LOGICAL ARITHMETIC

John G. Cleary [1]
The University of Calgary, Alberta, Canada.

Abstract

In the past, implementations of real arithmetic within logic programming have been non–logical. Difficulties include an inability to alter the order of execution of statements and incorrect handling of the different results caused by finite precision arithmetic. Using interval analysis a simple description of real arithmetic is possible. This can be translated to an implementation within Prolog. As well as having a sound logical basis the resulting system allows a very concise and powerful programming style and is potentially very efficient.

I Introduction

Logic programming aims to use sets of logical formulae as statements in a programming language. Because of many practical difficulties the full generality of logic cannot (yet) be used in this way. However, by restricting the class of formulae used to Horn clauses, practical and efficient languages such as Prolog are obtained. One of the main problems in logic programming is to extend this area of practicality and efficiency to an ever wider range of applications [Kowalski, 1979]. This paper considers such an extension for arithmetic.

To see why arithmetic as it is commonly implemented in Prolog systems is not logical consider the following example:

X = 0.67, Y = 0.45, Z is X*Y, Z = 0.30

This uses the notation of the 'Edinburgh style' Prologs [Clocksin, 1981]. (For the moment an underlying floating point decimal arithmetic with two significant digits is assumed). The predicate 'is' assumes its righthand side is an arithmetic statement, computes its value, and unifies the result with its lefthand side. In this case the entire sequence succeeds; however, there are some serious problems. The first is that the answer Z = 0.30 differs from the correct infinite precision answer Z = 0.3015. This causes difficulties in long sequences of arithmetic operations, where the propagation of such errors can lead the final result to have little or no resemblance to the correct answer.

The second problem arises because in a pure logic program the order of statements should be irrelevant to the correctness of the result (at worst termination or efficiency might be affected). This is not true of the example above. The direction of execution of 'is' is strictly one way so that

Y = 0.45, Z = 0.30, Z is X*Y

will deliver an error when X is found to be uninstantiated inside 'Z is X*Y'. Even if some form of arithmetic were devised capable of binding X it is not clear what value it should be bound to. For, if the underlying arithmetic rounds the results of operations, both the following sequences of statements should succeed:

X = 0.66, Y = 0.45, Z = 0.30, Z is X*Y
X = 0.67, Y = 0.45, Z = 0.30, Z is X*Y

So, it is unclear which value should be given to X, 0.66 or 0.67.

The problem then, is to implement arithmetic in as logical a manner as possible while still making use of efficient floating point arithmetic. The solution has three parts. The first is to represent Prolog's arithmetic variables internally as intervals of real numbers. So the result of 'Z is 0.45*0.67' is to bind Z to the interval (0.30,0.31). This says that Z lies somewhere in the interval, which is certainly true and probably as informative as possible given finite precision arithmetic. (Note that Z is *not* bound to the data structure (0.30,0.31); this is a hidden representation in much the same way that pointers are used to implement logical variables in Prolog but are not explicitly visible to the user. Throughout this paper brackets such as (...) or [...] will be used to represent open and closed intervals, not Prolog data structures.)

The second part of the solution is to translate expressions such as 'Z is (X*Y)/2' to relational form, in this case to 'multiply(X,Y,T), multiply(2,Z,T)' where 'T' is an extra variable introduced as part of the translation. Both the * and / operators have been translated to 'multiply' (with parameters in a different order). In the final implementation this relational form is insensitive to which parameters are instantiated and which are not. The third part is to provide a control predicate called 'split' to guide an iterative search for solutions.

The resulting system not only provides a correct logical form of arithmetic but is sufficiently powerful to be able to solve equations such as '0 is X*(X–2)+1' directly. This is an example of the fact that in logical arithmetic most expressions are invertable; they can be used to evaluate the expression or to solve for an unknown variable. As in standard Prolog the underlying implementation provides a weak problem solving technique which is sometimes sufficient in itself or can be used to program more powerful techniques.

Interval arithmetic has traditionally been treated as a functional rather than a relational theory. (Alefeld,1983) provides a good introduction to this 'classical' theory. Such functional theories are able to compute values but are unable to invert or solve equations. They also have difficulties in defining some operations such as division, a problem which does not arise in logical arithmetic (*ibid*, p5). A number of computer packeages for such interval arithmetic have been implemented (Cole,1982), (Yohe, 1979). Bundy [1984] has implemented an interval arithmetic package in Prolog. Like the previous systems mentioned it is functional and was built to provide correct information about the ranges of functions with a major application in qualitative physical reasoning. Bundy's work differs from that here in that it provides no way of representing logical real variables and as it is functional rather than relational it does not provide any technique for solving equations. Like Bundy (and unlike most interval systems) I carefully distinguish open and closed intervals. Bundy also used rational arithmetic whereas I seek to correctly handle the problems of finite precision real arithmetic such as that commonly available in hardware.

The next section gives a somewhat more formal description of the underlying interval arithmetic used. Section III shows how to describe primitive predicates in the arithmetic and gives effectively computable procedures for a number of them. Section IV

shows that it is necessary to provide extra control if the arithmetic is to be used invertably to solve equations. Section V examines a number of example programs and analyses their usefulness and convergence properties. Section VI concludes with a summary and some remarks on implementing logical arithmetic.

II. Interval Representation

So that lower and upper bounds can be operated on as single entities, an interval will be treated as a pair of bounds. Each bound has an attribute of being open or closed and an associated number. For example the half open interval (0.31,0.33] will be treated as the pair $0.31_($ and $0.33_]$ written as $\langle 0.31_(, 0.33_] \rangle$. (The brackets are subscripted to minimize visual confusion.) As well as the usual real numbers ∞ and $-\infty$ will be used as part of bounds. Let the set of reals be $\mathbb{R}$, then the set of upper bounds, $U(\mathbb{R})$, is defined as all the bounds of the form $x_]$ and $x_)$ where x ranges over the reals including ∞ but excluding $-\infty$. The set of lower bounds, $L(\mathbb{R})$, is similarly defined as all bounds of the form $x_[$ and $x_($ where x ranges over the reals including $-\infty$ and excluding ∞. The set of all intervals, $I(\mathbb{R})$, is the product of these two sets $L(\mathbb{R}) \times U(\mathbb{R})$. The notation for these objects is: intervals will be written as *I*, *J* or *K*; bounds within intervals will be written as g or h as in $\langle g,h \rangle$; the real part of a bound will be written as x or y and the bracket as u or v, as in x_u; and logical variables within Prolog programs will be written as X, Y or Z.

Using this notation loosely, intervals will be identified with the appropriate subset of the reals. For example, the following identifications will be made:

$$[-\infty,1.5) = \langle -\infty_[, 1.5_) \rangle = \{x: x < 1.5\}$$

$$[-\infty,\infty] = \langle -\infty_[, \infty_] \rangle = \mathbb{R}$$

and $(-13,2.4] = \langle -13_(, 2.4_] \rangle = \{x: -13 < x \leq 2.4\}$.

(These notations will be used interchangeably as convenient).

It is useful to have a linear ordering on all the bounds in $L(\mathbb{R}) \cup U(\mathbb{R})$:

$$x_u < y_v \leftrightarrow (x<y) \vee (x=y \wedge u<v)$$

where x<y is the usual ordering on the reals extended to include ∞ and $-\infty$ and where u<v is determined by the order $x_) < x_[< x_] < x_($. This ordering on the brackets is carefully chosen so that intervals such as (3.1,3.1) map correctly to the empty set. The general rule is that an interval with a lower bound g and upper bound h (written as the pair $\langle g,h \rangle$) will be empty iff h<g. For example, according to the definition above $3.1_(> 3.1_)$, so (3.1,3.1) is, correctly, computed as being empty. The definition also allows the intersection of two intervals to be computed using

$$\langle g,h \rangle \cap \langle g',h' \rangle = \langle \min(g,g'), \max(h,h') \rangle .$$

The finite arithmetic available on computers is represented by a finite subset, $\mathbb{S}$, of the reals. For example, $\mathbb{S}$ might be the set of numbers representable by a fixed length exponent and mantissa floating point format, a bounded set of integers or a bounded set

of rational numbers. In what follows it is only necessary to assume that $0,1 \in \mathbb{S}$. The set of intervals over $\mathbb{S}$ is $\mathbb{I}(\mathbb{S})$ defined as above for $\mathbb{I}(\mathbb{R})$.

It is useful to have a function, 'approx', from subsets of $\mathbb{R}$ to $\mathbb{I}(\mathbb{S})$ which associates with each subset the smallest interval in $\mathbb{I}(\mathbb{S})$ which contains it. Because of the availability of ∞ and $-\infty$ in the bounds such an interval is always uniquely defined. 'approx' can also be used as a mapping from $\mathbb{I}(\mathbb{R})$ to $\mathbb{I}(\mathbb{S})$, for example, 'approx([3.141592...,3.141592...])=(3.1,3.2)', that is, the best interval approximation to π in $\mathbb{I}(\mathbb{S})$ is (3.1,3.2).

Intervals are introduced into logic by extending the notion of unification. A logical variable X can be bound to an interval I, written X:I. Unification of X to any other variable Y gives the following results:

- if Y is unbound then it is bound to the interval I, written as Y:I;
- if Y is bound to the interval J then X and Y are rebound to the same interval $I \cap J$ (the unification fails if $I \cap J$ is empty);
- a constant C is equivalent to the interval 'approx([C,C])';
- if Y is bound to anything other than an interval the unification fails.

Note that unlike standard Prolog a variable may be rebound many times, although each time the length of the interval it is bound to will be shorter. This is analogous to the situation where a variable is bound to a partially uninstantiated data structure, the components of which are later bound.

Following are two simple Prolog programs and the bindings that result when they are run (assuming two significant digits of accuracy):

X = 3.141592	X:(3.1,3.2)
X > –5.22, Y ≤ 31, X=Y	X:(–5.3,31] Y:(–5.3,31]

III Primitive predicates

In order to develop a useful arithmetic it is necessary to specify some primitive arithmetic predicates. To do this I first develop a general way of specifying such predicates as subsets of the n-dimensional reals, $\mathbb{R}^n$, and I then show effectively executable routines for implementing them. The primitive predicates defined are addition, multiplication, ordering, inequalities and integer. The techniques developed are sufficient, however, to handle a much wider range of predicates.

Addition

Addition is implemented by the predicate 'add(X,Y,Z)' which says that Z is the sum of X and Y. 'add' can be viewed as a subset of $\mathbb{R}^3$ defined by:

$$\text{add} \equiv \{ \langle x,y,z \rangle : x,y,z \in \mathbb{R} , x+y=z \}$$

Given an initial binding of variables to intervals, X:I, Y:J and Z:K, the full set of solutions of the constraint 'add(X,Y,Z)' is given by all triples of real numbers in the set 'add $\cap\, I \times J \times K$ '. To get an effective representation of this set it is necessary to approximate it while ensuring that no triples satisfying the constraints are excluded. The

underlying representation of variables only allows them to be bound to intervals, so the only way to achieve this approximation is to *narrow* the intervals to which X, Y and Z are bound. As a preliminary to giving a rigorous procedure for doing this, Figure 1 illustrates this process of narrowing. The initial bindings are X:[0,2], Y:[1,3] and Z:[4,6]. After applying 'add(X,Y,Z)' the smallest possible bindings are X:[1,2], Y:[2,3] and Z:[4,5]. Note that all three intervals have been narrowed.

To get the best approximation the intervals should be narrowed as much as possible, while ensuring that no triples satisfying the constraints are excluded. That is, if the new bindings after narrowing are $X{:}I'$, $Y{:}J'$, and $Z{:}K'$ they must obey:

$$I' \times J' \times K' \supseteq \text{add} \cap I \times J \times K$$

The best values of I',J' and K' can be determined from the projected values of the set 'add $\cap$ $I \times J \times K$ '. For example, I' must be a superset of the projection on the x–axis :

$$I' \supseteq \{x: \exists\, y,z \langle x,y,z\rangle \in \text{add} \cap I \times J \times K\}$$

Each such projected set uniquely defines a smallest interval which contains it (this is true of any subset of the reals provided ∞ and $-\infty$ are available as bounds). Extending the domain of the function 'approx' introduced earlier I will write

$$I' \times J' \times K' = \text{approx}(\text{add} \cap I \times J \times K)$$

to indicate that I',J' and K' are the unique smallest intervals determined in this way.

If narrowing is done more than once for a single constraint then it can be proven that the intervals change only once on the first narrowing (space does not allow the proof to be given here). So it is not necessary to repeat the narrowing calculations more than once for a particular constraint although it will be seen later that the existance of other constraints may force further execution of the narrowing calculation.

These results about 'add' depend in no way on its properties other than that it is a subset of $\mathbb{R}^3$. The analysis and results can trivially be extended to any subset of $\mathbb{R}^n$. However, 'add' does have the property that all projections of 'add $\cap$ $I \times J \times K$ ' are intervals themselves. I will refer to this property as *interval convexity* or convexity for short. If a set is convex then the calculation of optimum narrowings for it is greatly simplified. Unfortunately, some sets such as 'multiply' are not convex. A technique for dealing with them is described in the next sub–section.

Because 'add' is convex it is only necessary to compute the end points of the projected sets when narrowing. To see how to do this consider the variable Z in 'add(X,Y,Z)'. The narrowed binding K' is known to be a subset of K and also of the sums of all the possible values in I and J. That is:

$$K \supseteq K'$$

and $$I+J \supseteq K'$$

where $$I+J \equiv \{z: \exists\, x \in I, y \in J, x+y=z\}$$

It can be shown that these two constraints completely determine K', that is:

$$K' = K \cap (I+J)$$

$I+J$ can be computed easily from the end points of its intervals as follows:

$$\langle g,h\rangle + \langle g',h'\rangle = \langle g+g',\ h+h'\rangle$$

where addition of the bounds is defined by

$$x_u+y_v = (x+y)_{u+v}$$

and x+y is the usual real addition extended to include ∞. This and addition of brackets (u+v) are defined in the two tables below. The entries marked ? do not occur and so need not be specified.

+	$-\infty$	x	∞
$-\infty$	$-\infty$	$-\infty$	?
y	$-\infty$	x+y	∞
∞	?	∞	∞

+	)	[	]	(
)	)	?	)	?
[	?	[	?	(
]	)	?	]	?
(	?	(	?	(

Similarly for the other two variables:

$$J' = J \cap (K-I)$$

$$I' = I \cap (K-J)$$

where $K-J \equiv K+(-J)$

and $-J \equiv \{-y: y\in J\}$

$-J$ can be calculated using $-\langle l,h\rangle = \langle -h,-l\rangle$ and $-(x_u)= (-x)_{-u}$ where $-x$ is the usual negation of real numbers extended to include ∞. The table for negating the brackets is:

u	(	[	]	)
-u	)	]	[	(

Multiply

The predicate 'multiply' can be defined in the same way as 'add' but because it is not interval convex a different approach is needed when constructing a narrowing algorithm for it. The definition of 'multiply' as a subset of $\mathbb{R}^3$ is

multiply $\equiv \{ \langle x,y,z\rangle : x,y,z \in \mathbb{R}\ ,\ x*y=z \}$.

To see that this is not convex consider the following example:

multiply(X,Y,Z) X:[–2,3] Y:[$-\infty$,∞] Z:[1,1]

In functional form this is 'Y is 1/X'. Taking the smallest intervals containing the projections of the constrained set 'multiply $\cap$[–2,3] × [$-\infty$,∞] × [1,1]', there is no change after narrowing. However, this does not tell the whole story, as Y cannot take on the value 0 (no real number between –2 and 3 multiplied by 0 can give 1). In fact, there is a gap of values from –1/2 to 1/3 which Y cannot validly take on. That is, the projection

of the constrained set onto the y–axis is not an interval, but is the union of the two intervals $[-\infty,-1/2]\cup[1/3,\infty]$. One way to ensure the maximum possible amount of narrowing in this case is to bind Y separately to the two intervals by backtracking. When this is done two different narrowings are obtained:

X:[–2,0] Y:$[-\infty,-1/2]$ Z:[1,1]

and X:[0,3] Y:$[1/3,\infty]$ Z:[1,1]

(note that both X and Y are bound to different intervals).

Such backtracking can be placed on a firmer footing by noting that the set 'multiply' can be partitioned into two separate sets each of which is convex, viz:

$$\text{multiply} = \text{mult}^+ \cup \text{mult}^-$$

where $\text{mult}^+ \equiv \{ \langle x,y,z\rangle : x,y,z \in \mathbb{R} , x\geq 0, x*y=z \}$

and $\text{mult}^- \equiv \{ \langle x,y,z\rangle : x,y,z \in \mathbb{R} , x<0, x*y=z \}$.

(There are many possible such partitions, the one chosen is as good as any other).

The narrowing algorithm for 'multiply' then selects one of the subsets of 'multiply' and narrows using just that set; eventually after backtracking the other set will be used to do the narrowing. The narrowing algorithm for 'multiply' given below computes I', J', and K' given that

$I' \times J' \times K' = \text{approx}(\text{multiply} \cap I \times J \times K)$.

It is given without the rather lengthy proof of its correctness.

Narrowing algorithm for multiply

In order to simplify a number of special cases an initial check is done to see if any of I, J, or K are bound to [0,0].

if K = [0,0] **then** visit the following two bindings by backtracking

$I'= I\cap$ [0,0], $J'=J$, $K'=K=$[0,0]

$I'=I$, $J'=J\cap$ [0,0], $K'=K=$[0,0]

else if I=[0,0] **or** J=[0,0] **then**

$I'=I$, $J'=J$ $K'=K=$[0,0]

else narrow using mult^+ and mult^-, visiting each in turn by backtracking;

The narrowing algorithm for mult^+ is given below (the algorithm for mult^- is essentially identical apart from appropriate manipulations of signs):

$$I' = I \cap[0,\infty] \cap$$

$$(((K \cap [0,\infty])* \text{inv}(J \cap [0,\infty])) \cup ((-K \cap [0,\infty]) * \text{inv}(-J \cap [0,\infty])))$$

$$J' = J \cap \text{inv}((I \cap [0,\infty]) *K)$$

$$K' = K \cap ((I \cap [0,\infty]) * J)$$

where $I *J \equiv \{x*y: x\in I, y\in J\}$,

$\text{inv}(I) \equiv \{y: x\in I, x*y=1\}$,

and $-I$ is defined above for 'add'.

Because of the way it is used above, 'inv' need only apply to positive intervals not equal to [0,0]. It can be calculated using the following rules:

$$\text{inv}(\langle l,h \rangle) = \langle \text{inv}(l), \text{inv}(h) \rangle$$

$$\text{inv}(x_u) = \textbf{if } x = 0 \textbf{ then } \infty_{-u} \textbf{ else}$$
$$\textbf{if } x = \infty \textbf{ then } 0_{-u}$$
$$\textbf{else } (1/x)_{-u}$$

where 1/x is the usual real division and –u is taken from the tables for add.

Again, because of the way it used, $I*J$ always has I positive and not equal to [0,0] . It can be calculated using

$$\langle g,h \rangle * \langle g',h' \rangle = \langle \min((g*g',h*g'), \max(h*h',g*h') \rangle .$$

If x or y is zero then a special case occurs and $x_u*y_v = 0_{u\#v}$ as shown below:

#	)	[	]	(
[	)	(	)	(
(	)	[	]	(

If neither x or y is zero then $x_u*y_v = (x*y)_{u*v}$ where x*y is the usual real arithmetic extended to include ∞ . The two tables below show this and the multiplication of brackets u*v:

*	)	[	]	(
)	)	(	)	(
[	)	[	]	(
]	)	[	]	(
(	)	(	)	(

*	$-\infty$	y(<0)	y(>0)	∞
x(>0)	$-\infty$	x*y	x*y	∞
∞	$-\infty$	$-\infty$	∞	∞

Divide by zero

It is interesting to see what happens to our definition of 'multiply' when a divide by zero is attempted. In most computer systems this leads to some form of exception or error routine being executed. In relational notation 'X is Y/0' becomes 'multiply(X,0,Y)'. If X and Y are unbound then they become narrowed to X:[$-\infty$,∞] and Y:[0,0]. That is, X can take on any real value and Y must be 0.

Inequalities

Inequalities such as 'X ≥ Y' are readily accommodated. Given initial bindings X:⟨g,h⟩ Y:⟨g',h'⟩ and the constraint X ≥ Y then the new narrowed bindings are:

X:⟨max(g,h'),max(g',h)⟩

and Y:⟨min(g',h),min(g,h')⟩ .

As with other predicates both of X and Y may be narrowed at the same time. For example if initially X:[1,5], Y:[2,6] and 'X ≥ Y' then after narrowing X:[2,5] and Y:[2,5]. '>' and '≥' are both interval convex so it is not necessary to investigate more complex narrowings.

An inequality such as 'X ≠ Y' is more problematic. If X and Y are both bound to extended intervals, say X:[1,2] and Y:[0,2] then no narrowing is possible. All values of X and Y constitute possible solutions. If X were 1.5 then Y could be 1.495 or 0.5 and so on. The only case where some narrowing can be done is if one of X or Y is a point. If X:[1,1], Y:[0,2] and 'X ≠ Y', then Y can be split to the two intervals: Y:[0,1) and Y:(1,2].

Integers

A predicate to restrict a variable to integers is easily realized by taking the integers as a subset of $\mathbb{R}$ to define the predicate. Given X:⟨g,h⟩ and the constraint 'integer(X)' then the binding after narrowing is:

X:⟨$\lceil g \rceil$,$\lfloor h \rfloor$⟩

where $\lceil x^u \rceil \equiv i_[$, i is the smallest integer such that $i_[\geq x_u$,

and $\lfloor x^u \rfloor \equiv i_]$, i is the largest integer such that $i_] \leq x_u$.

Note that the definition is careful to ensure that for example $\lfloor 3) \rfloor = 2]$.

As an example, if X:(2,4.5] and 'integer(X)', then after narrowing the binding is X:[3,4]. 'integer' is not interval convex, so, the preceding example might be handled by the alternate bindings X:[3,3] and X:[4,4]. This does not seem to be useful in practice and could involve a very large number of backtracking steps for initial bindings such as X:[−∞,100).

IV Control

The theory so far provides a means for computing arithmetic functions in the usual way building up values from previously known ones. Interestingly, the definitions of the individual predicates given above are not sensitive to which parameters are instantiated. This gives some hope that it might be possible to solve equations merely by stating them. In this section I introduce an additional control predicate called 'split' which is necessary to make this practical. In the next section I examine the range of equations which are practically soluble using 'split' and narrowing.

Splitting

The example equation in Figure 2 shows why more control is needed over execution of predicates if equations are to be solved. The equation there is 'X+X is 10', which translates to the relational form 'add(X,X,10)'. Initially X:[$-\infty$,∞] and when the narrowing described above for 'add' is attempted nothing happens; X remains bound to the same interval. Splitting uses Prolog–like backtracking to examine different possible solutions to the equation. In this case X can be bound to the alternate intervals [0,∞] or [$-\infty$,o) and provided both are eventually tried no possible solutions will be excluded. Figure 2 shows that when X is bound to [0,∞] narrowing immediately further binds it to [0,10]. Further progress requires another split, this time into the two intervals [0,5] and (5,10]. [0,5] is used first and narrows to the solution [5,5]. Upon backtracking (5,10] narrows to the empty set, that is, no solution is possible. Finally [$-\infty$,0) is tried and it too fails by narrowing to the empty set.

Splitting is invoked by the control predicate 'split(X)'. Its effect is to cause the argument variable to be split whenever no further narrowing is possible by any of the primitive predicates. It is important to understand that a predicate such as 'add(X,X,10)' is not completed after one narrowing is done, it may have to be reinvoked for another narrowing whenever X is narrowed by some other predicate. Thus, in the simple program 'add(X,X,10), split(X)', normal Prolog control first invokes 'add', which executes its local narrowing algorithm, then executes 'split' which binds X:[0,∞). This causes the local narrowing algorithm for 'add' to be executed again, which is a departure from the normal Prolog control.

In situations where there are many primitive predicates including 'split' the control algorithm used is as follows:

- the local narrowing algorithms for any primitive predicates called are executed until no further narrowing occurs;
- then one of the current 'split' predicates is executed and the process of narrowing is started again;
- finally when there is no further splitting or narrowing to be done normal Prolog control resumes (any failure during narrowing causes a normal Prolog fail and backtrack).

Although very simple, this control is sufficient for all the examples considered in the next section.

The control predicate 'split' can be programmed many ways; all that is required is that the subintervals generated and visited by backtracking completely cover the original interval. The form used here splits the original interval in half. There are a number of spacial cases that need to be handled: when 0 is in the interval it is used as the split point; when ∞ or -∞occurs as a bound it is treated as if it were respectively the smallest or largest number in $\mathbb{S}$; and when the end points of the interval are adjacent members of $\mathbb{S}$, as in [x,y], then the three intervals [x,x], (x,y) and [y,y] are generated. More sophisticated algorithms have been used to good effect but they would unnecessarily complicate the following examples.

V. Example Equations

Square Root

Figure 3 shows how the square root of a number is computed using the equation 'X*X is Y' (it is assumed that all such functional forms are translated to relational form for execution). Two extra constraints are added: 'Y ≥ 0' which ensures that only the positive solution is found; and 'split(X)' which forces an accurate solution to be found for X. The system alternates between narrowing and splitting six times before converging to the best possible interval (1.414,1.415) (for this and subsequent examples decimal floating point arithmetic with four digit accuracy is used).

Given the weak and general nature of the narrowing algorithm for multiplication it is perhaps surprising that the calculation converges quadratically. That is, if the error before one cycle of narrowing and splitting is ε, then afterwards it is $\varepsilon^2/2$. This can be seen in the list of errors in Figure 3, which shows the difference from the final value for the upper and lower bounds in each iteration. Logical arithmetic also automatically takes care of a number of other programming details:

- the second iteration binds X:[1,2], this is equivalent to the normal practice when computing a square root iteratively of making the value of the argument itself the first estimate for the square root;
- the last two iterations squeeze the last possible drops of precision from the system by searching the region from 1.414 to 1.416;
- the system automatically terminates when the limits of precision are reached and no further narrowing is possible;
- if the constraint 'Y ≥ 0' is lifted then both the positive and negative solutions will be found by backtracking.

General polynomial

These properties can be extended to polynomials of arbitrary degree. Any polynomial will automatically generate all solutions by backtracking. A little care is needed however to ensure that convergence is as fast as possible and to prevent too many spurious solutions being generated by inaccuracies inherent in the equations. Consider, for example the equation $x^3-6x^2-7x-6=0$. It has a single root lying in the interval (7.104,7.105). If the program for solving it is stated as:

X * (X*(X–6) – 7) – 6 is 0, split(X)

then this interval is the only answer returned. However, the form that the equation is stated in can affect the accuracy of the answer. For example, if it is stated as:

X*X*X – 6*X*X – 7*X – 6 is 0, split(X)

then a number of other intervals near this, for example (7.103,7.104), are also returned as solutions. This does not invalidate the claim that interval arithmetic is logically correct, as this principle only ensures that correct solutions are never rejected. The moral here is that while different forms of an equation may be equivalent in infinite precision arithmetic thier computational properties and the number of potential solutions they offer may differ. The 'problem' of the second of the equations returning additional solutions is intrinsic to the equation itself; it is impossible to evaluate that form of the polynomial and be sure that

a zero does not lie in the interval (7.103,7.104). The first of the equations above is in Horner form which is already known from standard numerical analysis to be a good way of evaluating a polynomial. The general principle here is that if an equation is in a form that can be evaluated accurately then the number of potential solutions returned by splitting will be reduced.

The convergence for a polynomial is quadratic (as with the square root above) whenever the current interval being examined contains only a single root and the interval is small. This is satisfactory except when a polynomial has multiple roots or two roots very close together. Convergence is still guaranteed but may be very much slower. This can be remedied at the cost of a more complex program by standard techniques such as finding a zero of the derivative (α) and dividing the final equation by ($x-\alpha$).

Simultaneous Equations

The ability to solve equations merely by stating them carries over to simultaneous equations in a number of variables. For example the following linear equation is solved directly by narrowing and splitting:

```
X + Y is 1,
X - Y is 2,
split(X), split(Y).
```

Figure 4 shows the search tree for the solution. Just stating the equations in this way is entirely satisfactory for small problems. Unfortunately, the search for a solution takes time $O(2^n)$ where n is the number of variables in the equations. In the case of linear equations this can be reduced to time $O(n^3)$ by writing a Gaussian elimination program in logical arithmetic (or using other well known techniques for solving linear equations). Programming this way in logic still gives substantial advantages: the final solutions are given with guaranteed error bounds; if the problem has no solution then the equations fail; it is not necessary to code a special test to take care of this case; and in Gaussian elimination the final step of back substitution to obtain the final values for the variables can be omitted completely.

Integer divisors

Consider the problem of finding all integer divisors of some integer. This could be coded as:

```
div(X,Y,Z):- X*Y is Z, integer(X), integer(Y), integer(Z).
```

This is a little naive as it allows all of div(3,4,12), div(4,3,12), div(–3,–4,12) and div(–4,–3,12) as well as div(1,12,12) etc. To get a more realistic set X should be constrained to be greater than 1 and also less than Y, that is:

```
div(X,Y,Z):- X*Y is Z, X>1, X≤Y, integer(X), integer(Y), integer(Z).
```

Figure 5 shows the resulting search tree for the calls 'div(X,Y,12), split(X)'. The narrowing steps are shown in detail so that the alternation between the narrowing caused by 'multiply' and that caused by 'integer' can be seen. The execution time for the resulting algorithm is $O(\sqrt{n})$ where the divisors of n are being sought. This is what you would get using the simple algorithm of testing all numbers from $2...\sqrt{n}$ to see if they divide n.

modulus

A good example of the power of logical arithmetic is the following program for 'mod(X,Y,Z)' (where Z = X mod Y):

```
mod(X,Y,Z):– Y > 0, X is Y*N + Z, integer(N), Z ≥ 0, Z < Y.
mod(X,Y,Z):– Y < 0, X is Y*N + Z, integer(N), Z ≤ 0, Z > Y.
```

The two clauses allow for the cases where Y is positive and where it is negative (the remainder may then be negative). This has very much the flavor of a specification of 'mod' and is eminently executable as well. For example, given the call 'mod(23,5,Z), split(Z)', Z is first bound to [0,5) by the two inequalities, this allows N to be narrowed to (3.6,4.6] by 'multiply', this is further narrowed by 'integer(N)' to [4,4], and finally the binding Z:[3,3] is obtained. When the second clause is visited by backtracking it fails because of the inequality 'Y < 0'.

factorial

A factorial function can be straightforwardly defined as:

```
fact(0,1).
fact(1,1).
fact(N,R):– integer(N), N > 1, M is N–1, fact(M,Z), R is Z*N.
```

This works as expected for calls such as 'fact(2,Z)' where Z is bound to [2,2]. Interestingly, the inverse call 'fact(N,2)' also works correctly, binding N:[2,2]. Unfortunately, it then goes into an infinite loop searching for solutions with N=3, N=4 ... . This can be easily fixed by adding the redundant specification that R must be greater than N, giving a new final clause:

```
fact(N,R):– integer(N), N > 1, M is N–1, fact(M,Z), R is Z*N, R ≥ N.
```

The resulting program is fully invertable with a call of the form 'fact(N,Z)' taking time O(N) when Z is bound to a single integer.

VI. Conclusion

The examples above show that logical arithmetic has all the properties that have come to be expected from a logic programming language:

- the code is close to or identical with specifications;
- many programs are invertable or can be used in a number of ways;
- the resulting programs are concise;
- in casting arithmetic in a logical or relational style a very elegant theory of arithmetic has emerged.

The semantics of a Prolog program including this style of arithmetic differ from standard prolog in that an answer is not guaranteed to be correct; all that is guaranteed is that no possible answers will be excluded. For example, extracting the square root of 2 ended by binding the result to (1.414,1.415) an interval which contains an infinite number of reals only one of which is actually a solution. Similarly, it is possible that

intervals will be returned which contain no solution at all, although it will not be possible to tell this within the constraints of the finite accuracy arithmetic available.

I would like to suggest that logical arithmetic as described here could provide the lowest level of arithmetic available to a Prolog programmer. This requires that an efficient implementation be possible. The narrowing algorithms for the primitive predicates are obviously more complex than single floating point arithmetic operations but they should not be beyond modern microcoded systems, and within the decade would possibly fit on a single CPU chip. One problem for an interpreter is that standard floating point arithmetic instructions can be very awkward to use as they often do not provide sufficient control over the direction in which truncation and rounding occur (when writing my own system for this I handled my own exponents and mantissas directly as integers in order to get a correct implementation). There is some hope, however, that given the correct low level support the arithmetic could be very efficient. For example, in some cases it is possible to optimize the bindings resulting from narrowings as simple assignments without any need to record information on a backtrack trail.

Acknowledgments

I would like to thank Jon Rokne for introducing me to interval arithmetic and for his encouragement in this work and Ian Witten and Brian Wyvill for comments and suggestions. Support was received from the Natural Sciences and Engineering Research Council of Canada.

References

Alefeld, G. and Herzberger, J. (1983), "Introduction to interval computations," Academic Press, New York (translated from "Elemente der Intervallrechnung" by Rokne, J.).

Bundy, A. (1984), A generalized interval package and its use for semantic checking," *ACM Trans. on Mathematical Systems*, **10**(4)397–409.

Clocksin, W.F. and Mellish, C.S (1984), "Programming in Prolog," Springer Verlag, New York.

Cole, A.J. and Morrison, R. (1982), "Triplex: a system for interval arithmetic," *Software Practice and Experience* , **12**(4)341-350

Kowalski, R.A. (1979), "Logic for problem solving," Elsevier North Holland, New York.

add(X,Y,Z)

Initial bindings	X:[0,2]	Y:[1,3]	Z:[4,6]
X+Y=[0,2]+[1,3]			[1,5]
Z-X=[4,6]-[0,2]		[2,6]	
Z-Y=[4,6]-[1,3]	[1,5]		
Resulting narrowed bindings	X:[1,2]	Y:[2,3]	Z:[4,5]

Figure 1. Execution of narrowing algorithm for 'add'

Functional form: X+X is 10, solve(X)

Relational form: add(X,X,10), solve(X)

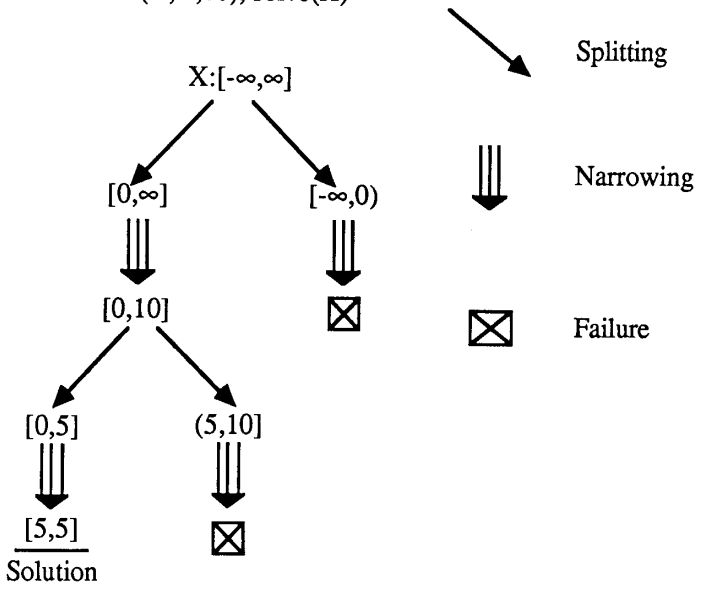

Figure 2. Example of use of narrowing and splitting to solve an equation

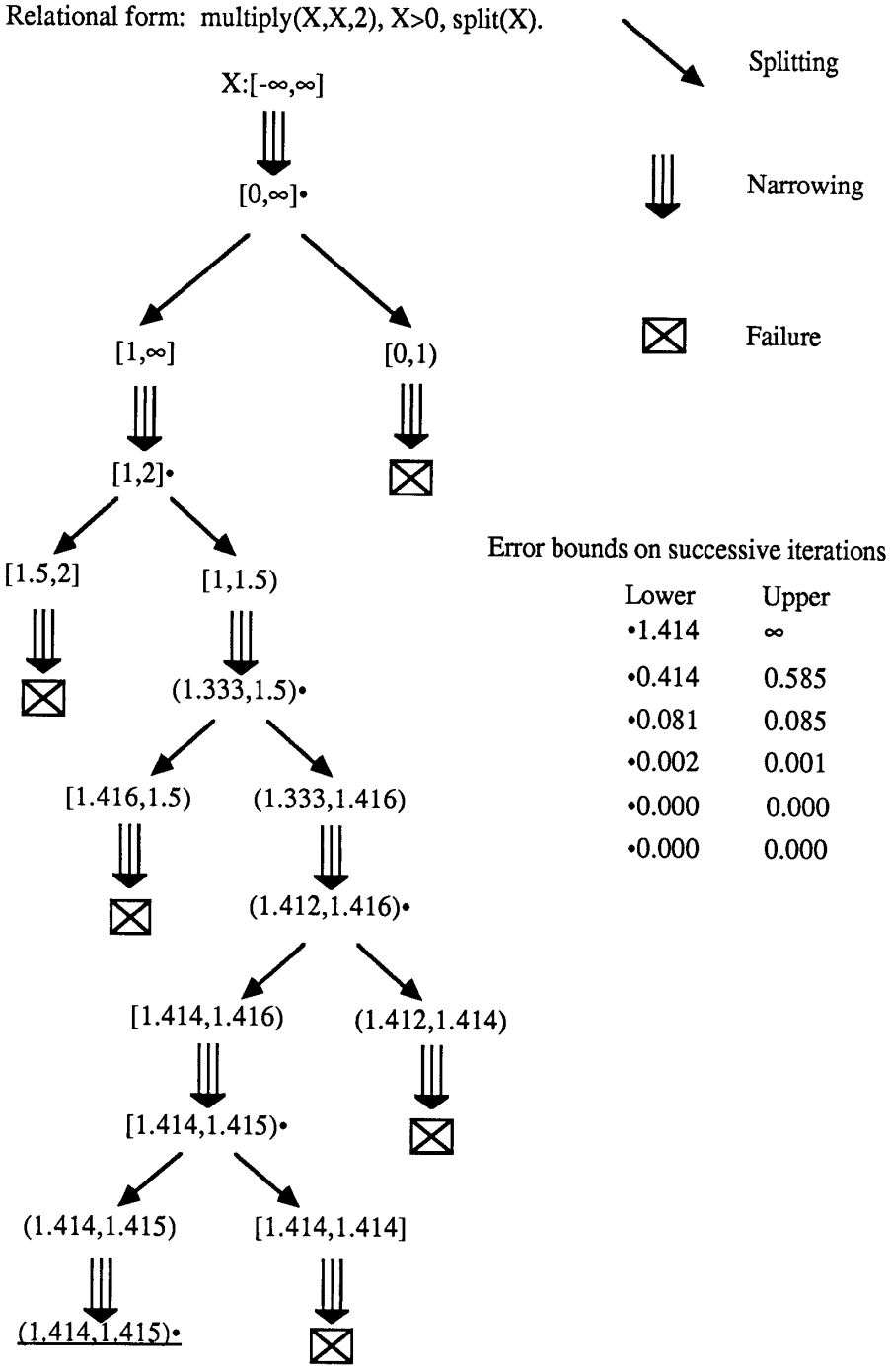

Figure 3. Solution of square root.

Functional form: X+Y is 1, X-Y is 2, split(X), split(Y).

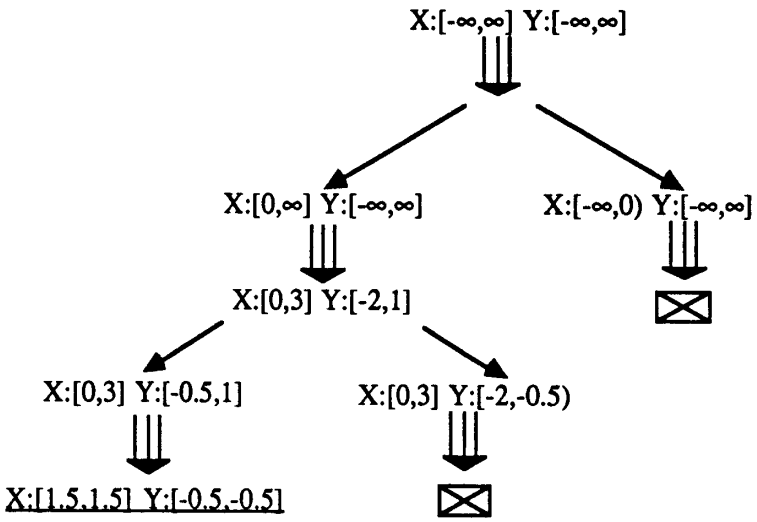

Figure 4. Linear simultaneous equation.

```
div(X,Y,Z):- X*Y is Z, X > 1, X ≤ Y,
             integer(X), integer(Y), integer(Z).
:-div(X,Y,12), split(X).
```

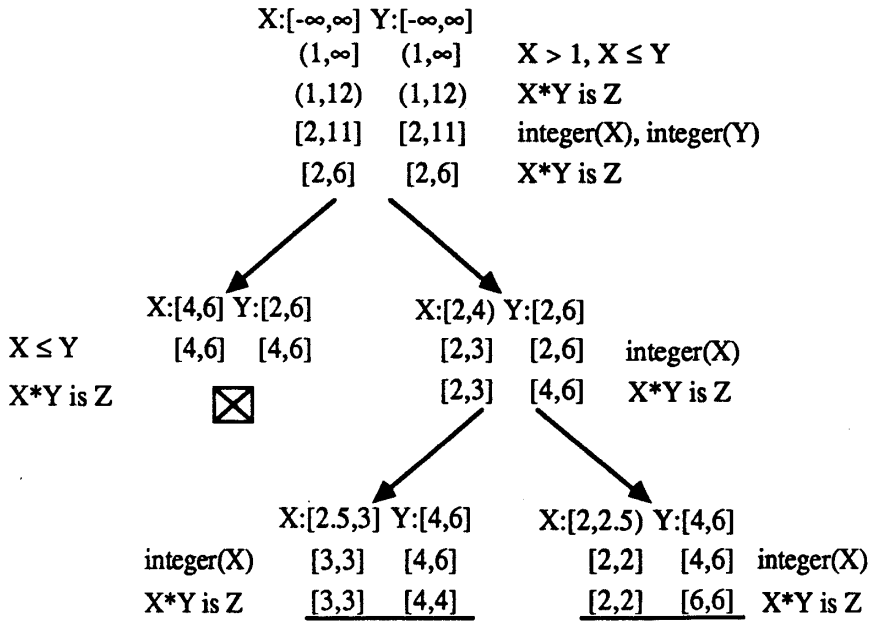

Figure 5. Integer divisors.