DEPARTMENT OF ARTIFICIAL INTELLIGENCE
UNIVERSITY OF EDINBURGH

DAI Software Paper No. 12
Date: August 1987

# A PROLOG Version of ACRONYM's CMS - Version 1.1)

Robert Fisher

Abstract:

This document summarizes a re-implementation of ACRONYM's [1] algebraic constraint manipulation system (in PROLOG), largely developed using Brooks original rules. It gives the syntax for the algebraic expressions bounded, how the CMS is invoked and explanations of error messages produced. Debugging facilities are described. Some previously unpublished details of operation are reported, concerning program structure, expression simplification, efficiency approximate bounding and parity determination. Various extensions, improvements and bug fixes are also given, along with an example.

Acknowledgements

Funding for this paper was under Alvey grant GR/D/1740.3. The ideas in this paper benefited greatly from discussions with J. Aylett and M. Orr.

## 1.0 Introduction

This document describes a PROLOG implementation of ACRONYM's Constraint Manipulation System (CMS), as described in Brooks' thesis [Brooks 1981]. Little effort is made (here) to describe how the CMS works, except that needed for understanding this particular implementation. The main body of this note describes the format of expressions and constraints used by the CMS, how to invoke the CMS to bound an expression and how expressions are simplified. Some improvements are described as are some debugging tools.

## 2.0 Declarations, Expressions and Constraints

All variables must have their type defined. The two types are 'linear', for the usual unbounded variables, and 'wraparound' for modular variables, like angles. The wraparound variables also have an upper and lower bound associated with them. The syntax for declarations is:

declare ( <string> , <variable_type> )

<variable_type> <- linear
|
wraparound ( <number> , <number> )

where the <string> is the variable name, and the two <number>s are the low and high bounds for the wraparound variables range (usually 0 and 2*PI). This also implicitly declares expressions over the declared variables. Expressions may not mix variables of different types, unless converted by some operator (such as a 'cos' function).

Expressions are defined using nested lists of descriptive terms, embedding numbers, variables and operators. The syntax for expressions is:

<expression> <- [ value, <number> ]

defines a constant. <number> can be positive or negative, integer or floating point, or 'p_infinity' or 'n_infinity' or 'undefined' (section 7.12).

<expression> <- [ variable, <string>, <positive_integer> ]

defines a variable of name <string> raised to the power <positive_integer>

<expression> <- [ plus, <expression>, <expression> ]

defines the sum of two expressions

<expression> <- [ times, <expression>, <expression> ]

defines the product of two expressions

<expression> <- [ recip, <expression> ]

defines the reciprocal of an expression

<expression> <- [ srecip, <expression> ]

defines the signed reciprocal of an expression (see section 7)

<expression> <- [ cos, <expression> ]

defines the cosine of an expression

<expression> <- [ sin, <expression> ]

defines the sine of an expression

<expression> <- [ sqrt, <expression> ]

defines the positive square root of an expression

<expression> <- [ min, [ <expression_list> ] ]

defines the minimum of a list of expressions

<expression> <- [ max, [ <expression_list> ] ]

defines the maximum of a list of expressions

<expression_list> <- <expression>
| <expression> , <expression_list>

Multiple powers of a variable is expressed using the exponent field of the variable variable construct. Hence,

2*x*x*x

is represented by

[times, [value, 2], [variable, x, 3]]

Division is expressed by multiplying by the reciprocal and subtraction by adding -1 times the value. That is,

<expression1> / <expression2>

is represented by

[times, <expression1>, [recip, <expression2> ]]

and

<expression1> - <expression2>

is represented by

```
[plus, <expression1>,
       [times, [value, -1], <expression2> ]
]
```

This seems awkward, but simplifies the cases needed handling by the CMS.

Constraints are defined over isolated variables, and give a bound-

ing expression that must have the relation ≤ or ≥ to the variable. The syntax for a constraint is:

<constraint> <- constraint ( <string> , <relation> ,
<expression> ) .

<relation> <- greatereq | lesseq

where <string> gives the name of the bound variable and <expression> is the bounding expression.

For example, the representation of the constraint

a ≤ 4/b

is

constraint(a,lesseq,[times, [value, 4],
[recip, [variable, b, 1]]]).

## 3.0 Bounding in Variables

The general form for invoking ACRONYM's bounding of expressions is:

cms(<input_expression>,[<variable_list>],<bound>,
<result_expression>)

where

<input_expression> and <result_expression>

are expressions as defined above

<variable_list> <- <string> | <string> , <variable_list>

is a list of variable names over which the expression is to be bounded, and

<bound> <- sup | inf

depending on which bound is desired.

The first three values are inputs. The program executes to completion silently, producing the bounding expression.

## 4.0 Simplification

Expressions get simplified at various points in the processing. This is partly necessary for cancelling terms (e.g. "x - x" or "x / x") and partly for efficiency by reducing the complexity of the expressions being manipulated (e.g. reducing "0 + x" or "1 + 2"). Brooks used three simplification applications each in his sup and inf functions and claimed that further applications would not affect correctness, but would reduce efficiency. To get the algorithm to work, we needed to

also apply simplification whenever a constant value was required, because otherwise there might be an unsimplified expression having a constant value (e.g. "1 + 2") (We also tried applying simplification after each completed application of sup and inf, and generally found that ACRONYM did run slower - especially on larger expressions.)

The simplifications 'simp' enacts on each of the different expression types are:

min - simplifies all arguments, flattens nested 'min's, removes duplicate entries, keeps only the smallest 'value' and removes 'min' if only one element is left in the list

max - simplifies all arguments, flattens nested 'max's, removes duplicate entries, keeps only the largest 'value' and removes 'max' if only one element is left in the list

times - simplifies arguments, does constant multiplications (0,1), multiplies together 'value's and flattens out multiply lists to collect variables and values

plus - simplifies arguments, removes zero additions, reduces additions of infinity, adds 'value's and reduces terms of the form ax + bx, where a and b are constants.

distribution laws - simplifies arguments, distributes 'times' over 'min' and 'max' (accounting for parity) and distributes 'plus' over 'min' and 'max'

recip - simplifies argument, divides 'value's and distributes 'recip' over 'times'

srecip - simplifies argument and divides 'value's

sqrt, cos, sin - simplifies argument and evaluates the function if the result is a 'value'

## 5.0 Debugging

If the CMS fails, then the most likely cause is an improperly constructed constraint. Then, the result of constraining an expression will be 'no'. Unfortunately, because this is a rule-application program, if one rule fails because of a bug, another might apply, leading to a weaker result.

Bugs in the CMS are difficult to detect, but a tool is provided that allows checking of the constraints. This is the function 'checkallconstraints' which checks all constraints for correct form and reports errors.

Some tracing tools are provided (besides the usual prolog tracing). These allow listing of the inputs into and out of each rule application, and into and out of the simplifier. No interactive control is allowed. Rule tracing is enabled by asserting the clause 'debug(rule)' and simplification tracing is enabled by asserting 'debug(simp)'. Retracting the clauses disables tracing.

An example of the rule tracing comes from applying:

```
cms([variable, a, 1],[],sup,Ans)
```

over the constraints:

```
constraint(a,greatereq,[value,2]).
constraint(a,lesseq,[times, [value, 4],
                             [recip, [variable, b, 1]]
                    ]).
constraint(b,greatereq,[value,1]).
constraint(b,lesseq,[times, [value, 4],
                             [recip, [variable, a, 1]]
                    ]).
```

producing:

```
Apply(0) sup rule2d over []:[variable,a,1]
 Apply(1) sup rule3a over [a]:[times,[value,4],
                                 [recip,[variable,b,1]]]
  Apply(2) sup rule9 over [a]:[recip,[variable,b,1]]
   Apply(3) inf rule2d over [a]:[variable,b,1]
    Apply(4) inf rule1 over [b,a]:[value,1]
    Result:[value,1]
    Apply(4) inff rule1 over [a]:[value,1]
    Result:[value,1]
   Result:[value,1]
  Result:[recip,[value,1]]
 Result:[times,[value,4],[recip,[value,1]]]
 Apply(1) supp rule1 over []:[value,4]
 Result:[value,4]
Result:[value,4]

Ans = [value,4]
```

The input tracing gives the depth level number (0), the rule applied (sup rule2d), the set of variables ([]) and the input expression ([variable,a,1]). The corresponding output tracing gives the resulting expression. The trace messages are indented according to depth level for easier inspection.

Some examples of the simplifier tracing on the above example are:

```
Simp in:[min,[[times,[value,4],[recip,[variable,b,1]]]]]
Simp out:[times,[value,4],[recip,[variable,b,1]]]

Simp in:[max,[[value,1]]]
Simp out:[value,1]

Simp in:[times,[value,4],[recip,[value,1]]]
Simp out:[value,4]
```

No indenting is done here.

## 6.0 Invoking the CMS

The CMS is loaded by consulting the following files:

```
sup,supp,suppp,inf,inff,infff,
utils,support,simp,interface,
arithmetic,setops,debugging,
checkconst
```

The file 'loadcms' contains a 'load' clause that loads all necessary files.

The CMS is invoked by the 'cms' function, as described in section 3.0. Before this, all constraints that might act on the expression must be included as part of the database. Two functions have been constructed to assist with this:

```
loadconstraints(<name>)
```

and

```
unloadconstraints(<name>)
```

These are useful for bounding expressions over different sets of constraints. The former asserts all constraints in the designated file and the second retracts all the constraints in the given file. Clauses in the file that are not constraints are neither asserted nor retracted.

## 7.0 Extensions to Brooks' CMS

Several changes and new rules have been added to Brooks' original CMS. These are described below:

(1) Simplification has been applied to the output of constant result calls to 'sup' and 'inf' in rules 9, 12 and 13 to simplify expressions involving constants.

(2) Whenever a single variable is bounded over a set of variables, that bound is recorded into the database and used again in the future. This does not change the functionality of the CMS but makes dramatic savings on re-constraining the same variable over again, which can occur if the variable occurs several times in parallel in the expression (such as in a 'max' expression). This is implemented as a new rule 2c between 2b and 2d (formerly 2.2 and 2.4).

(3) The CMS goes into an infinite loop at rule 4 over expressions of the form '2*x + x*x' = 'r*v + A' The problem is the rule does not reduce the second term (A) before recalling the simplifier. The solution adopted here requires that the variable v does not occur in A. Otherwise rule 6 applies.

(4) Simple extensions were added to: (a) apply 'sup' to 'max' and 'inf' to 'min' and (b) allow 'min' and 'max' to have an arbitrary number of arguments. The first extension obtains because 'sup(max(A,B))' ≤ 'max(sup(A),sup(B))'.

(5) A minor extension is for the sqrt function which is monotonic,

takes a positive value and returns a positive value. Here, 'sup(sqrt(A))' ≤ 'sqrt(sup(A))' and 'inf(sqrt(A))' ≥ 'sqrt(inf(A))'. These boundings are left in symbolic form (but possibly simplified later). The same extensions were added to suppp and infff.

(6) A new sup and inf case was added for expressions of the form: 'variable^n'. One reason for this was the CMS does not know how to handle quadratic terms when the parity of the variable is unknown. Another reason is the original rule 10 does not (strictly) apply when A = 1 in 'v^n * A'. The logic of the new rule constructs the following symbolic expressions:

sup(v^n) =

if n is odd then sup(v)^n
else if parity(v) = positive then sup(v)^n
else if parity(v) = negative then inf(v)^n
else max( sup(v)^n, inf(v)^n ).

inf(v^n) =

if n is odd then inf(v)^n
else if parity(v) = positive then inf(v)^n
else if parity(v) = negative then sup(v)^n
else 0.

Suppp and infff were also extended for expressions of the form: v^n. The resulting expressions follow the logic of the above extension only using suppp and infff in the recursive steps.

(7) The values of frequently recalculated but lengthy functions are sometimes saved using the database 'record' facility. An example of where this occurs is in the sup of a single variable occurring several places in a min expression (e.g. min(2x,1-x)). Previous evaluation is tested for with three groups of functions:

a) sup/inf of a single variable
b) hival/loval of a variable
c) parity of an expression

Saving other evaluations did not seem to make an appreciable difference.

This trades off database search against recalculation, but was found to produce a remarkable improvement in performance (e.g. a factor of 100), especially with complex expressions.

(8) The numerical values positive and negative infinity are represented explicitly as "p_infinity" and "n_infinity" and all arithmetic functions and boundings take these into account.

(9) Brooks' rule 13 for inf is wrong as described in the footnote to Brooks' sup table. The inequalities in parts 13.1 and 13.2 test for inconsistencies, so should be reversed in the inf

version. Alternatively, the role of sup and inf in the preparations could be reversed.

(10) Bounding a product over all the variables in the product (e.g. x*y over {x,y}) when the parity of neither multiplicand is known, results in a numerical bound (Brooks' rule 13). The original product is a suitable bound, given all variables are in the variable set. Another version of this problem occurs when all variables of A in A*B are in the variable set. This suggests two new sup and inf rules (rule 11_5 and 11_6). The rule numbering gives the appropriate rule ordering with respect to the other rules. Following Brooks' notation, the new rules are:

Let:

fulloccurs(Expr,Set) hold if all variables in Expr occur in Set

| If | Action | Return |
|---|---|---|
| sup rule 11_5 | | |
| fulloccurs(J,H) | | J |
| sup rule 11_6 | | |
| J = A*B<br>fulloccurs(A,H) | S = sup(B,H)<br>I = inf(B,H) | |
| if A positive | | A*S |
| if A negative | | A*I |
| Else | | max(A*S,A*I) |
| inf rule 11_5 | | |
| fulloccurs(J,H) | | J |
| inf rule 11_6 | | |
| J = A*B<br>fulloccurs(A,H) | S = sup(B,H)<br>I = inf(B,H) | |
| if A positive | | A*I |
| if A negative | | A*S |
| Else | | min(A*S,A*I) |

Other extensions are possible. If some of the variables in the variable set occur in the expression they might (in the future) be left as they are (which is what occurs in rules 3-11). Further, while rule 13 is correct, it is a bit strict since it leaves no variables in the result expression. Instead, they might be left and the resulting expression other-

wise simplified. This might introduce more complex expressions involving 'max' and 'min' over several new expressions. This argument also applies to rule 12.

(11) There is a problem of how to decompose product constraints (e.g. $A*B \leq c$) into constraints on the individual terms, when the parity of the individual terms is unknown. The solution adopted here is to introduce a new operator, the signed reciprocal and corresponding sup, inf and simplification rules. The result splits the cases by parity and turns off the inappropriate cases when further information is obtained.

If a constraint of the form

$$A*B \leq c$$

is encountered and no information is available on the parity of B, it is replaced during pre-processing by:

$$A \leq c * srecip(B)$$

$$A \geq -c * srecip(-B)$$

A similar reduction is made to isolate B and the inequalities are reversed for $A*B \geq c$.

The definition of the srecip function is:

$$\text{if } x > 0 \text{ then } srecip(x) = 1/x$$
$$\text{else } srecip(x) \text{ is undefined}$$

The use of the undefined evaluation disables one of the split inequalities. This allows the isolation of individual terms from a product.

Now, the simplification of srecip is

$$simp(srecip(A)) = srecip(simp(A))$$

with the

The sup rule is

$$\text{if } inf(A) > 0 \text{ then } sup\{srecip(A)\} = recip(inf(A))$$
$$\text{else } sup\{srecip(A)\} = \text{undefined}$$

and the inf rule is

$$\text{if } inf(A) > 0 \text{ then } inf\{srecip(A)\} = recip(sup(A))$$
$$\text{else } inf\{srecip(A)\} = \text{undefined}$$

Similar rules are used for suppp, infff, hival and loval.

(12) With the introduction of the srecip operator (above), terms in expressions may become undefined. This is handled using an 'undefined' value in the [value, V] construction. One implication of this is that expressions involving undefined values

also have undefined value. Hence, the bounding, simplification and arithmetic routines reduce any expression in undefined terms to be undefined.

An exception occurs when taking the maximum (or minimum) of a list of terms, whereby any undefined values are removed and the maximum applies over the remaining terms. If no terms remain, the result is undefined. The reason for the exception is the splitting of products described in (11) produces additional bounds that the CMS groups into max or min expressions.

## 8.0 Parity Determination

Brooks [Brooks 1981] did not detail the parity determination function, a version of which is now described. Some extensions (denoted by "*") have been made and others probably remain. This function returns the parity if it can be determined, otherwise it fails.

| Expression | Rule |
|---|---|
| [value, A] | if A > 0, then return(positive)<br>else return(negative) |
| [sqrt, A] | return(positive) |
| [variable, X, Exp] | if even(Exp) then return(positive)<br>else if odd(Exp) and Exp > 1 *<br>then return(parity([variable,X,1]))<br>else (Exp=1 case) if loval ≥ 0<br>then return(positive)<br>else if hival < 0 then return(negative)<br>else fail |
| [times, A, B] | if (parity(A)=positive<br>and parity(B)=positive<br>or parity(A)=negative<br>and parity(B)=negative)<br>then return(positive)<br>if (parity(A)=positive<br>and parity(B)=negative<br>or parity(A)=negative<br>and parity(B)=positive)<br>then return(positive) |
| [plus, A, B] | if parity(A)=positive<br>and parity(B)=positive<br>then return(positive)<br>if parity(A)=negative<br>and parity(B)=negative)<br>then return(negative)<br>if loval(A) + loval(B) ≥ 0 *<br>then return(positive)<br>if hival(A) + hival(B) < 0 *<br>then return(negative) |
| [recip, A] | return(parity(A)) |

| | |
|---|---|
| [srecip, A] | if parity(A) = positive return(positive)<br>else fail * |
| [min, List] | return(positive) if parity of all members in List is positive<br>return(negative) if parity of any member in List is negative |
| [max, List] | return(negative) if parity of all members in List is negative<br>return(positive) if parity of any member in List is positive |

## 9.0 Rough Bounds

This describes the upper/lower and hival/loval functions.

The upper/lower functions act over variables and are as Brooks described them - upper forms the min of all the symbolic upper bounds on the variable and returns this after simplification. Lower returns the simplified max of all the symbolic lower bounds.

The hival/loval functions estimate a rough numerical bound for expressions. Some extensions have been added (denoted by *). Their reasoning is:

| Expression | hival |
|---|---|
| [value, A] | A |
| [variable, A, Exp] | if even(Exp)<br>let H = hival([variable, A, 1])<br>L = loval([variable, A, 1])<br>hival = max(H^Exp, L^Exp) *<br>if Exp = 1<br>let Set = set of all numerical upper bounds on the variable<br>hival = minimum(Set + p_infinity)<br>else<br>let H = hival([variable, A, 1])<br>hival = H^Exp |
| [plus, A, B] | hival(A) + hival(B) |
| [recip, A] | let H = hival([variable, A, 1])<br>L = loval([variable, A, 1])<br>if H < 0 or L > 0 then hival = 1/L<br>else hival = p_infinity |
| [srecip, A] | let L = loval([variable, A, 1])<br>if L > 0 then hival = 1/L |

```
                        else hival = undefined

[min, List]             minimum(hival of each member of List)

[max, List]             maximum(hival of each member of List)

[sqrt, A]               sqrt(hival(A))

[times, A, B]
                        let HA = hival(A) and LA = loval(A)
                            HB = hival(B) and LB = loval(B)
                        hival = maximum(HA*HB,HA*LB,LA*HB,LA*LB)
```

| Expression | loval |
|---|---|

```
[value, A]              A

[variable, A, Exp]      if even(Exp)                                    *
                                let H = hival([variable, A, 1])
                                    L = loval([variable, A, 1])
                                if H < 0 then loval = H^Exp
                                else if L > 0 then loval = L^Exp
                                else loval = 0
                        if Exp = 1
                                let Set = set of all numerical
                                          lower bounds on the
                                          variable
                                loval = maximum(Set + n_infinity)
                        else
                                let L = loval([variable, A, 1])
                                loval = L^Exp

[plus, A, B]            loval(A) + loval(B)

[recip, A]
                        let H = hival([variable, A, 1])
                            L = loval([variable, A, 1])
                        if H < 0 or L > 0 then loval = 1/H
                        else loval = n_infinity

[srecip, A]
                        let H = hival([variable, A, 1])
                            L = loval([variable, A, 1])
                        if L > 0 then loval = 1/H
                        else loval = undefined

[min, List]             minimum(loval of each member of List)

[max, List]             maximum(loval of each member of List)

[sqrt, A]               sqrt(loval(A))

[times, A, B]
                        let HA = hival(A) and LA = loval(A)
                            HB = hival(B) and LB = loval(B)
                        loval = minimum(HA*HB,HA*LB,LA*HB,LA*LB)
```

## 10.0 Unimplemented features and Areas for Extensions

At present, the only feature of the original ACRONYM CMS left unimplemented is the 'trig' function used for bounding the 'cos' and 'sin' functions. This function is particularly weak anyway, since it only produces numerical bounds for the function based on the numerical bounds on its argument. Some extensions of parity and simplification remain for these functions.

No code has been implemented for the wraparound variable types.

Logic can be developed to bound terms of the form "$a*x^2 + b*x$" and "$a*\cos(x) + b*\sin(x)$". This is straightforward, but somewhat time consuming.

## 11.0 Messages, Causes and Actions

Several messages and actions can occur as a result of operation of the CMS, as detailed in the subsections below.

### 11.1 Prolog Messages

"! Out of global stack during execution." - cprolog has run out of work space, so needs a larger "-g" option, like "-g4000".

See section 13.

### 11.2 Constraint Checking

All declarations checked - means all variable declarations are correctly formed.

Bad declaration of X as: Y - means the variable X is not properly defined

Previous declaration of: X - means variable X has been previously defined; perhaps two sets of constraints use the same variable name

All constraints checked - means all defined constraints are correctly formed.

X is not a valid atom - means that a name was expected where X was encountered in the midst of a '[variable, X, Exp]' structure.

Relation X is not valid - means that X was neither 'lesseq' or 'greatereq', which are the two allowable relation types.

Variable X not declared - means X was not previously declared of an appropriate type

Expression: X bad in constraint: Y - reports that subexpres-

sion X is improperly formed in the context of the whole expression Y

These checks do not cause any actions, but the constraints should be fixed before the CMS is invoked.

### 11.3 Arithmetic Messages

These report arithmetic errors resulting from the designated type of invalid inputs. The routines return a sensible but usually incorrect answer. The messages are:

all exponents must be positive numbers

sqrt of negative number

cosine of plus infinity attempted

cosine of minus infinity attempted

sine of plus infinity attempted

sine of minus infinity attempted

### 11.4 Constraint Sets

***No upper bounds given for: X
***No lower bounds given for: X

This means that variable X has not been explicitly given a bound of the appropriate type. Either +infinity or -infinity gets chosen.

### 11.5 CMS Limitations

***No simplifier for: X

This means X is an expression that cannot be simplified, yet maybe could be, if the simplifier were extended. The original expression is returned. Please report the expressions causing this message.

***No parity expression for: X

The type of expression X was not understood well enough to deduce its parity (or even if unknown parity). The parity deducer should be extended. Unknown parity is returned. Please report the expressions causing this message.

*** CMS ERROR (please report)

The rule evaluator has failed for some reason. Please report the expressions causing this message.

## 12.0 Program Modules

This section briefly describes the contents of each file needed for the CMS:

arithmetic - arithmetic operations, predicates and relations
bounds - determines numeric and symbolic bounds for
expressions, implementing upper, lower, hival and and
loval functions
checkconst - checks the constraints for correct form
debugging - debugging functions
inf - implements the inf rules
inff - implements the inff rules
infff - implements the infff rules
interface - loading and unloading constraints, invoking the CMS
loadcms - loads all needed source files
parity - implements the parity function
setops - set operations
simp - the expression simplifier
sup - implements the sup rules
supp - implements the supp rules
suppp - implements the suppp rules
utils - minor misc. functions

## 13.0 General Comments

The CMS is generally slow, so it is best to not apply it often or when real-time operation is needed (or use a fast machine). For example, the simple test case shown in section 5.0 required 0.17 sec on a GEC 63 under cprolog. Larger problems can easily take more than a minute per bounding.

Complex problems require more storage allocation because of the depth to which the CMS can recurse during operation (e.g. 50 levels). Hence, cprolog should be invoked at least as:

```
cprolog -g2000
```

## 14.0 Example

See the listing in section 5.0. >From the tracing it is easy to identify the rules applied in the process of correctly finding the sup of variable a over the constraints:

$2 \leq a \leq 4/b$
$1 \leq b \leq 4/a$

## 15.0 References

[Brooks 1981] Brooks, R. A., "Symbolic Reasoning Among 3D Models and 2D Images", Stanford University AIM-343, STAN-CS-81-861, 1981.