Outline

- what is a constraint
- the constraint logic programming framework
- declarative and procedural readings of CLP programs
- the CLP(R) language and its implementation

Some Practical Applications of Constraints

- planning, scheduling, timetabling (see ILOG solver especially)
- configuration
- electrical circuit analysis, synthesis, and diagnosis
- financial: options trading, financial planning
- cutting stock problems
- natural language processing
- restriction site mapping (genetics application)
- generating test data for communications protocols

Definition of a Constraint

What is a constraint? Informally, a constraint is a relation that we would like to be satisfied.

Examples:

- two columns in a table be of equal widths
- one window on a screen be above another window
- a resistor in a circuit simulation should obey Ohm's Law

Advantages: declarative, high-level, natural for many applications,

Disadvantages: easier to state than to satisfy, debugging and usability issues, complex interactions with state and object identity

Constraints in Constraint Logic Programming

The CLP and logic programming community uses the following more formal definition of constraint.

The form and meaning of a constraint is specified by a *domain* $\mathcal{D}$, including syntax for the constraints, permissible values for the variables, and meanings of the symbols in the constraint.

Example: $Y \times Y < 1$ means different things for integers, reals, or complex numbers.

Definition: a *primitive constraint* consists of a constraint relation symbol from $\mathcal{D}$ with the correct number of arguments. Each argument is constructed from variables and the constants and functions of $\mathcal{D}$.

Domains and Constraints

Example domain: the real numbers with the standard arithmetic functions and relations. The domain is $\mathcal{R}$, the function symbols are $+$, $-$, $\times$ and $/$, and the constraint relation symbols are $=$, $<$, $\leq$, $\geq$, and $>$.

Some other domains: integers, booleans, trees (finite or infinite).

Definition: a *constraint* is of the form
$c_1 \wedge \ldots \wedge c_n$ where $n \geq 0$ and $c_1, \ldots c_n$ are primitive constraints.

(Note: the UI people normally do not make a distinction between constraints and primitive constraints, and also regard for example $+$ as a constraint itself, rather than a function symbol.)

Valuations and Satisfiability

Definition: a *valuation* θ for a set V of variables is an assignment of values from the domain to those variables. For variables $V = \{X_1, \ldots X_n\}$ θ may be written $\{X_1 \rightarrow d_1, \ldots X_n \rightarrow d_n\}$.

Let *vars*(e) be the variables occuring in an expression e, and *vars*(C) be the variables occuring in a constraint C.

If θ is a valuation for V where *vars*$(C) \subseteq V$ then it is a *solution* of C if $\theta(C)$ holds in the constraint domain.

A constraint C is satisfiable if it has a solution. Otherwise it is unsatisfiable.

Two constraints are *equivalent*, written $C_1 \leftrightarrow C_2$ if they have the same set of solutions.

Solver Properties

Constraint solvers accept a constraint as input. (Remember these can be composed of multiple primitive constraints.) Output is:

- *true* (constraints are satisfiable)
- *false* (constraints are unsatisfiable)
- *unknown*

Definition: a solver is *complete* if for every constraint in $\mathcal{D}$ the solver's output is either *true* or *false*.

We will not be interested in unsound solvers (which might output true for unsatisfiable constraints or false for satisfiable constraints).

We prefer that solvers be complete, but for some domains this is not practical or even possible.

Mini-Exercises

Consider the domain of the reals.

Suppose we have a solver that always outputs *unknown*. Is this solver sound? Complete?

Suppose we have a solver that always outputs *true*. Is this solver sound? Complete?

Let C_1 be the constraint $X \geq 10 \wedge X + 5 = Y$. Is C_1 satisfiable?

Give a valuation that is a solution for C_1, and a valuation that is not a solution.

Is C_1 equivalent to $X \geq 10 \wedge Y \geq 5$?

The CLP Scheme

CLP($\mathcal{D}$) is a language framework, where D is the domain of the constraints.

Example CLP languages:

- Prolog
- CHIP
- Prolog III – domain is rationals, booleans, and trees
- CLP (Σ*) – domain is regular sets
- CLP($\mathcal{R}$) – domain is reals (plus trees, i.e. the data types that Prolog uses)

CLP($\mathcal{R}$) – Domain and Solver

CLP($\mathcal{R}$) can solve arbitrary collections of linear equality and inequality constraints.

It can also solve other kinds of constraints over the reals if it can find the answer using one-step deductions (first find this variable using one constraint, then find another variable using another constraint, etc — but no simultaneous equations).

CLP($\mathcal{R}$) Examples

Sample goals (just using primitive constraints – no user-defined rules):

```
?- X=Y+1, Y=10.
  X=11, Y=10

?- 2*A+B=7, 3*A+B=9.
  A=2, B=3

? X>=2*Y, Y>=5, X<=10.
  X=10, Y=5.

?- X*X*X + X = 10.
  maybe
```

(The last goal does have a solution `X=2`. The “maybe” answer means the constraints are too hard for CLP($\mathcal{R}$) to solve.)

CLP($\mathcal{R}$) Examples

CLP($\mathcal{R}$) programs are collections of *facts* and *rules*.

Sample rule:

```
/* centigrade-fahrenheit relation */
cf(C,F) :-
  F = 1.8*C + 32.
```

Sample Goals:

```
?- cf(100,A).
  A=212.0

?- cf(A,B), A>100, B<200.
  no.

?- cf(X,X).
  X=-40.0
```

Formal Definitions - CLP Constituents

A *user defined constraint* is of the form $p(t_1, \ldots t_n$ where p is an n-ary predicate and $t_1, \ldots t_n$ are expressions from the constraint domain.

A *literal* is either a primitive constraint or a user-defined constraint.

A *goal* G is a sequence of literals. G has the form $L_1, L_2, \ldots L_m$ where $m \geq 0$. If $m = 0$ the goal is *empty* and is represented by $\Box$

A *rule* R is of the form $A : -B$ where A is a user-defined constraint and B is a goal. A is the head of the rule and B is the body. (Read this as B implies A.)

A *fact* is a rule with the empty goal as the body $A : -\Box$ and is just written as A. (Read this as A is true.)

A *constraint logic program* P is a sequence of rules.

The *definition* of a predicate p in a program P is the sequence of rules appearing in P which have a head involving predicate p. (More formally "involving predicate p" means that the head of those rules can be unified with p, in other words, we can solve a tree equality constraint between them.)

Evaluation in CLP Languages – Informal Discussion

Given an initial goal, a CLP interpreter rewrites any user-defined constraints in the goal using their definitions.

This may yield more user defined constraints, which are then rewritten.

Primitive constraints are kept in a *constraint store*.

We continue until there are only primitive constraints, which are solved by the system.

However, if the constraint store contains an unsatisfiable set of constraints, we can stop rewriting immediately.

We may have multiple rules for a given user-defined constraint. We try these in order, backtracking if one fails.

Rewriting – More Formal Definition

Let a goal G be of the form

$L_1, \ldots L_{i-1}, L_i, L_{i+1}, \ldots L_m$

where L_i is the user-defined constraint $p(t_1, \ldots, t_n)$ and rule R is of the form

$p(s_1, \ldots, s_n) : -B.$

A *rewriting* of G at L_i by R using ρ is the goal

$L_1, \ldots L_{i-1}, t_1 = \rho(s_1), \ldots, t_n = \rho(s_n), \rho(B), L_{i+1}, \ldots L_m$

where ρ is a renaming such that the variables in $\rho(R)$ do not appear in G.

Example Rewriting

Consider the goal `cf(100,A), B=A+100`.

Let's rewrite this using the rule

```
cf(C,F) :-
  F = 1.8*C + 32.
```

We can use the empty renaming, since there are no variables in common between the goal and the rule. The new goal is
$100 = C, A = F, F = 1.8 * C + 32, B = A + 100$

Mini-Exercise

Rewrite the goal `cf(F,F)` using the `cf` rule:

```
cf(C,F) :-
  F = 1.8*C + 32.
```


Evaluation in CLP Languages – More Formal Treatment

The state of the computation at any point consists of the current goal G and the constraint store C. Remember that G is a conjunction of literals, and the constraint store holds primitive constraints.

Formally, a state is a pair written $\langle G \mid C \rangle$.

A *derivation step* involves processing a constraint from G.

A derivation step is written as
$\langle G_1 \mid C_1 \rangle \Rightarrow \langle G_2 \mid C_2 \rangle$.

Evaluation (continued)

Suppose $G_1 = L_1, \ldots L_m$ for literals $L_1, \ldots L_m$.

Case 1. L_1 is a primitive constraint. Then the next state is $\langle G_2 \mid C_2 \rangle$, where $C_2 = C_1 \wedge L_1$.

If *solv*(C_2) $\not\equiv$ *false* $G_2 = L_2, \ldots L_m$.

If *solv*(C_2) $\equiv$ *false* $G_2 = \square$.

Case 2. L_1 is a user defined constraint. Then $C_2 = C_1$, and G_2 is a rewriting of G_1 at L_1 by some rule R in the program. If there is no rule defining the predicate of L_1 then C_2 is *false* and G_2 is the empty goal.

Success and Failure

A *success state* is a state $\langle \Box \mid C \rangle$ where *solv*$(C) \not\equiv$ *false*.

A *fail state* is a state $\langle \Box \mid C \rangle$ where *solv*$(C) \equiv$ *false*.

A derivation $\langle G_0 \mid C_0 \rangle \Rightarrow \ldots \Rightarrow \langle G_n \mid C_n \rangle$ is successful if $\langle G_n \mid C_n \rangle$ is a success state.

The constraints resulting from simplifying C_n with respect to the variables in the original goal G_0 are the *answer* to $\langle G_0 \mid C_0 \rangle$.

If $\langle G_n \mid C_n \rangle$ is a failed state then the derivation is *failed*.

Answer to Example Rewriting

Earlier we rewrote the goal `cf(100,A), B=A+100` using the rule

```
cf(C,F) :-
  F = 1.8*C + 32.
```

The new goal is
$100 = C, A = F, F = 1.8 * C + 32, B = A + 100$

The *answer* is $A = 212, B = 312$

Example Derivation

CLP($\mathcal{R}$) program:

```
cf(C,F) :-   /* rule R1 */
  F = 1.8*C + 32.

double(X,Y) := Y=2*X.  /* rule R2 */
```

Consider the goal `cf(A,B), double(A,200)`.

$\langle \mathit{cf}(A,B), \mathit{double}(A,200) \mid \mathit{true} \rangle$

$\Rightarrow$

using R1:
$\langle A = C, B = F, F = 1.8 * C + 32,$
$\mathit{double}(A,200) \mid \mathit{true} \rangle$

$\Rightarrow$

$\langle B = F, F = 1.8 * C + 32, \mathit{double}(A,200) \mid$
$A = C \rangle$

$\Rightarrow$

$\langle F = 1.8 * C + 32, \mathit{double}(A,200) \mid A = C,$
$B = F \rangle$

$\Rightarrow$

$\langle \mathit{double}(A,200) \mid A = C, B = F,$
$F = 1.8 * C + 32 \rangle$

$\Rightarrow$

using R2:
$\langle A = X, 200 = Y, Y = 2 * X \mid A = C, B = F,$
$F = 1.8 * C + 32 \rangle$

$\Rightarrow$

$\langle 200 = Y, Y = 2 * X \mid A = C, B = F,$
$F = 1.8 * C + 32, A = X\rangle$

$\Rightarrow$

$\langle Y = 2 * X \mid A = C, B = F, F = 1.8 * C + 32,$
$A = X, 200 = Y\rangle,$

$\Rightarrow$

$\langle \square \mid A = C, B = F, F = 1.8 * C + 32,$
$A = X, 200 = Y, Y = 2 * X\rangle$

Simplifying with respect to the variables in G_0 (namely A, B) we get the answer $A = 100, B = 212$

Infinite Derivations

Consider the following rules defining the natural numbers:

```
natural(N) :- natural(N-1).  /* Rule R1 */
natural(1).                  /* rule R2 */
```

Given the goal `natural(3)` one derivation is

$\langle \mathit{natural}(3) \mid \mathit{true}\rangle$

$\Rightarrow$ (using R1)

$\langle N = 3, \mathit{natural}(N - 1) \mid \mathit{true}\rangle$

$\Rightarrow$

$\langle \mathit{natural}(N - 1) \mid N = 3\rangle$

$\Rightarrow$ (using R1)

$\langle N - 1 = N', \mathit{natural}(N' - 1) \mid N = 3\rangle$

$\Rightarrow$

$\langle \mathit{natural}(N' - 1) \mid N = 3, N - 1 = N'\rangle$

$\Rightarrow$ (using R1)

$\langle N' - 1 = N'', \mathit{natural}(N'' - 1) \mid$
$N = 3, N - 1 = N'\rangle$

$\Rightarrow$

$\langle \mathit{natural}(N'' - 1) \mid$
$N = 3, N - 1 = N', N' - 1 = N''\rangle$

$\Rightarrow$ (using R1)

$\langle N'' - 1 = N''', \mathit{natural}(N''' - 1) \mid$
$N = 3, N - 1 = N', N' - 1 = N'', N'' - 1 = N'''\rangle$

$\Rightarrow \ldots$

A Different Derivation

Another derivation for the goal `natural(3)` is

$\langle \mathit{natural}(3) \mid \mathit{true}\rangle$

$\Rightarrow$ (using R1)

$\langle N = 3, \mathit{natural}(N - 1) \mid \mathit{true}\rangle$

$\Rightarrow$

$\langle \mathit{natural}(N - 1) \mid N = 3\rangle$

$\Rightarrow$ (using R1)

$\langle N - 1 = N', \mathit{natural}(N' - 1) \mid N = 3\rangle$

$\Rightarrow$

$\langle \mathit{natural}(N' - 1) \mid N = 3, N - 1 = N'\rangle$

$\Rightarrow$ (using R2)

$\langle N' - 1 = 1 \mid N = 3, N - 1 = N' \rangle$

$\Rightarrow$

$\langle \Box \mid N = 3, N - 1 = N', N' - 1 = 1 \rangle$
(a success state)

Choices in CLP($\mathcal{R}$)

1. When a goal contains more than one literal, which literal to select?

2. When more than one rule matches the selected literal, which rule to select?

Theorem: for a derivation that is successful, choosing different literals still gives the same result. (Intuition: for a successful derivation we have to process each literal sooner or later.)

Our CLP($\mathcal{R}$) implementation always selects the leftmost literal.

The choice of rule is more interesting.

We can get different answers for the same goal given different rule choices. (In other words, there can be more than one successful derivation.)

Also some rule choices may result in infinite derivations — so it matters which order we try them in.

Definition: a *derivation tree* for a goal G and program P is a tree with states as nodes. The root of the tree is $\langle G \mid \mathit{true} \rangle$. The children of each state $\langle G_i \mid C_i \rangle$ are those states that can be reached in a single derivation step.

A state with two or more children is a *choice-point*. (This happens only for user-defined constraints that have multiple matching rules.)

CLP($\mathcal{R}$) first selects the rule that occurs first in the program text. If this derivation fails, it selects the next rule, and so forth. To implement this it keeps a stack of backtrack points, and uses depth-first search.

The order of rules in the program can make a difference!

Mini-Exercises

Write CLP($\mathcal{R}$) rules to define the "max" relation. Here are some sample goals:

```
?- max(10,20,X).
   X=20

?- max(10,20,30).
   no
```

What are the answers for the following goals? If there is more than one answer give all of them. Show the derivation tree (skipping some details of processing the primitive constraints if you wish).

```
?- max(1,2,A).

?- max(X,Y,20).
```


Tree Constraints

Besides the domain of the real numbers, CLP($\mathcal{R}$) has another domain: trees. These allow us to model data structures such as lists, records, and trees.

Definitions: a *tree constructor* is a symbol beginning with a lower-case letter. A *tree* is either a constant, or a tree constructor together with an ordered list of one or more trees, which are its children.

A *term* is either a constant, a variable, or a tree constructor together with an ordered list of one or more trees, which are its children.

The only relation among trees we will use is equality.

In CLP($\mathcal{R}$) atoms start with lower-case letters, and variables with capital letters.

(Prolog fans: trees are the data structures that Prolog uses. Unification is the algorithm used to solve equality constraints over trees.)

Examples of trees:

```
fred
X
point(10,20)
point(X,Y)
line(point(X1,Y1),point(X2,Y2))
node(3,emptynode,node(4,emptynode,emptynode))
```

There is special syntax for lists:

```
[ ]  /* the empty list */
[1,2,3]
[1 | [2,3]]  /* the same as [1,2, */
[a, b, [c], [d] ]  /* sublists OK */
```

Examples of tree constraints:

`A = point(10,20)`
has solution `A=point(10,20)`

`point(X,X) = point(10,Y)`
has solution `X=Y=10`

`point(X,X) = point(10,20)`
has no solution

`[A,B,C] = [1,2,3]`
has solution `A=1, B=2, C=3`

`[X|Xs] = [1,2,3]`
has solution `X=1, Xs=[2,3]`

`[X|Xs] = [100]`
has solution `X=100, Xs=[]`

Tree Constraint Solver

To solve $e_1 = e_2$

If e_1 is a variable v, then succeed, and return the substitution $v = e_2$

If e_2 is a variable v, then succeed, and return the substitution $v = e_1$

If e_1 and e_2 are constants, if they are the same succeed; if they are different fail.

If only one of e_1 and e_2 is a constant, fail.

Otherwise both e_1 and e_2 consist of a tree constructor with an ordered list of children. If e_1 and e_2 have different constructors or different numbers of children, then fail. Otherwise $e_1 = p(s_1, \ldots, s_n)$ and $e_2 = p(t_1, \ldots, t_n)$. Recursively solve the constraints $s_1 = t_1, \ldots s_n = t_n$. Succeed if all of them can be solved, and return the combined subsitution. Otherwise fail.

Some Simple Recursive CLP($\mathcal{R}$) Programs

```
/* LENGTH OF LIST */

length([],0).
length([H|T],N) :-
  N > 0,
  length(T,N-1).

/* compare this with a scheme program:
(define (length x)
  (if (null? x) 0
      (+ 1 (length (cdr x)))))
*/


/* SUM OF THE ELEMENTS IN A LIST */

sum([],0).
sum([X|Xs],X+S) :- sum(Xs,S).
```



```
/* FACTORIAL */
factorial(0, 1).
factorial(N, N * F) :-
        N > 0 ,
        fact(N - 1, F).
```

Mini-Exercises

What are the outputs for the following goals? Show the derivation tree (skipping some details of processing the primitive constraints if you wish).

```
?- length([a,b,c],N).

?- length([X|Xs],N).

?- length(L,2).
```

Greatest Common Divisor

```
/* GREATEST COMMON DIVISOR
   (USING EUCLID'S ALGORITHM) */

gcd(A,B,G) :-
   A < B,
   gcd(A,B-A,G).

gcd(A,B,G) :-
   A > B,
   gcd(A-B,B,G).

gcd(A,A,A).
```


Quicksort

```
quicksort([],[]).
quicksort([X|Xs],Sorted) :-
  partition(X,Xs,Smalls,Bigs),
  quicksort(Smalls,SortedSmalls),
  quicksort(Bigs,SortedBigs),
  append(SortedSmalls,[X|SortedBigs],Sorted).

partition(Pivot,[],[],[]).
partition(Pivot,[X|Xs],[X|Ys],Zs) :-
   X <= Pivot,
   partition(Pivot,Xs,Ys,Zs).
partition(Pivot,[X|Xs],Ys,[X|Zs]) :-
   X > Pivot,
   partition(Pivot,Xs,Ys,Zs).


append([],X,X).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```


Electrical Circuit Example

```
resistor(lead(I1,V1),lead(I2,V2),Ohms) :-
   I1+I2=0,
   V2-V1=I1*Ohms.

battery(lead(I1,V1),lead(I2,V2),Volts) :-
   V1 = V2+Volts,
   I1+I2=0.

electrical_ground(lead(0,0)).

ammeter(lead(I1,V),lead(I2,V),I1) :-
   I1+I2=0.

voltmeter(lead(0,V1),lead(0,V2),Volts) :-
   V1-V2=Volts.
```



```
/* rule to connect a list of leads together
   (makes all the voltages the same, and the
   sum of the currents be 0 */
connect(Leads) :-
  same_voltages(Leads),
  currents_sum(Leads,0).

same_voltages([]).
same_voltages([L]).
same_voltages([lead(I1,V),lead(I2,V)|More]) :-
   same_voltages([lead(I2,V)|More]).

currents_sum([],0).
currents_sum([lead(I1,V1)|More],I1+Sum) :-
   currents_sum(More,Sum).
```

```
/* RULES TO BUILD THE SAMPLE CIRCUITS */

/* simple battery-resistor circuit */
one_resistor(Volts,Ohms,Amps) :-
   battery(B1,B2,Volts),
   resistor(R1,R2,Ohms),
   ammeter(A1,A2,Amps),
   electrical_ground(G),
   connect([B2,A1]), connect([A2,R1]),
   connect([R2,B1,G]).

/* same circuit, but no ground */
one_noground(Volts,Ohms,Amps) :-
   battery(B1,B2,Volts),
   resistor(R1,R2,Ohms),
   ammeter(A1,A2,Amps),
   connect([B2,A1]), connect([A2,R1]),
   connect([R2,B1]).

/* voltage divider */
divider(Volts,Ohms1,Ohms2,Amps,VCenter) :-
   battery(B1,B2,Volts),
   resistor(R1,R2,Ohms1),
   resistor(S1,S2,Ohms2),
   ammeter(A1,A2,Amps),
   voltmeter(V1,V2,VCenter),
   electrical_ground(G),
   connect([B2,A1]), connect([A2,R1]),
   connect([R2,S1,V1]), connect([S2,V2,B1,G]).

divider_noground(Volts,Ohms1,Ohms2,Amps,VCenter) :-
   battery(B1,B2,Volts),
   resistor(R1,R2,Ohms1),
   resistor(S1,S2,Ohms2),
   ammeter(A1,A2,Amps),
   voltmeter(V1,V2,VCenter),
   connect([B2,A1]), connect([A2,R1]),
   connect([R2,S1,V1]), connect([S2,V2,B1]).

/* Wheatstone bridge */
wheat(Volts,WOhms,XOhms,YOhms,ZOhms,Amps) :-
   battery(B1,B2,Volts),
   resistor(W1,W2,WOhms),
   resistor(X1,X2,XOhms),
   resistor(Y1,Y2,YOhms),
   resistor(Z1,Z2,ZOhms),
   ammeter(A1,A2,Amps),
   electrical_ground(G),
   connect([B2,W1,X1]), connect([B1,Y2,Z2,G]),
   connect([W2,Y1,A1]), connect([X2,Z1,A2]).

wheat_noground(Volts,WOhms,XOhms,YOhms,ZOhms,Amps) :-
   battery(B1,B2,Volts),
   resistor(W1,W2,WOhms),
   resistor(X1,X2,XOhms),
   resistor(Y1,Y2,YOhms),
   resistor(Z1,Z2,ZOhms),
   ammeter(A1,A2,Amps),
   connect([B2,W1,X1]), connect([B1,Y2,Z2]),
   connect([W2,Y1,A1]), connect([X2,Z1,A2]).

ladder(T1,T2,Ohms,1) :-
   /* one rung */
   resistor(T1,A,Ohms),
   resistor(T2,B,Ohms),
   resistor(R1,R2,Ohms),
   connect([A,R1]), connect([B,R2]).

ladder(T1,T2,Ohms,N) :-
   N>1,
   ladder(X1,X2,Ohms,N-1),
   resistor(T1,A,Ohms),
   resistor(T2,B,Ohms),
   resistor(R1,R2,Ohms),
   connect([A,R1,X1]), connect([B,R2,X2]).


/* SAMPLE GOALS */

go1 :- one_resistor(100,50,A), dump([A]).
go2 :- one_noground(100,50,A), dump([A]).
go3 :- divider(100,30,20,Amps,VCenter),
       dump([Amps,VCenter]).
go4 :- divider_noground(100,30,20,Amps,VCenter),
```

```
       dump([Amps,VCenter]).
go5(XOhms) :- wheat(100,100,XOhms,50,30,Amps),
       dump([XOhms,Amps]).
go6(XOhms) :-
       wheat_noground(100,100,XOhms,50,30,Amps),
       dump([XOhms,Amps]).

go7(N) :-
  ladder(T1,T2,10,N),
   battery(B1,B2,100),
   ammeter(A1,A2,Amps),
   electrical_ground(G),
   connect([B2,A1]), connect([A2,T1]),
   connect([T2,B1,G]),
   dump([Amps]).
```

CLP($\mathcal{R}$) Implementation

Constraint Logic Abstract Machine (CLAM) — derived from Warren Abstract Machine (WAM)

The Engine

The engine is a structure sharing Prolog interpreter (see Figure 2, page 360)

Distinguish between constraints that can be handled in the engine (e.g. nonsolver variable = number) and those that must be passed to the interface.

Constraints that can be handled in the engine are shown in figure 3, page 361.

The Interface

Simplify input constraint by evaluating arithmetic expressions. If constraint is ground, test it.

If there is one non-solver variable, set up a binding. Otherwise put constraint into a canonical form and invoke solver.

Solver

solver modules:

- equality solver
- inequality solver
- nonlinear handler

Equality Solver

Equality solver uses variant of Gaussian elimination.

Represent *nonparametric variables* in terms of *parametric variables* and constants. Central data structure: a tableau (2d array). Each row represents some nonparametric variable as a linear combination of parametric variables.

Equality solver is invoked by the interface with a new equality, from the inequality solver with an implicit equality, or with a delayed constraint from the nonlinear handler.

Inequality Solver and Nonlinear Handler

Inequality Solver: adapted from first phase of two-phase Simplex algorithm.

Simplex augmentations:

- unconstrainted variables and slack variables
- symbolic entries denoting infinitesimal values
- negative or positive coefficients for basic unconstrained variables

Solver detects implicit equalities (could scrap equality solver and just do it all with Simplex ...)

Nonlinear handler: delay nonlinear constraints until they become linear