? a)
                    (cp:has-value? b))
               (cp:set-value c
                             (and (cp:value-of a) (cp:value-of b))
                             (list a b)
                             constraint))
              ((not (cp:has-value? c))
               (cond ((cp:value-is? a #f)
                      (cp:set-value c #f (list a) constraint))
                     ((cp:value-is? b #f)
                      (cp:set-value c #f (list b) constraint))))
              ((cp:value-is? c #f)
               (cond ((cp:value-is? a #t)
                      (cp:set-value b #f (list a c) constraint))
                     ((cp:value-is? b #t)
                      (cp:set-value a #f (list b c) constraint))))
              (else
               (cp:set-value a #t (list c) constraint)
               (cp:set-value b #t (list c) constraint)))))))

(define cp:C=AvB
  (cp:make-constraint-type 'C=AvB '(c b a)
    (lambda (constraint)
      (let ((c (cp:get-connector constraint 'c))
            (b (cp:get-connector constraint 'b))
            (a (cp:get-connector constraint 'a)))
        (cond ((and (cp:has-value? a)
                    (cp:has-value? b))
               (cp:set-value c
                             (or (cp:value-of a) (cp:value-of b))
                             (list a b)
                             constraint))
              ((not (cp:has-value? c))
               (cond ((cp:value-is? a #t)
                      (cp:set-value c #t (list a) constraint))
                     ((cp:value-is? b #t)
                      (cp:set-value c #t (list b) constraint))))
              ((cp:value-is? c #t)
               (cond ((cp:value-is? a #f)
                      (cp:set-value b #t (list a c) constraint))
                     ((cp:value-is? b #f)
                      (cp:set-value a #t (list b c) constraint))))
              (else
               (cp:set-value a #f (list c) constraint)
               (cp:set-value b #f (list c) constraint)))))))
```

```
(let ((cn
       (cp:make-network 'test
                        (cp:make-value-model boolean?
                                             boolean=?
                                             (lambda (x y) #f)))))
  (let ((j1 (cp:make-connector 'j1 cn))
        (j2 (cp:make-connector 'j2 cn))
        (p1 (cp:make-connector 'p1 cn))
        (p2 (cp:make-connector 'p2 cn))
        (t (cp:make-connector 't cn)))
    (cp:assume-value t #t)
    (cp:make-constraint cp:B=~A #f cn j1 p1)
    (cp:make-constraint cp:B=~A #f cn j2 p2)
    (cp:make-constraint cp:C=AvB #f cn t j1 j2)
    (cp:make-constraint cp:C=AvB #f cn t p1 p2)
    (cp:assume-value j2 #t)
    (write-line (cp:value-of j1))
    (write-line (cp:value-of j2))
    (write-line (cp:value-of p1))
    (write-line (cp:value-of p2))
    ))
#f
#t
#t
#f
;Unspecified return value

(let ((cn
       (cp:make-network 'test
                        (cp:make-value-model boolean?
                                             boolean=?
                                             (lambda (x y) #f)))))
  (let ((j1 (cp:make-connector 'j1 cn))
        (j2 (cp:make-connector 'j2 cn))
        (p1 (cp:make-connector 'p1 cn))
        (p2 (cp:make-connector 'p2 cn))
        (t (cp:make-connector 't cn)))
    (cp:assume-value t #t)
    (cp:make-constraint cp:B=~A #f cn j1 p1)
    (cp:make-constraint cp:B=~A #f cn j2 p2)
    (cp:make-constraint cp:C=AvB #f cn t j1 j2)
    (cp:make-constraint cp:C=AvB #f cn t p1 p2)
    (cp:assume-value j1 #t)
    (write-line (cp:value-of j1))
    (write-line (cp:value-of j2))
    (write-line (cp:value-of p1))
    (write-line (cp:value-of p2))
    ))
#t
#f
#f
#t
;Unspecified return value
```

```
#| -*-Scheme-*-

$Id$

Copyright 2006 Massachusetts Institute of Technology

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or (at
your option) any later version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301,
USA.

|#

;;;; Simple constraint propagator

(declare (usual-integrations))
```

```scheme
(define-record-type <cp:network>
    (cp:%make-network name value-model coincidence-handler
                      post-propagation-hook tms connectors activation-queue)
    cp:network?
  (name cp:network-name)
  (value-model cp:network-value-model)
  (coincidence-handler cp:coincidence-handler)
  (post-propagation-hook cp:post-propagation-hook)
  (tms cp:network-tms)
  (connectors cp:network-connectors cp:set-network-connectors!)
  (activation-queue cp:activation-queue))

(define-guarantee cp:network "constraint network")

(define (cp:make-network name value-model
                         #!optional coincidence-handler post-propagation-hook)
  (guarantee-symbol name 'cp:make-network)
  (guarantee-cp:value-model value-model 'cp:make-network)
  (cp:%make-network name
                    value-model
                    (if (default-object? coincidence-handler)
                        #f
                        coincidence-handler)
                    (if (default-object? post-propagation-hook)
                        #f
                        post-propagation-hook)
                    (tms:make name cp:event-handler)
                    '()
                    (cp:make-queue)))

(define (cp:add-connector-to-network connector network)
  (cp:set-network-connectors! network
                              (cons connector
                                    (cp:network-connectors network))))
```

```
(define-record-type <cp:value-model>
    (cp:%make-value-model membership-predicate
                          equality-predicate
                          specificity-predicate
                          plunk-generator)
    cp:value-model?
  (membership-predicate cp:value-membership-predicate)
  (equality-predicate cp:value-equality-predicate)
  (specificity-predicate cp:value-specificity-predicate)
  (plunk-generator cp:plunk-generator))

(define-guarantee cp:value-model "constraint value model")

(define (cp:make-value-model membership-predicate
                             equality-predicate
                             specificity-predicate
                             #!optional
                             plunk-generator)
  (cp:%make-value-model membership-predicate
                        equality-predicate
                        specificity-predicate
                        (if (default-object? plunk-generator)
                            #f
                            plunk-generator)))

(define (cp:value-acceptable? v connector)
  ((cp:value-membership-predicate (cp:connector-value-model connector))
   v))

(define (cp:values-equal? v1 v2 connector)
  ((cp:value-equality-predicate (cp:connector-value-model connector))
   v1 v2))

(define (cp:value-more-specific-than? v1 v2 connector)
  ((cp:value-specificity-predicate (cp:connector-value-model connector))
   v1 v2))
```

```
;;;; Constraints

(define-record-type <cp:constraint-type>
    (cp:make-constraint-type name connector-names propagator)
    cp:constraint-type?
  (name cp:constraint-type-name)
  (connector-names cp:constraint-type-connector-names)
  (propagator cp:constraint-type-propagator))

(define-guarantee cp:constraint-type "constraint type")

(define-record-type <cp:constraint>
    (cp:%make-constraint type name network connectors)
    cp:constraint?
  (type cp:constraint-type)
  (name cp:constraint-name)
  (network cp:constraint-network)
  (connectors cp:constraint-connectors))

(define-guarantee cp:constraint "constraint")

(define (cp:make-constraint type name network . connectors)
  (let ((n-given (length connectors))
        (n-expected (length (cp:constraint-type-connector-names type))))
    (if (not (= n-given n-expected))
        (error "Wrong number of connectors:" n-given n-expected)))
  (let ((constraint (cp:%make-constraint type name network connectors)))
    (for-each (lambda (connector)
                (cp:add-constraint-to-connector constraint connector))
              connectors)
    (cp:awaken-constraint constraint)
    constraint))

(define (cp:constraint-connector-names constraint)
  (cp:constraint-type-connector-names (cp:constraint-type constraint)))

(define (cp:get-connector constraint name)
  (let loop
      ((names (cp:constraint-connector-names constraint))
       (conns (cp:constraint-connectors constraint)))
    (if (not (pair? names))
        (error:bad-range-argument name 'cp:get-connector))
    (if (eq? (car names) name)
        (car conns)
        (loop (cdr names) (cdr conns)))))
```

```scheme
;;;; Connectors

(define-record-type <cp:connector>
    (cp:%make-connector name network datum value-model constraints assignments)
    cp:connector?
  (name cp:connector-name)
  (network cp:connector-network)
  (datum cp:connector-datum)
  (value-model cp:%connector-value-model)
  (constraints cp:connector-constraints cp:set-connector-constraints!)
  (assignments cp:connector-assignments cp:set-connector-assignments!))

(define-guarantee cp:connector "connector")

(set-record-type-unparser-method! <cp:connector>
  (standard-unparser-method 'connector
    (lambda (connector port)
      (write-char #\space port)
      (write (cp:connector-name connector) port))))

(define (cp:make-connector name network datum #!optional value-model)
  (let ((value-model (if (default-object? value-model) #f value-model)))
    (if value-model
        (guarantee-cp:value-model value-model 'cp:make-connector))
    (let ((connector
           (cp:%make-connector name
                               network
                               datum
                               value-model
                               '()
                               '())))
      (cp:add-connector-to-network connector network)
      connector)))

(define (cp:connector-value-model connector)
  (or (cp:%connector-value-model connector)
      (cp:network-value-model (cp:connector-network connector))))

(define (cp:add-constraint-to-connector constraint connector)
  (cp:set-connector-constraints!
   connector
   (cons constraint (cp:connector-constraints connector))))

(define (cp:add-assignment-to-connector assignment connector)
  (cp:set-connector-assignments!
   connector
   (cons assignment (cp:connector-assignments connector))))

(define (cp:supported-assignments connector)
  (keep-matching-items (cp:connector-assignments connector)
    tms:node-supported?))

(define (cp:supported-assignment connector)
  (let ((assignment
         (find-matching-item (cp:supported-assignments connector)
           (lambda (assignment)
             (not (there-exists? (cp:assignment-specializations assignment)
                    tms:node-supported?))))))
    (if (not assignment)
        (error:bad-range-argument connector 'cp:supported-assignment))
    assignment))

(define (cp:has-value? connector)
  (there-exists? (cp:connector-assignments connector)
```

```
      tms:node-supported?))

  (define (cp:value-of connector)
    (cp:assignment-value (cp:supported-assignment connector)))

  (define (cp:value-is? connector value)
    (tms:node-supported? (cp:assignment-for-value connector value)))

  (define (cp:assignment-for-value connector value)
    (or (find-matching-item (cp:connector-assignments connector)
          (lambda (assignment)
            (cp:values-equal? (cp:assignment-value assignment)
                              value
                              connector)))
        (cp:make-assignment connector value)))
```

```
;;;; Assignments

(define (cp:make-assignment connector value)
  (if (not (cp:value-acceptable? value connector))
      (error:bad-range-argument value 'cp:make-assignment))
  (let ((assignment
         (tms:make-node (cp:network-tms (cp:connector-network connector))
                        (cp:make-assignment-datum connector value))))
    (cp:add-assignment-to-connector assignment connector)
    assignment))

(define (cp:assignment? object)
  (and (tms:node? object)
       (cp:assignment-datum? (tms:node-datum object) )))

(define (cp:assignment-connector assignment)
  (cp:assignment-datum-connector (tms:node-datum assignment)))

(define (cp:assignment-value assignment)
  (cp:assignment-datum-value (tms:node-datum assignment)))

(define-record-type <cp:assignment-datum>
    (cp:make-assignment-datum connector value)
    cp:assignment-datum?
  (connector cp:assignment-datum-connector)
  (value cp:assignment-datum-value))

(define (cp:assume-value connector value)
  (let ((assignment (cp:assignment-for-value connector value)))
    (tms:assume-node assignment)
    (cp:reconcile-assignments connector assignment))
  (cp:propagate (cp:connector-network connector)))

(define (cp:retract-value connector)
  (let loop
      ((assignments (cp:connector-assignments connector))
       (retractions? #f))
    (if (pair? assignments)
        (if (tms:node-assumed? (car assignments))
            (begin
              (tms:retract-node (car assignments))
              (loop (cdr assignments) #t))
            (loop (cdr assignments) retractions?))
        (begin
          (if retractions?
              (cp:propagate (cp:connector-network connector)))
          retractions?))))
```

```
;;;; Assigning values

(define (cp:set-value result value operands constraint)
  (if (cp:has-value? result)
      (cp:check-functional-coincidence result value operands constraint)
      (cp:set-value-1 result value operands constraint)))

(define (cp:check-functional-coincidence result value operands constraint)
  (let ((result-asn (cp:supported-assignment result))
        (operand-asns (map cp:supported-assignment operands)))
    (if (not (or (cp:constraint-already-satisfied?
                  constraint
                  (cons result-asn operand-asns))
                 (cp:values-equal? (cp:assignment-value result-asn)
                                   value
                                   result)))
        ;; Forces contradiction:
        (cp:set-value-1 result value operands constraint))))

(define (cp:constraint-already-satisfied? constraint supports)
  (there-exists? supports
    (lambda (support)
      (there-exists? (tms:node-justifications support)
        (lambda (j)
          (and (eq? (tms:justification-rule j) constraint)
               (tms:justification-supported? j)
               (for-all? (tms:justification-antecedents j)
                 (lambda (node)
                   (memq node supports)))))))))

(define (cp:set-value-1 connector value connectors constraint)
  (let ((assignment (cp:assignment-for-value connector value)))
    (tms:justify-node assignment
                      constraint
                      (map cp:supported-assignment connectors))
    (cp:reconcile-assignments connector assignment)))

(define (cp:reconcile-assignments connector assignment)
  (for-each
   (lambda (other)
     (cond ((eq? other assignment)
            unspecific)
           ((cp:value-more-specific-than? (cp:assignment-value other)
                                          (cp:assignment-value assignment)
                                          connector)
            (tms:justify-node assignment 'more-general (list other)))
           ((cp:value-more-specific-than? (cp:assignment-value assignment)
                                          (cp:assignment-value other)
                                          connector)
            (tms:justify-node other 'more-general (list assignment)))
           ((let ((handler
                   (cp:coincidence-handler (cp:connector-network connector))))
              (and handler
                   (handler assignment other)))
            unspecific)
           (else
            (cp:signal-contradiction (list assignment other) connector))))
   (cp:supported-assignments connector)))

(define (cp:assignment-specializations assignment)
  (map tms:justification-consequent
       (keep-matching-items (tms:node-consequents assignment)
         (lambda (justification)
           (eq? (tms:justification-rule justification) 'more-general)))))
```

```
(define (cp:signal-contradiction assignments connector)
  (tms:justify-node (tms:contradiction
                     (cp:network-tms (cp:connector-network connector)))
                    connector
                    assignments))
```

```scheme
;;;; Propagation

(define (cp:event-handler node justification)
  (if (cp:assignment? node)
      (cp:awaken-connector (cp:assignment-connector node))))

(define (cp:awaken-connector connector)
  (cp:awaken-constraints (cp:connector-constraints connector)))

(define (cp:awaken-constraints constraints)
  (for-each cp:awaken-constraint constraints))

(define (cp:awaken-constraint constraint)
  (cp:enqueue constraint
            (cp:activation-queue (cp:constraint-network constraint))))

(define (cp:propagate network)
  (let ((queue (cp:activation-queue network))
        (contra (tms:contradiction (cp:network-tms network))))
    (let loop ()
      (if (and (tms:node-supported? contra)
               cp:warn-on-contradiction?)
          (warn "Contradiction:" contra))
      (let ((constraint (cp:dequeue queue #f)))
        (if constraint
            (begin
              ;;(write-line `(activate ,constraint))
              ((cp:constraint-type-propagator (cp:constraint-type constraint))
               constraint)
              (loop))
            (let ((hook (cp:post-propagation-hook network)))
              (if hook
                  (begin (hook network)
                         (if (not (cp:empty-queue? queue))
                             (loop)))))))))))

(define cp:warn-on-contradiction? #t)

(define (cp:make-queue)
  (cons '() '()))

(define (cp:enqueue item queue)
  (if (not (memq item (car queue)))     ;don't queue twice
      (let ((old (cdr queue))
            (new (list item)))
        (if (pair? old)
            (set-cdr! old new)
            (set-car! queue new))
        (set-cdr! queue new))))

(define (cp:dequeue queue default)
  (let ((old (car queue)))
    (if (pair? old)
        (begin
          (let ((new (cdr old)))
            (set-car! queue new)
            (if (not (pair? new))
                (set-cdr! queue new)))
          (car old))
        default)))

(define (cp:empty-queue? queue)
  (not (pair? (car queue))))
```

```
#| -*-Scheme-*-

$Id$

Copyright 2006 Massachusetts Institute of Technology

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or (at
your option) any later version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301,
USA.

|#

;;;; Equality constraints make two connectors hold the same values.

(declare (usual-integrations))

(define cp:B=A
  (cp:make-constraint-type 'B=A '(b a)
    (lambda (constraint)
      (let ((b (cp:get-connector constraint 'b))
            (a (cp:get-connector constraint 'a)))
        (cond ((cp:has-value? a)
               (cp:set-value b (cp:value-of a) (list a) constraint))
              ((cp:has-value? b)
               (cp:set-value a (cp:value-of b) (list b) constraint)))))))
```

```
(define *equations* '())

(define (coincidence-handler new-assignment old-assignment)
  (let ((connector (cp:assignment-connector old-assignment)))
    (if (not (eq? (cp:assignment-connector new-assignment) connector))
        (error "Inconsistent connectors: coincidence"
               new-assignment old-assignment))
    (let ((new-value (cp:assignment-value new-assignment))
          (old-value (cp:assignment-value old-assignment)))
      (let ((residual (simplify (- new-value old-value))))
        (if (number? residual)
            (if (= residual 0)
                #t                      ;ignore -- wrong thing?
                #f                      ;contradiction
                )
            (let ((equation
                   (make-equation residual
                                  (list new-assignment old-assignment))))
              (if equation
                  (set! *equations* (cons equation *equations*)))
              #t))))))


;;; expr should have value zero
;;; justs is a list of tms nodes

(define (make-equation expr justs)
  (let* ((specs (standardize-equation expr '() '() #f list))
         (pexpr (car specs))
         (vspecs (cadr specs))
         (tms (tms:node-tms (car justs))))
    (if (and (number? pexpr) (not (= pexpr 0)))
        (begin (tms:justify-node (tms:contradiction tms) 'equation justs)
               (list pexpr justs vspecs))
        (let ((node (tms:make-node tms (cp:make-equation-datum pexpr))))
          (tms:justify-node node 'equation justs)
          (list pexpr (list node) vspecs)))))

(define (equation-supported? equation)
  (for-all? (cadr equation) tms:node-supported?))

(define-record-type <cp:equation-datum>
    (cp:make-equation-datum residual)
   cp:equation-datum?
  (residual cp:equation-datum-residual))

(define (cp:equation? x)
  (and (tms:node? x)
       (cp:equation-datum? (tms:node-datum x))))
```

```
(define (post-propagation-processor network)
  (let* ((results
          (solve-incremental
           (keep-matching-items *equations* equation-supported?)
           (map car
                (keep-matching-items *plunked-variables*
                  (lambda (varpair)
                    (eq? (cp:value-of (cdr varpair)) (car varpair)))))))
         (substitutions (caddr results)))
    (for-each (lambda (subst)
                (let ((var (substitution-variable subst))
                      (expr (substitution-expression subst))
                      (justs (substitution-justifications subst)))
                  (let ((entry (assoc var *plunked-variables*)))
                    (if (not entry)
                        (error "variable appeared?" var))
                    (cp:assign-value (cdr entry) expr
                                     justs 'equation-solver))))
              substitutions)))

;;; cp:assign-value is like cp:set-value, except that the justs are
;;; tms nodes rather than connectors, and the constraint is a symbol.
;;; The following is a dummy.

(define (cp:assign-value connector value justification rule)
  (pp-comment `(,connector <== ,value ,justification ,rule)))
```

```
(define (standardize-equation residual variables functions variable continue)
  ;; continue = (lambda (new-residual new-map functions) ...)
  (let ((redo #f))
    (define (walk-expression expression map functions continue)
      (cond ((pair? expression)
             (let ((rator (operator expression))
                   (rands (operands expression)))
               (cond ((and (= (length rands) 1) (eq? (car rands) variable))
                      (walk-expression rator
                                       (list-adjoin rator map)
                                       (list-adjoin rator functions)
                                       continue))
                     ((or (D? expression) (D2? expression))
                      (continue expression
                                (list-adjoin expression map)
                                (list-adjoin expression functions)))
                     (else
                      (walk-expression rator map functions
                        (lambda (rator-result rator-map rator-functions)
                          (walk-list rands rator-map rator-functions
                                     (lambda (rands-result rands-map rands-functions)
                                       (continue (cons rator-result
                                                       rands-result)
                                                 rands-map
                                                 rands-functions)))))))))
            ((number? expression)
             (continue (if (and (inexact? expression)
                                (< (abs expression) *zero-threshold*))
                           (begin (set! redo #t) 0)
                           expression)
                       map
                       functions))
            ((memq expression '(+ - / * D expt exp sin cos))
             (continue expression map functions))
            (else
             (continue expression
                       (list-adjoin expression map)
                       functions))))
    (define (walk-list elist map functions continue)
      (if (pair? elist)
          (walk-expression (car elist) map functions
            (lambda (car-result car-map car-functions)
              (walk-list (cdr elist) car-map car-functions
                         (lambda (cdr-result cdr-map cdr-functions)
                           (continue (cons car-result cdr-result)
                                     cdr-map
                                     cdr-functions)))))
          (continue elist map functions)))
    (let lp ((residual (simplify residual)))
      (walk-expression (if (quotient? residual)
                           (symb:dividend residual)
                           residual)
                       variables
                       functions
                       (lambda (expression map funs)
                         (if redo
                             (begin (set! redo #f)
                                    (lp (simplify expression)))
                             (continue expression map funs)))))))
```

```
(define *zero-threshold* 1e-15)         ;for small numbers

(define (D? x)
  (and (pair? x)
       (eq? (car x) 'D)))

(define (D2? x)
  (and (pair? x)
       (equal? (car x) '(expt D 2))))
```

```
(declare (usual-integrations))

(define (make-line-prefix-port port prefix)
  (make-port line-prefix-port-type
             (vector port prefix #t)))

(define line-prefix-port-type
  (make-port-type
   `((write-char
      ,(lambda (port char)
         (let ((v (port/state port)))
           (let ((port (vector-ref v 0))
                 (prefix (vector-ref v 1)))
             (if (vector-ref v 2)
                 (write-string prefix port))
             (write-char char port)
             (vector-set! v 2 (char=? char #\newline)))))))
   #f))


(define ((pp-line-prefix prefix) object #!optional port . rest)
  (let ((port
         (make-line-prefix-port
          (if (default-object? port)
              (current-output-port)
              port)
          prefix)))
    (apply pp (cons object (cons port rest)))))


(define pp-comment (pp-line-prefix ";"))

(set! repl:write-result-hash-numbers? #f)
```

```
;;; This is set up to work in scmutils (mechanics) system

(with-working-directory-pathname (directory-pathname (current-load-pathname))
  (lambda ()

    ;; Infrastructure
    (load "tms")
    (load "constraints")

    ;; Common Constraint Types
    (load "equality-constraint")
    (load "boolean-constraints")
    (load "numerical-constraints")

    ;; Output Utilities
    (load "line-prefix")
    (load "proofs")

    ;; Equation handling (still in development)
    ;; solve must be loaded before handler
    (load "solve")
    (load "handler")
    (load "plunker")

    ))
```

```
#| -*-Scheme-*-

$Id$

Copyright 2006 Massachusetts Institute of Technology

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or (at
your option) any later version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301,
USA.

|#
```

```scheme
;;;; Numerical constraint types

;;; Want generic arithmetic to allow symbolic stuff.
;;; (declare (usual-integrations))

(define adder
  (cp:make-constraint-type 's=a+b '(s a b)
    (lambda (constraint)
      (let ((s (cp:get-connector constraint 's))
            (a (cp:get-connector constraint 'a))
            (b (cp:get-connector constraint 'b)))
        (cond ((and (cp:has-value? a)
                    (cp:has-value? b))
               (cp:set-value s
                             (+ (cp:value-of a)
                                (cp:value-of b))
                             (list a b)
                             constraint))
              ((and (cp:has-value? a)
                    (cp:has-value? s))
               (cp:set-value b
                             (- (cp:value-of s)
                                (cp:value-of a))
                             (list s a)
                             constraint))
              ((and (cp:has-value? b)
                    (cp:has-value? s))
               (cp:set-value a
                             (- (cp:value-of s)
                                (cp:value-of b))
                             (list s b)
                             constraint)))))))
```

```
(define multiplier
  (cp:make-constraint-type 'p=a*b '(p a b)
    (lambda (constraint)
      (let ((p (cp:get-connector constraint 'p))
            (a (cp:get-connector constraint 'a))
            (b (cp:get-connector constraint 'b)))
        (cond ((cp:value-is? a 0)
               (cp:set-value p 0 (list a) constraint))
              ((cp:value-is? b 0)
               (cp:set-value p 0 (list b) constraint))
              ((and (cp:has-value? a)
                    (cp:has-value? b))
               (cp:set-value p
                             (* (cp:value-of a)
                                (cp:value-of b))
                             (list a b)
                             constraint))
              ((and (cp:has-value? a)
                    (cp:has-value? p))
               (cp:set-value b
                             (/ (cp:value-of p)
                                (cp:value-of a))
                             (list p a)
                             constraint))
              ((and (cp:has-value? b)
                    (cp:has-value? p))
               (cp:set-value a
                             (/ (cp:value-of p)
                                (cp:value-of b))
                             (list p b)
                             constraint))))))
```

```
;;; Celsius-Farenheit conversion example

(define network
  (cp:make-network 'network
                   (cp:make-value-model numerical-quantity?
                                        =
                                        (lambda (x y) #f))))

(define c (cp:make-connector 'c network #f))
(define f (cp:make-connector 'f network #f))
(define u (cp:make-connector 'u network #f))
(define v (cp:make-connector 'v network #f))

(define nine (cp:make-connector 'nine network #f))
(define five (cp:make-connector 'five network #f))
(define thirty-two (cp:make-connector 'thirty-two network #f))

(cp:assume-value nine 9)
(cp:assume-value five 5)
(cp:assume-value thirty-two 32)

(cp:make-constraint multiplier #f network u c nine)
(cp:make-constraint multiplier #f network u v five)
(cp:make-constraint adder #f network f v thirty-two)

(cp:assume-value c 100)
;Unspecified return value

(cp:has-value? f)
;Value: #t

(cp:value-of f)
;Value: 212

(cp:retract-value c)
;Value: #t

(cp:has-value? f)
;Value: #f

(cp:assume-value c -40)
;Unspecified return value

(cp:has-value? f)
;Value: #t

(cp:value-of f)
;Value: -40

(cp:retract-value c)
;Value: #t

(cp:assume-value f 32)
;Unspecified return value

(cp:has-value? c)
;Value: #t

(cp:value-of c)
;Value: 0

(cp:why? c)
;(n19 (c = 0) (p=a*b n17 n18))
;(n17 (u = 0) (p=a*b n20))
;(n18 (nine = 9) (premise))
```

```
;(n20 (v = 0) (s=a+b n21 n22))
;(n21 (f = 32) (premise))
;(n22 (thirty-two = 32) (premise))
;Unspecified return value

(cp:retract-value f)
;Value: #t

(cp:assume-value f 98.6)
;Unspecified return value

(cp:has-value? c)
;Value: #t

(cp:value-of c)
;Value: 37.

(cp:why? c)
;(n25 (c = 37.) (p=a*b n24 n18))
;(n24 (u = 333.) (p=a*b n26 n27))
;(n18 (nine = 9) (premise))
;(n26 (v = 66.6) (s=a+b n28 n22))
;(n27 (five = 5) (premise))
;(n28 (f = 98.6) (premise))
;(n22 (thirty-two = 32) (premise))
;Unspecified return value
```

```
;;; Simple Electrical Circuit : source and resistor

(define ckt1
  (cp:make-network 'ckt1
                    (cp:make-value-model numerical-quantity?
                                          =
                                          (lambda (x y) #f))))

;;; terminal potentials

(define es1 (cp:make-connector 'es1 ckt1 #f))
(define es2 (cp:make-connector 'es2 ckt1 #f))

(define er1 (cp:make-connector 'er1 ckt1 #f))
(define er2 (cp:make-connector 'er2 ckt1 #f))


;;; terminal currents

(define is1 (cp:make-connector 'is1 ckt1 #f))
(define is2 (cp:make-connector 'is2 ckt1 #f))

(define ir1 (cp:make-connector 'ir1 ckt1 #f))
(define ir2 (cp:make-connector 'ir2 ckt1 #f))


;;; branch voltages

(define vs (cp:make-connector 'vs ckt1 #f))
(define vr (cp:make-connector 'vr ckt1 #f))


;;; parameters

(define vss (cp:make-connector 'vss ckt1 #f))
(define r (cp:make-connector 'r ckt1 #f))

;;; utility
(define zero-i (cp:make-connector 'zero-i ckt1 #f))
(cp:assume-value zero-i 0)


;;; KVL
;;;   Nodes
(cp:make-constraint cp:B=A 'KVL-N ckt1 es1 er1)
(cp:make-constraint cp:B=A 'KVL-N ckt1 es2 er2)

;;;   Elements
(cp:make-constraint adder 'KVL-E ckt1 es1 vs es2)
(cp:make-constraint adder 'KVL-E ckt1 er1 vr er2)


;;; KCL
;;;   Nodes
(cp:make-constraint adder 'KCL-N ckt1 zero-i is1 ir1)
(cp:make-constraint adder 'KCL-N ckt1 zero-i is2 ir2)

;;;   Elements
(cp:make-constraint adder 'KCL-E ckt1 zero-i is1 is2)
(cp:make-constraint adder 'KCL-E ckt1 zero-i ir1 ir2)
```

```
;;; VIC
;;;   Voltage-source
(cp:make-constraint cp:B=A 'Voltage-source ckt1 vss vs)

;;;   Resistor
(cp:make-constraint multiplier 'OHM ckt1 vr ir1 r)

;;; Specific situation
(cp:assume-value vss 12)
(cp:assume-value r 6)

(cp:has-value? ir1)
;Value: #f

;;; Declare ground.
(cp:assume-value es1 0)

(cp:has-value? ir1)
;Value: #t

(cp:value-of ir1)
;Value: 2

;;; Correct!

(cp:retract-value r)
;Value: #t

(cp:assume-value is2 3)

(cp:value-of r)
;Value: 4

;;; Yup...

;;; BTW: cp:B=A is defined in boolean-constraints, but it is used here
;;; for numerical constraints.  Equality is more general and need not
;;; be restricted to booleans.

;;; Ground potential should be a CP hypothetical...
;;; Note: algebra works under scmutils.

(cp:retract-value es1)
;Value: #t

(cp:assume-value es1 'E)

(cp:why? r)
;(n117 (r = 4) (OHM n115 n116))
;(n115 (vr = 12) (KVL-E n118 n119))
;(n116 (ir1 = 3) (KCL-N n120 n121))
;(n118 (er1 = E) (KVL-N n122))
;(n119 (er2 = (+ -12 E)) (KVL-N n123))
;(n120 (zero-i = 0) (premise))
;(n121 (is1 = -3) (KCL-E n120 n124))
;(n122 (es1 = E) (premise))
;(n123 (es2 = (+ -12 E)) (KVL-E n122 n125))
;(n124 (is2 = 3) (premise))
;(n125 (vs = 12) (Voltage-source n126))
;(n126 (vss = 12) (premise))
;Unspecified return value

;;; If an expression derived from a variable plunk premise
;;; does not contain the plunk variable, we may conclude
```

```
;;; that the assumption of the plunk variable is not
;;; an essential ingredient of that expression and edit the
;;; dependencies to eliminate the plunk premise.


;;; Now make a contradiction.

(cp:retract-value es1)
;Value: #t

(cp:has-value? r)
;Value: #f

(cp:assume-value r 6)
;Unspecified return value

(cp:assume-value es2 0)
;Contradiction: #[tms:node 85]
;;; Error signalled here.

;;; Contradictions are not SCHEME errors.
;;; They are deductions of falshoods...
;;; They should not stop the process!
;;; They should report, if monitoring,
;;; enqueue the contradiction handler,
;;; and continue.  Otherwise the system
;;; may get inconsistent.

(cp:explain-contradiction ckt1)
;(n127 contradiction (er1 n128 n129))
;(n128 (er1 = 18) (KVL-E n130 n131))
;(n129 (er1 = 12) (KVL-N n132))
;(n130 (vr = 18) (OHM n116 n133))
;(n131 (er2 = 0) (KVL-N n134))
;(n132 (es1 = 12) (KVL-E n125 n134))
;(n116 (ir1 = 3) (KCL-N n120 n121))
;(n133 (r = 6) (premise))
;(n134 (es2 = 0) (premise))
;(n125 (vs = 12) (Voltage-source n126))
;(n120 (zero-i = 0) (premise))
;(n121 (is1 = -3) (KCL-E n120 n124))
;(n126 (vss = 12) (premise))
;(n124 (is2 = 3) (premise))
;Unspecified return value
```

```
;;; Still under development: solve, handler, plunker.

;;; Simple Electrical Circuit : source and voltage divider

;;; These will disappear soon.
(set! *equations* '())
(set! *plunked-variables* '())

(define (numerical-quantities-equal? x y)
  (if (and (number? x) (number? y))
      (= x y)
      (zero? (simplify (- x y)))))

(define ckt2
  (cp:make-network 'ckt2
                   (cp:make-value-model numerical-quantity?
                                        numerical-quantities-equal?
                                        (lambda (x y) #f))
                   coincidence-handler
                   post-propagation-processor))

;;; terminal potentials

(define esh (cp:make-connector 'esh ckt2 #f))
(define esl (cp:make-connector 'esl ckt2 #f))

(define er1h (cp:make-connector 'er1h ckt2 #f))
(define er1l (cp:make-connector 'er1l ckt2 #f))

(define er2h (cp:make-connector 'er2h ckt2 #f))
(define er2l (cp:make-connector 'er2l ckt2 #f))


;;; terminal currents

(define ish (cp:make-connector 'ish ckt2 #f))
(define isl (cp:make-connector 'isl ckt2 #f))

(define ir1h (cp:make-connector 'ir1h ckt2 #f))
(define ir1l (cp:make-connector 'ir1l ckt2 #f))

(define ir2h (cp:make-connector 'ir2h ckt2 #f))
(define ir2l (cp:make-connector 'ir2l ckt2 #f))


;;; branch voltages

(define vs (cp:make-connector 'vs ckt2 #f))
(define vr1 (cp:make-connector 'vr1 ckt2 #f))
(define vr2 (cp:make-connector 'vr2 ckt2 #f))


;;; parameters

(define vss (cp:make-connector 'vss ckt2 #f))
(define r1 (cp:make-connector 'r1 ckt2 #f))
(define r2 (cp:make-connector 'r2 ckt2 #f))

;;; utility
(define zero-i (cp:make-connector 'zero-i ckt2 #f))
(cp:assume-value zero-i 0)


;;; KVL
```

```
  ;;;   Nodes
  (cp:make-constraint cp:B=A 'KVL-N ckt2 esh er1h)
  (cp:make-constraint cp:B=A 'KVL-N ckt2 er1l er2h)
  (cp:make-constraint cp:B=A 'KVL-N ckt2 esl er2l)

  ;;;   Elements
  (cp:make-constraint adder 'KVL-E ckt2 esh vs esl)
  (cp:make-constraint adder 'KVL-E ckt2 er1h vr1 er1l)
  (cp:make-constraint adder 'KVL-E ckt2 er2h vr2 er2l)


  ;;; KCL
  ;;;   Nodes
  (cp:make-constraint adder 'KCL-N ckt2 zero-i ish ir1h)
  (cp:make-constraint adder 'KCL-N ckt2 zero-i ir1l ir2h)
  (cp:make-constraint adder 'KCL-N ckt2 zero-i isl ir2l)

  ;;;   Elements
  (cp:make-constraint adder 'KCL-E ckt2 zero-i ish isl)
  (cp:make-constraint adder 'KCL-E ckt2 zero-i ir1h ir1l)
  (cp:make-constraint adder 'KCL-E ckt2 zero-i ir2h ir2l)
```

```
;;; VIC
;;;   Voltage-source
(cp:make-constraint cp:B=A 'Voltage-source ckt2 vss vs)

;;;   Resistor
(cp:make-constraint multiplier 'OHM ckt2 vr1 ir1h r1)
(cp:make-constraint multiplier 'OHM ckt2 vr2 ir1h r2)

;;; Specific situation
(cp:assume-value vss 12)
(cp:assume-value r1 9)
(cp:assume-value r2 3)


;;; Declare ground.
(cp:assume-value esl 0)

(cp:has-value? er2h)
;Value: #f

(cp:plunk-connector er2h)
;(#[connector 196 er2h] <== 3 (#[tms:node 195]) equation-solver)

(pp-comment *equations*)
;(((+ 12 (* -4 x-er2h)) (#[tms:node 195]) (x-er2h))
; ((+ -12 (* 4 x-er2h)) (#[tms:node 199]) (x-er2h))
; ((+ -12 (* 4 x-er2h)) (#[tms:node 198]) (x-er2h))
; ((+ -12 (* 4 x-er2h)) (#[tms:node 197]) (x-er2h)))


;;; These are the right equations, but
;;; why are there 4 equivalent equations?

(cp:explain-node (unhash 195))
;(n195 (+ 12 (* -4 x-er2h)) (equation n214 n215))
;(n214 (esh = 12) (KVL-E n216 n217))
;(n215 (esh = (* 4 x-er2h)) (KVL-N n218))
;(n216 (vs = 12) (Voltage-source n219))
;(n217 (esl = 0) (premise))
;(n218 (er1h = (* 4 x-er2h)) (KVL-E n220 n221))
;(n219 (vss = 12) (premise))
;(n220 (vr1 = (* 3 x-er2h)) (OHM n222 n223))
;(n221 (er1l = x-er2h) (KVL-N n224))
;(n222 (ir1h = (* 1/3 x-er2h)) (OHM n225 n226))
;(n223 (r1 = 9) (premise))
;(n224 (er2h = x-er2h) (premise))
;(n225 (vr2 = x-er2h) (KVL-E n224 n227))
;(n226 (r2 = 3) (premise))
;(n227 (er2l = 0) (KVL-N n217))
;Unspecified return value

;;; Note: n224 is a premise that must be dismissed when the value from
;;; the equation solver is asserted at connector er2h.
```

```
;;; The value model of a connector must supply a plunk-generator
;;;   if there are units, for example.

(define (cp:plunk-network network #!optional chooser)
  (if (default-object? chooser) (set! chooser car))
  (let* ((unassigned-connectors
          (keep-matching-items (cp:network-connectors network)
            (lambda (conn) (not (cp:has-value? conn)))))
         (chosen-connector (chooser unassigned-connectors)))
    (cp:plunk-connector connector)))

(define (cp:plunk-connector connector)
  (let ((new-variable
         ((or (cp:plunk-generator
               (cp:connector-value-model connector))
              (lambda (name)
                (symbol "x-" name)))
          (cp:connector-name connector))))
    (set! *plunked-variables*
          (cons (cons new-variable connector)
                *plunked-variables*))
    (cp:assume-value connector new-variable)))

(define *plunked-variables* '())

#|
;;; In scmutils, after building c<-->f converter

(cp:plunk-connector c)

(pe (cp:value-of c))
c4

(pe (cp:value-of f))
(+ 32 (* 9/5 c4))
|#
```

```
#| -*-Scheme-*-

$Id$

Copyright 2006 Massachusetts Institute of Technology

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or (at
your option) any later version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301,
USA.

|#

;;; Explains why a connector has a particular assignment.

(define (cp:why? connector #!optional port)
  (let ((assignment
         (find-matching-item (cp:connector-assignments connector)
           tms:node-supported?)))
    (if (not assignment)
        (display "; The connector is not assigned.\n" port)
        (cp:explain-node assignment port))))

(define (cp:explain-contradiction network #!optional port)
  (let ((node (tms:contradiction (cp:network-tms network))))
    (if (tms:node-supported? node)
        (cp:explain-node node)
        (display "; There is no contradiction outstanding.\n" port))))

(define (cp:explain-node node #!optional port)
  (tms:walk-node-support node
     (lambda (consequent just antecedents)
       (cond ((cp:assignment? consequent)
              (let ((conn (cp:assignment-connector consequent)))
                (pp-comment
                 `(,(tms:node-id consequent)
                   ,(list (cp:connector-name conn)
                          '=
                          (simplify (cp:assignment-value consequent)))
                   ,(cond ((tms:premise-justification? just) '(premise))
                          (else
                           `(,(cp:just-id just)
                             ,@(map tms:node-id antecedents)))))
                port))
             ((cp:equation? consequent)
              (pp-comment
               `(,(tms:node-id consequent)
                 ,(cp:equation-datum-residual (tms:node-datum consequent))
                 ,(cond ((tms:premise-justification? just) '(premise))
                        (else
                         `(,(cp:just-id just)
                           ,@(map tms:node-id antecedents)))))))
             (else
              (pp-comment
               `(,(tms:node-id consequent)
```

```
                    ,(tms:node-datum consequent)
                    ,(cond ((tms:premise-justification? just) '(premise))
                           (else
                            `(,(cp:just-id just)
                              ,@(map tms:node-id antecedents)))))))))))

  (define (tms:node-id node)
    (symbol 'n (object-hash node)))

  (define (cp:just-id just)
    (let ((rule (tms:justification-rule just)))
      (cond ((cp:constraint? rule)
             (let ((name (cp:constraint-name rule)))
               (or name
                   (cp:constraint-type-name
                    (cp:constraint-type rule)))))
            ((cp:connector? rule)
             (cp:connector-name rule))
            (else
             (or rule just)))))
```

```
#| -*-Scheme-*-

$Id$

Copyright 2006 Massachusetts Institute of Technology

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or (at
your option) any later version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301,
USA.

|#

;;; Must be loaded before handler.scm if used in constraint propagator.
```

```
;;;; Simple catch-as-catch-can solver #42.
;;;  By GJS, November 2003.  For use with SCMUTILS.

(declare (usual-integrations))

;;; Assumes standardized equations, as produced below.

#|
(define (solve-incremental equations variables) ...)

;;; Variables list gives variables to be eliminated in the preferred
;;; order of elimination.

;;; To access the parts of a solution returned by solve-incremental

(define (residual-equations solution) ...)
(define (residual-variables solution) ...)
(define (substitutions solution) ...)
(define (hopeless-variables solution) ...)


;;; To access substitutions produced by solve-incremental

(define (substitution-variable subst) ...)
(define (substitution-expression subst) ...)
(define (substitution-justifications subst) ...)


;;; To access residual equations of a solution

(define (equation-expression eqn) ...)
(define (equation-justifications eqn) ...)
|#
```

```
(define (residual-equations solution) (car solution))
(define (residual-variables solution) (cadr solution))
(define (substitutions solution) (caddr solution))
(define (hopeless-variables solution) (cadddr solution))

(define (solve-incremental equations variables)
  (let lp ((residual-eqs      equations)
           (residual-vars     variables)
           (substitutions     '())
           (hopeless-vars     '()))
    (if (or (null? residual-vars) (null? residual-eqs)) ;we are done.
        (list residual-eqs residual-vars substitutions hopeless-vars)
        (isolate-var (car residual-vars)
                     (sort residual-eqs
                           (lambda (eqn1 eqn2)
                             (< (length (equation-variables eqn1))
                                (length (equation-variables eqn2)))))
                     (lambda (new-substitution equation-used)
                       (lp (filter (lambda (eqn) ;flush tautologies
                                     (let ((expr (equation-expression eqn)))
                                       (not (and (number? expr) (= expr 0)))))
                                   (map (lambda (equation)
                                          (backsubstitute-equation new-substitution
                                                                   equation))
                                        (delete equation-used residual-eqs)))
                           (cdr residual-vars)
                           (cons new-substitution
                                 (map (lambda (substitution)
                                        (backsubstitute new-substitution
                                                        substitution))
                                      substitutions))
                           hopeless-vars))
                     (lambda ()         ; fail
                       (lp residual-eqs
                           (cdr residual-vars)
                           substitutions
                           (cons (car residual-vars) hopeless-vars)))))))

(define (isolate-var var eqs succeed fail)
  ;; succeed = (lambda (new-substitution equation-used) ...)
  ;; fail    = (lambda () ...)
  (let lp ((eqs-to-scan eqs))
    (cond ((null? eqs-to-scan) (fail))
          ((member var (equation-variables (car eqs-to-scan)))
           (isolatable? var (car eqs-to-scan)
                        (lambda (value)
                          (succeed (make-substitution var value
                                     (equation-justifications (car eqs-to-scan)))
                                   (car eqs-to-scan)))
                        (lambda ()
                          (lp (cdr eqs-to-scan)))))
          (else (fail)))))
```

```
(define (isolatable? var eqn succeed fail)
  (let lp ((expr (equation-expression eqn)))
    (cond ((equal? var expr) (succeed 0))
          ((and (expt? expr)
                (number? (cadr (operands expr)))
                (> (cadr operands) 0))
           (lp (car (operands expr))))
          ((product? expr)
           ;; If var^n in operands then could be zero.
           (let lp ((factors (operands expr)))
             (cond ((null? factors) (fail))
                   ((symbol? (car factors))
                    (if (eq? var (car factors))
                        (succeed 0)
                        (lp (cdr factors))))
                   ((and (expt? (car factors))
                         (number? (cadr (operands (car factors))))
                         (> (cadr (car factors)) 0)
                         (eq? var (caddr (operands (car factors)))))
                    (succeed 0))
                   (else (lp (cdr factors))))))
          ((sum? expr)
           ;; Split operands into ones with var^1 and without...
           (let lp ((addends (operands expr)) (with '()) (without '()))
             (cond ((null? addends)
                    (if (null? without)
                        (succeed 0)
                        (succeed
                         (symb:quo (symb:negate (symb:add:n without))
                                   (symb:add:n with)))))
                   ((occurs? var (car addends))
                    (let ((addend (car addends)))
                      (cond ((equal? var addend)
                             (lp (cdr addends) (cons 1 with) without))
                            ((product? addend)
                             (let ((factors (operands addend)))
                               (if (member var factors)
                                   (lp (cdr addends)
                                       (cons (symb:mul:n (delete var factors))
                                             with)
                                       without)
                                   (fail)))) ;occurs more painfully
                            (else (fail)))))
                   (else
                    (lp (cdr addends) with (cons (car addends) without))))))
          (else (fail)))))

(define (occurs? var expr)
  (or (equal? var expr)
      (and (pair? expr)
           (or (occurs? var (car expr))
               (occurs? var (cdr expr))))))
```

```
(define (make-substitution var value justs)
  (list (list '= var (simplify value)) justs))

(define (substitution-variable subst) (lhs (car subst)))
(define (substitution-expression subst) (rhs (car subst)))
(define (substitution-justifications subst) (cadr subst))

#|
;;; This is in handler.scm -- to interface with tms and constraint-propagator

(define (make-equation expr justs)
  (let* ((specs (standardize-equation expr '() '() #f list))
         (pexpr (car specs))
         (vspecs (cadr specs))
         (tms (tms:node-tms (car justs))))
    (if (and (number? pexpr) (not (= pexpr 0)))
        (begin (tms:justify-node (tms:contradiction tms) 'equation justs)
               (list pexpr justs vspecs))
        (let ((node (tms:make-node tms (cp:make-equation-datum pexpr))))
          (tms:justify-node node 'equation justs)
          (list pexpr (list node) vspecs)))))

(define-record-type <cp:equation-datum>
    (cp:make-equation-datum residual)
    cp:equation-datum?
  (residual cp:equation-datum-residual))

(define (cp:equation? x)
  (and (tms:node? x)
       (cp:equation-datum? (tms:node-datum x))))
|#

;;; For use or testing without tms or constraint propagator

(define (make-equation expr justs)
  (let* ((specs (standardize-equation expr '() '() #f list))
         (pexpr (car specs))
         (vspecs (cadr specs)))
    (if (and (number? pexpr) (not (= pexpr 0)))
        (begin (write-line `(contradiction ,justs))
               (list pexpr justs vspecs))
        (list pexpr justs vspecs))))


(define (lhs exp)
  (cadr exp))

(define (rhs exp)
  (caddr exp))


(define (equation-expression eqn) (car eqn))
(define (equation-justifications eqn) (cadr eqn))
(define (equation-variables eqn) (caddr eqn))
```

```
(define (backsubstitute new-substitution substitution)
  (if (occurs? (substitution-variable new-substitution)
               (substitution-expression substitution))
      (make-substitution
       (substitution-variable substitution)
       (substitute (substitution-expression new-substitution)
                   (substitution-variable new-substitution)
                   (substitution-expression substitution))
       (list-union (substitution-justifications substitution)
                   (substitution-justifications new-substitution)))
      substitution))

(define (backsubstitute-equation substitution equation)
  (if (occurs? (substitution-variable substitution)
               (equation-expression equation))
      (make-equation
       (substitute (substitution-expression substitution)
                   (substitution-variable substitution)
                   (equation-expression equation))
       (list-union (equation-justifications equation)
                   (substitution-justifications substitution)))
      equation))
```

```
#|
;;; In handler
(define (standardize-equation residual variables functions variable continue)
  ;; continue = (lambda (new-residual new-map functions) ...)
  (let ((redo #f))
    (define (walk-expression expression map functions continue)
      (cond ((pair? expression)
             (let ((rator (operator expression))
                   (rands (operands expression)))
               (cond ((and (= (length rands) 1) (eq? (car rands) variable))
                      (walk-expression rator
                                       (list-adjoin rator map)
                                       (list-adjoin rator functions)
                                       continue))
                     ((or (D? expression) (D2? expression))
                      (continue expression
                                (list-adjoin expression map)
                                (list-adjoin expression functions)))
                     (else
                      (walk-expression rator map functions
                        (lambda (rator-result rator-map rator-functions)
                          (walk-list rands rator-map rator-functions
                                     (lambda (rands-result rands-map rands-functions)
                                       (continue (cons rator-result
                                                       rands-result)
                                                 rands-map
                                                 rands-functions))))))))
            ((number? expression)
             (continue (if (and (inexact? expression)
                                (< (abs expression) *zero-threshold*))
                           (begin (set! redo #t) 0)
                           expression)
                       map
                       functions))
            ((memq expression '(+ - / * D expt exp sin cos))
             (continue expression map functions))
            (else
             (continue expression
                       (list-adjoin expression map)
                       functions))))
    (define (walk-list elist map functions continue)
      (if (pair? elist)
          (walk-expression (car elist) map functions
            (lambda (car-result car-map car-functions)
              (walk-list (cdr elist) car-map car-functions
                         (lambda (cdr-result cdr-map cdr-functions)
                           (continue (cons car-result cdr-result)
                                     cdr-map
                                     cdr-functions)))))
          (continue elist map functions)))
    (let lp ((residual (simplify residual)))
      (walk-expression (if (quotient? residual)
                           (symb:dividend residual)
                           residual)
                       variables
                       functions
                       (lambda (expression map funs)
                         (if redo
                             (begin (set! redo #f)
                                    (lp (simplify expression)))
                             (continue expression map funs)))))))
```

```
(define *zero-threshold* 1e-15)         ;for small numbers

;;; Derivative expressions

(define (D? x)
  (and (pair? x)
       (eq? (car x) 'D)))

(define (D2? x)
  (and (pair? x)
       (equal? (car x) '(expt D 2))))
|#
```

```
#|
;;; Shows signs of life.

(pp (solve-incremental
     (list (make-equation '(+ (* 3 x)     y  -7)  (list 'A))
           (make-equation '(+ (* 3 x) (- y)  -5)  (list 'B)))
     '(x y)))
(() () (((= y 1) (B A)) ((= x 2) (B A))) ())


(pp (solve-incremental
     (list (make-equation '(+  x   y   z  1)  (list 'A))
           (make-equation '(+  x   y      2)  (list 'B))
           (make-equation '(+  x          1)  (list 'C)))
     '(x y z)))
(() () (((= z 1) (A B C)) ((= y -1) (B C)) ((= x -1) (C))) ())


(pp (solve-incremental
     (list (make-equation '(+  x          1)  (list 'C))
           (make-equation '(+  x   y      2)  (list 'B))
           (make-equation '(+  x   y   z  1)  (list 'A)))
     '(x y z)))
(() () (((= z 1) (A B C)) ((= y -1) (B C)) ((= x -1) (C))) ())

;;; The following signals a contradiction, as it should:

(pp (solve-incremental
     (list (make-equation '(+ (* 3 x)     y  -7)  (list 'A))
           (make-equation '(+ (* 3 x)     y  -5)  (list 'B)))
     '(x y)))
(contradiction (B A))
(((2 (B A))) () (((= x (+ 7/3 (* -1/3 y))) (A))) (y))


;;; Some slightly nonlinear systems can be solved:

(pp (solve-incremental
     (list (make-equation '(-  3 (+ x y))  (list 'A))
           (make-equation '(-  5 (- x y))  (list 'B))
           (make-equation '(-  3 (+ (* (sqrt x) z) (square y)))  (list 'C)))
     '(x y z)))
(() () (((= z 1) (C B A)) ((= y -1) (B A)) ((= x 4) (B A))) ())
```

```
;;; Underdetermined systems can be reduced:

(pp (solve-incremental
     (list (make-equation '(+ (* (+ a b) (- a c)) c)  (list 'A))
           (make-equation '(- 3 (+ a b))  (list 'B)))
     '(a b c)))
(() (c) (((= b (+ 3 (* -2/3 c))) (A B)) ((= a (* 2/3 c)) (A B))) ())


(pp (solve-incremental
     (list (make-equation '(+ (* (+ a b) (- a c)) c)  (list 'A))
           (make-equation '(- 3 (- a c))  (list 'B)))
     '(a b c)))
(() (c) (((= b (+ -3 (* -4/3 c))) (A B)) ((= a (+ 3 c)) (B))) ())


;;; Even very hard ones are clarified.

(pp (solve-incremental
     (list (make-equation '(+ (* (+ a b) (- a c)) c)  (list 'A))
           (make-equation '(- 3 (- a b))  (list 'B)))
     '(a b c)))
(()
 ()
 (((= c (/ (+ 9/2 (expt b 2) (* 9/2 b)) (+ 1 b))) (A B))
  ((= a (+ 3 b)) (B)))
 (b))
|#
```

```
#| -*-Scheme-*-

$Id$

Copyright 2006 Massachusetts Institute of Technology

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or (at
your option) any later version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301,
USA.

|#

;;;; Simple truth-maintenance system

(declare (usual-integrations))

(define-record-type <tms>
    (tms:%make name event-handler contradiction)
    tms?
  (name tms:name)
  (event-handler tms:event-handler)
  (contradiction tms:contradiction tms:set-contradiction!))

(define (tms:make name event-handler)
  (let ((tms (tms:%make name event-handler #f)))
    (tms:set-contradiction! tms (tms:make-node tms 'contradiction))
    tms))

(define (tms:signal-event node justification)
  ((tms:event-handler (tms:node-tms node)) node justification))

(define-record-type <tms:node>
    (tms:%make-node tms datum supported? justifications consequents)
    tms:node?
  (tms tms:node-tms)
  (datum tms:node-datum)
  (supported? tms:node-supported? tms:set-node-supported?!)
  (justifications tms:node-justifications tms:set-node-justifications!)
  (consequents tms:node-consequents tms:set-node-consequents!))

(define (tms:make-node tms datum)
  (tms:%make-node tms datum #f '() '()))

(define (tms:node-is-contradiction? node)
  (eq? node (tms:contradiction (tms:node-tms node))))
```

```
;;;; Justifications

(define (unary-generic default-method)
  (let ((generic (make-generic-procedure 1)))
    (set-generic-procedure-default-generator!
     generic
     (lambda (procedure tags) default-method))
    generic))

(define (define-unary-record-method generic record-type method)
  (add-generic-procedure-generator
   generic
   (let ((record-tag (record-type-dispatch-tag record-type)))
     (lambda (procedure tags)
       (and (eq? (car tags) record-tag)
            method)))))

(define tms:justification?
  (unary-generic
   (lambda (object) #f)))

(define tms:justification-consequent
  (unary-generic
   (lambda (object)
     (tms:not-justification object 'tms:justification-consequent))))

(define tms:justification-rule
  (unary-generic
   (lambda (object)
     (tms:not-justification object 'tms:justification-rule))))

(define tms:justification-antecedents
  (unary-generic
   (lambda (object)
     (tms:not-justification object 'tms:justification-antecedents))))

(define tms:justification-supported?
  (unary-generic
   (lambda (object)
     (tms:not-justification object 'tms:justification-supported?))))

(define (tms:not-justification object caller)
  (error:wrong-type-argument object "TMS justification" caller))

(define (tms:attach-justification just)
  (let ((node (tms:justification-consequent just)))
    (for-each (lambda (node)
                (let ((consequents (tms:node-consequents node)))
                  (if (not (memq node consequents))
                      (tms:set-node-consequents! node
                                                 (cons just consequents)))))
              (tms:justification-antecedents just))
    (tms:set-node-justifications!
     node
     (cons just (tms:node-justifications node)))
    (if (tms:justification-supported? just)
        (tms:support-node node just))))
```

```
;;;; Justification of premises

(define-record-type <tms:premise-justification>
    (tms:make-premise-justification consequent supported?)
    tms:premise-justification?
  (consequent tms:premise-justification-consequent)
  (supported? tms:premise-justification-supported?
              tms:set-premise-justification-supported?!))

(define (tms:new-premise node)
  (let ((just (tms:make-premise-justification node #t)))
    (tms:attach-justification just)
    just))

(define (tms:premise-justification node)
  (find-matching-item (tms:node-justifications node)
    tms:premise-justification?))

(define-unary-record-method tms:justification?
    <tms:premise-justification>
  (lambda (object) #t))

(define-unary-record-method tms:justification-consequent
    <tms:premise-justification>
  tms:premise-justification-consequent)

(define-unary-record-method tms:justification-rule
    <tms:premise-justification>
  (lambda (just) #f))

(define-unary-record-method tms:justification-antecedents
    <tms:premise-justification>
  (lambda (just) '()))

(define-unary-record-method tms:justification-supported?
    <tms:premise-justification>
  tms:premise-justification-supported?)

;;;; Justification by antecedents

(define-record-type <tms:conjunctive-justification>
    (tms:make-conjunctive-justification consequent rule antecedents)
    tms:conjunctive-justification?
  (consequent tms:conjunctive-justification-consequent)
  (rule tms:conjunctive-justification-rule)
  (antecedents tms:conjunctive-justification-antecedents))

(define (tms:justify-node consequent rule antecedents)
  (tms:attach-justification
   (tms:make-conjunctive-justification consequent rule antecedents)))

(define-unary-record-method tms:justification?
    <tms:conjunctive-justification>
  (lambda (object) #t))

(define-unary-record-method tms:justification-consequent
    <tms:conjunctive-justification>
  tms:conjunctive-justification-consequent)

(define-unary-record-method tms:justification-rule
    <tms:conjunctive-justification>
  tms:conjunctive-justification-rule)

(define-unary-record-method tms:justification-antecedents
```

```
      <tms:conjunctive-justification>
    tms:conjunctive-justification-antecedents)

  (define-unary-record-method tms:justification-supported?
      <tms:conjunctive-justification>
    (lambda (just)
      (for-all? (tms:conjunctive-justification-antecedents just)
        tms:node-supported?)))
```

```
;;;; Adding support

(define (tms:node-assumed? node)
  (let ((just (tms:premise-justification node)))
    (and just
         (tms:premise-justification-supported? just))))

(define (tms:assume-node node)
  (tms:support-node
   node
   (let ((just (tms:premise-justification node)))
     (if just
         (begin
           (tms:set-premise-justification-supported?! just #t)
           just)
         (tms:new-premise node)))))

(define (tms:support-node node justification)
  (if (not (tms:node-supported? node))
      (begin
        (tms:set-node-supported?! node #t)
        (tms:signal-event node justification)
        (for-each (lambda (just)
                    (if (tms:justification-supported? just)
                        (tms:support-node (tms:justification-consequent just)
                                          just)))
                  (tms:node-consequents node)))))
```

```
;;;; Removing support

(define (tms:retract-node node)
  (let ((just (tms:premise-justification node)))
    (if (not just)
        (error "Node not assumed, can't retract:" node))
    (tms:set-premise-justification-supported?! just #f)
    (if (tms:node-supported? node)
        (begin
          (tms:set-node-supported?! node #f)
          (tms:signal-event node #f)
          (tms:re-support-nodes (tms:propagate-node-retractions node))))))

(define (tms:propagate-node-retractions node)
  (append-map (lambda (just)
                (if (tms:justification-supported? just)
                    '()
                    (tms:propagate-just-retractions just)))
              (tms:node-consequents node)))

(define (tms:propagate-just-retractions just)
  (let ((node (tms:justification-consequent just)))
    (if (tms:node-supported? node)
        (begin
          (tms:set-node-supported?! node #f)
          (cons node (tms:propagate-node-retractions node)))
        '())))

(define (tms:re-support-nodes retracted)
  (let loop ((nodes retracted) (new-support? #f) (unsupported '()))
    (cond ((pair? nodes)
           (let ((node (car nodes))
                 (nodes (cdr nodes)))
             (cond ((tms:node-supported? node)
                    (loop nodes new-support? unsupported))
                   ((there-exists? (tms:node-justifications node)
                      tms:justification-supported?)
                    (tms:re-support-node node)
                    (loop nodes #t unsupported))
                   (else
                    (loop nodes new-support? (cons node unsupported))))))
          (new-support?
           (tms:re-support-nodes unsupported))
          (else
           (for-each (lambda (node)
                       (tms:signal-event node #f))
                     unsupported)))))

(define (tms:re-support-node node)
  (if (not (tms:node-supported? node))
      (begin
        (tms:set-node-supported?! node #t)
        (for-each (lambda (just)
                    (if (tms:justification-supported? just)
                        (tms:re-support-node
                         (tms:justification-consequent just))))
                  (tms:node-consequents node)))))
```

```
;;;; Printing proofs

(define (tms:why? node #!optional port)
  (tms:walk-node-support node
    (lambda (consequent just antecedents)
      (tms:write-comment
       (cons (tms:node-name consequent)
             (cond ((tms:premise-justification? just) '(premise))
                   (else
                    `(<==
                      (,(or (tms:justification-rule just) just)
                       ,@(map tms:node-name antecedents))))))
       port))))

(define (tms:walk-node-support node procedure)
  (let ((n-tree (tms:node-support-tree node)))
    (if (not n-tree)
        (error:bad-range-argument node 'tms:walk-support-proof))
    (let ((queue (make-queue))
          (seen (make-eq-hash-table)))
      (let ((maybe-enqueue
             (lambda (n-tree)
               (hash-table/intern! seen (car n-tree)
                 (lambda ()
                   (enqueue! queue n-tree)
                   #t)))))
        (maybe-enqueue n-tree)
        (do ()
            ((queue-empty? queue) unspecific)
          (let ((n-tree (dequeue! queue)))
            (if (pair? (cdr n-tree))
                (let ((consequent (car n-tree))
                      (j-tree (cadr n-tree)))
                  (if (pair? j-tree)
                      (begin
                        (procedure consequent
                                   (car j-tree)
                                   (map car (cdr j-tree)))
                        (for-each maybe-enqueue (cdr j-tree))))))))))))
```

```
(define (tms:node-support-tree node)
  (let ((tms (tms:node-tms node)))

    (define (walk-node node seen)
      ;; Returns #F or an n-tree.
      (if (memq node seen)
          #f
          (let ((j-trees
                 (delq! #f
                        (map (let ((seen (cons node seen)))
                               (lambda (just)
                                 (walk-just just seen)))
                             (tms:node-justifications node)))))
            (and (pair? j-trees)
                 (cons node j-trees)))))

    (define (walk-just just seen)
      ;; Returns #F or a j-tree.
      (if (tms:premise-justification? just)
          (and (tms:premise-justification-supported? just)
               (cons just '()))
          (let ((n-trees
                 (map (lambda (node)
                        (walk-node node seen))
                      (tms:justification-antecedents just))))
            (and (for-all? n-trees (lambda (x) x))
                 (cons just n-trees)))))

    (walk-node node '())))

(define (tms:node-name node)
  (list (symbol 'n (object-hash node))
        '=
        (tms:node-datum node)))

(define (tms:write-comment object #!optional port)
  (write-char #\; port)
  (write object port)
  (newline port))
```

```
#|
(define tms
  (tms:make "n"
            (lambda (node justification)
              (tms:write-comment
               `(,(if justification 'supported 'unsupported)
                 ,(tms:node-name node))))))

(define a (tms:make-node tms 'A))
(define b (tms:make-node tms 'B))
(define c (tms:make-node tms 'C))
(define d (tms:make-node tms 'D))
(define e (tms:make-node tms 'E))
(define f (tms:make-node tms 'F))
(define g (tms:make-node tms 'G))
(define h (tms:make-node tms 'H))
(define i (tms:make-node tms 'I))

(tms:justify-node e 'MP (list a b))
(tms:justify-node e 'X (list c))
(tms:justify-node g 'Y (list e))
(tms:justify-node h 'Z (list e f))
(tms:justify-node f 'W (list c d))

(tms:assume-node c)
;(supported (n78 = c))
;(supported (n79 = e))
;(supported (n80 = g))
;Unspecified return value

(tms:node-supported? c)
;Value: #t

(tms:node-supported? e)
;Value: #t

(tms:why? g)
;((n80 = g) <== (y (n79 = e)))
;((n79 = e) <== (x (n78 = c)))
;((n78 = c) premise)
;Unspecified return value

(tms:retract-node c)
;(unsupported (n78 = c))
;(unsupported (n80 = g))
;(unsupported (n79 = e))
;Unspecified return value

(tms:assume-node a)
;(supported (n81 = a))
;Unspecified return value

(tms:assume-node b)
;(supported (n82 = b))
;(supported (n79 = e))
;(supported (n80 = g))
;Unspecified return value
```

```
(tms:assume-node c)
;(supported (n78 = c))
;Unspecified return value

(tms:assume-node d)
;(supported (n83 = d))
;(supported (n84 = f))
;(supported (n85 = h))
;Unspecified return value

(tms:retract-node a)
;(unsupported (n81 = a))
;Unspecified return value

(tms:why? h)
;((n85 = h) <== (z (n79 = e) (n84 = f)))
;((n79 = e) <== (x (n78 = c)))
;((n84 = f) <== (w (n78 = c) (n83 = d)))
;((n78 = c) premise)
;((n83 = d) premise)
;Unspecified return value

(tms:retract-node d)
;(unsupported (n83 = d))
;(unsupported (n85 = h))
;(unsupported (n84 = f))
;Unspecified return value

(tms:justify-node i 'U (list f))
;Unspecified return value

(tms:justify-node i 'Q (list g))
;(supported (n86 = i))
;Unspecified return value

(tms:justify-node d 'R (list i))
;(supported (n83 = d))
;(supported (n84 = f))
;(supported (n85 = h))
;Unspecified return value

(tms:why? h)
;((n85 = h) <== (z (n79 = e) (n84 = f)))
;((n79 = e) <== (x (n78 = c)))
;((n84 = f) <== (w (n78 = c) (n83 = d)))
;((n78 = c) premise)
;((n83 = d) <== (r (n86 = i)))
;((n86 = i) <== (q (n80 = g)))
;((n80 = g) <== (y (n79 = e)))
;Unspecified return value
```

```
(tms:retract-node c)
;(unsupported (n78 = c))
;(unsupported (n85 = h))
;(unsupported (n84 = f))
;(unsupported (n83 = d))
;(unsupported (n86 = i))
;(unsupported (n80 = g))
;(unsupported (n79 = e))
;Unspecified return value

(tms:assume-node a)
;(supported (n81 = a))
;(supported (n79 = e))
;(supported (n80 = g))
;(supported (n86 = i))
;(supported (n83 = d))
;Unspecified return value

(tms:why? d)
;((n83 = d) <== (r (n86 = i)))
;((n86 = i) <== (q (n80 = g)))
;((n80 = g) <== (y (n79 = e)))
;((n79 = e) <== (mp (n81 = a) (n82 = b)))
;((n81 = a) premise)
;((n82 = b) premise)
;Unspecified return value

(tms:assume-node c)
;(supported (n78 = c))
;(supported (n84 = f))
;(supported (n85 = h))
;Unspecified return value

(tms:retract-node c)
;(unsupported (n78 = c))
;(unsupported (n84 = f))
;(unsupported (n85 = h))
;Unspecified return value

|#

;;; Edwin Variables:
;;; lisp-indent/define-unary-record-method: 2
;;; End:
```

```
(define (lhs exp)
  (cadr exp))

(define (rhs exp)
  (caddr exp))
```