

Using Meta-inference with Prolog to Preserve Accuracy in Numerical Equation Solving

Gabriel Renaldo Laureano* Daniel Santana de Freitas[†]

27 September 2004

Abstract

In this paper, we explore the powerful bidirectional interface between Prolog and C to build an equation solver that mixes infinite precision and numerical methods. Meta-inference rules in Prolog control the numerical process in C, so that exact methods of solution are used preferably to numerical approximations, leading to the preservation of as much significant digits as possible during the computations. On the other hand, the possibility to access numerical resources when symbolic treatment fails allowed us to handle a much wider range of equations than with a purely symbolic system. Another advantage of this hybrid approach is the possibility of recording previous results to aid in future computations.

1 Introduction

All equations in one variable can be reduced to simple relations in the form $f(x) = 0$. What is needed is a means to obtain the values of x that turn this equality into a true statement. This task can be accomplished in two, very different, forms: by symbolic computing, which consist in algebraic manipulations based on the axioms of algebra, and by numerical methods. Algebraic manipulations use computational power to produce logical inferences from initial declarations: the programmer “declares” the problem and

*Training Program in Computer Science (PET/CCO), Department of Computer Science (INE), Federal University of Santa Catarina (UFSC), BRAZIL. <mailto:laureano@inf.ufsc.br>

[†]Department of Computer Science (INE), Federal University of Santa Catarina (UFSC), BRAZIL. <mailto:santana@inf.ufsc.br>

the computer finds a way to solve it [1]. This approach allows for infinite precision, since numbers can be analyzed syntactically and some basic operations, like $\text{sqrt}(x)$, need not be executed unless it is absolutely necessary. A method is defined as purely numerical if it uses only a finite sequence of the most elementary operations that a computer can do ($+$, $-$, $*$, $/$), executed over numbers stored in a finite memory. All these steps of a numerical procedure must be anticipated by the programmer. Since the total memory is finite, each number must be manipulated according to a series of size limitations. As a result, apart from a very small subset of the rationals, the numerical approach can only deal with numbers that are contaminated by roundoff errors [2].

This comparison involves lots of other aspects. For example, symbolic methods, when applicable, usually obtain all values of x that satisfy $f(x) = 0$ with infinite precision. Numerical methods always depend on start values and usually, unless some kind of exhaustive search is performed, obtain only an approximated value for one element of the set of possible solutions. This shows that, whenever precision needs to be high, symbolic computation is clearly superior to its numeric counterpart. Nevertheless, there is a serious drawback in this last affirmation, since the application of symbolic manipulation is restricted to certain classes of problems: those covered by the specific collection of methods implemented. Numerical methods can handle $f(x)$ in almost any functional form, because numerically $f(x)$ is treated as a black box. This means that the numerical approach is the only who can produce results in many real situations. However, sometimes, part of the solution comes from simple, tractable, polynomial equations, for example, and even in these cases there would be roundoff errors present.

In this work, we take advantage of Prolog's capability to access numeric processing in the middle of a resolution process to present an equation solver that combines the flexibility of numerical methods and the infinite precision associated with symbolic manipulation. The result is a system that produces as many roots as possible, because it has a numerical root finding module in C, and that uses infinite precision whenever it is applicable, in order to avoid roundoffs where an exact solution could be obtained. What makes it possible is the use of meta-level inference with Prolog, as proposed by Bundy and Welham [3], to try to manipulate algebraically the equation as much as possible, before calling a numerical method. Even when a root must be numerically obtained, control is maintained with Prolog, so that the possibility of infinite precision is checked for all roots. Our system would produce better results, for example, in any equation presented in the form of the product of a polynomial and a transcendental function, as in $p(x)*f(x) = 0$, for at least two of the roots of the polinomial factor would be calculated

with infinite precision. Another interesting aspect of our system is that we can take advantage of Prolog's characteristics to include new strategies of algebraic equation solving as soon as we become aware of them, in a kind of continuous learning about this subject.

The mixing of symbolic and numeric methods has been used in the literature in the domain of equation solving. Kocbach and Liska [4], for example, present a system that translates algorithms developed in the computer algebra system Reduce to programs in Fortran. In their case, the symbolic computation tools of Reduce are used only to check correctness of the implementation proposed by the programmer, by comparison with built-in facilities. After this preliminary step, code is generated in Fortran and results are numerically obtained. Although related to equation solving, their work did not assess accuracy issues.

2 Equation solving and Logic Programming

The symbolic part of our solver is a direct application of logic programming. Logic programming uses logic to represent knowledge and solves problems by using deduction to derive logical consequences in a computational manner. When expressed in this way, knowledge can be interpreted both declaratively and procedurally, giving rise to a very computationally effective form of information definition. The most simple way to use logic programming is to represent knowledge as Horn clauses and then to use SLD-resolution [5] to perform reasoning, in a kind of mechanical theorem proving. But logic programming can also be generalized to include negation by failure or to be seen as a goal-oriented reasoning with equations, leading to a computational interpretation of a greater variety of proof procedures [6].

Equation solving is a domain that is naturally suitable to the application of logic programming. The mathematical reasoning behind this task depends basically upon familiarity with the axioms of algebra and also upon the mathematician's ability to find the best strategy to apply them. This behavior can roughly be modelled as a collection of methods and a decision procedure [7]. Each method contains the rules of algebra needed to successively transform equations of a certain type, rewriting them until a solution is reached. The decision procedures are designed to guide the use of the methods, in order to obtain a sequence that reaches the solution in a finite number of steps. Early attempts to implement this strategy included decision procedures which consisted simply in the exhaustive application of rewrite rules from two basic sets: simplification and reduction [8]. This procedure had many problems, such as the impossibility to include ad hoc knowledge about the equation

being solved and the difficulty to select which type of simplification is to be applied in each case (see [3]).

Meta-level inference is a much better alternative than the successive application of only two kinds of rules. Bundy and Welham [3] describe meta-level inference as a technique in which inference is conducted at two levels simultaneously: the meta-level and object-level. The object level encodes the rules of algebra, while the meta-level is responsible for the encoding of control or strategic knowledge (in this case, methods of algebraic manipulation). They implemented their ideas in a program for algebraic manipulation called PRESS.

3 Solving symbolic equations with Prolog

PRESS (PRolog Equation Solving System) is a Prolog program for solving symbolic equations developed in the mathematical reasoning group of the Department of Artificial Intelligence at the University of Edinburgh. PRESS is completely defined by Bundy and Welham [3]. Object-level knowledge in PRESS consists of rewrite rules organized into several sets, associated with different kinds of rules, depending on syntactic characterization. The meta-level knowledge reasons about the job to be performed and selectively applies the rules. Without a numerical module, this program performs at the level of a third year high school student [7].

We used a simplified version of PRESS presented by Sterling and Shapiro [7] as the basis for our research. In the rest of this section, we present a brief description of this program. More details can be found in [7]. The main predicate is *solve_equation(Equation, Unknown, Solution)*, which is successful only if *Solution* is the solution to the equation *Equation* in the variable *Unknown*. At the second level of execution, this program contains clauses for checking and resolution. There is at this level, for example, a predicate to test if a certain equation is polynomial, followed by a predicate which can solve linear and quadratic polynomial equations.

This program can solve four basic types of equations, using an adequate set of predicates. Accordingly, we call this set of predicates a “module”. These modules are described below.

Module for factorization: This module solves equations of the type $Eq1 * Eq2 = 0$, $Eq1$ and $Eq2$ being equations whose type is not known yet. After having their types recognized, both equations are solved as if they were distinct, as in $Eq1 = 0$ and $Eq2 = 0$, and the solutions thus obtained are computed to $Eq1 * Eq2 = 0$. Each subequation involves a new call to *solve_equation/3*.

Module for Isolation: This module uses inspection structures (see [7]) to build a list that indicates the path to the relative position of the variable in the equation. For example, the equation $5 + x * 6 = 0$ would be translated to an infix notation as $(+(5, *(x, 6)), 0)$, from which the path to the variable would be computed as $[1, 2, 1]$, since the variable is located in the first argument of $= / 2$, in the second of $+ / 2$ and is the first argument of $* / 2$. The use of this list and the application of some axioms of algebra in the form of rewriting rules lead to the isolation of the variable from the rest of the equation, obtaining *Variable = Equation*.

Module for Polynomial Equations: After being recognized as such, the equation to be solved is transformed into a list, in which each element represents the coefficient and the degree of each polynomial term. For example, the equation $x^2 + 5 * x - 8 = 0$ is represented as $[(1, 2), (5, 1), (-8, 0)]$. Then resolution is accomplished simply by applying the adequate rule to each polynomial degree (note: according to our goals in this work, we have equipped this module only with the capability to solve until the second degree).

Module for Homogenization: uses a set of heuristics to identify transformation that could possibly lead the equation to a simpler form, hopefully a form with a known resolution. The simplified version deals only with exponential equations, but new modules for the recognition of other types of equations can be easily added. In section 4.1 below, we explain how this program can be viewed as a white box, allowing easy addition of new heuristics about trigonometric equations.

4 Improving the simplified version of PRESS

We decided to increase a little the scope of applications of the simplified version of Sterling and Shapiro. This was important to eliminate some limitations that would certainly affect the balance between numeric and symbolic resources. Our modifications are described below.

4.1 Dealing with trigonometric equations

Starting from the structure already built for the predicate *homogeneize/3* to treat special cases of exponential equations, several heuristics were created in order to also identify convenient rewritings of equations in trigonometric terms. So, if a term like $\cos(2 * x)$ shows up, it can be substituted by $1 -$

$2 * \sin^2 x$, whenever this is adequate. This would be the case if, for example, besides this term, there were only sines in the equation.

4.2 Improving performance

According to Sterling and Shapiro [7], the most called predicate in the simplified version of PRESS was *free_of/2*. They suggest that restyling this predicate could certainly improve performance.

The predicate *free_of(SubTerm, Term)* is successful if *SubTerm* is not found in *Term*. In the original version, *free_of/2* worked just as a front-end for *occurrence(SubTerm, Term, N)*, which counts the number of occurrences of *SubTerm* in *Term*. It was implemented like this:

```
free_of(SubTerm, Term) :- occurrence(SubTerm, Term, 0).
```

The problem with this strategy is that, to solve a call to *free_of/2* that terminates in fail, it is necessary a complete counting of all occurrences of *SubTerm* in *Term*. Fail would occur only at the return of the predicate *occurrence*, when Prolog would try the unification of *N* and 0. To avoid useless processing, we followed Sterling and Shapiro's directions [7] and wrote a specialized predicate for *free_of/2* that fails as soon as a single occurrence of *SubTerm* is met. The new *free_of/2* is showed below.

```
free_of(Term, Term) :- !, fail.
free_of(SubTerm, Term) :- compound(Term), !,
    functor(Term, _F, N), free_of(N, SubTerm, Term).
free_of(SubTerm, Term) :- SubTerm \= Term.
free_of(N, SubTerm, Term) :- N > 0, !, arg(N, Term, Arg),
    free_of(SubTerm, Arg), N1 is N - 1,
    free_of(N1, SubTerm, Term).
free_of(0, _Subterm, _Term).
```

We made tests with these five equations below for measure execution time, but we could not use the conventional ways because the execution is too fast, instead of we decided to measure time complexity in terms of number of predicates, including all those in a branch of failure.

$$5 * x^2 - 20 * x + 12 = 0 \tag{1}$$

$$\cos x * (\sin x - 4) = 0 \tag{2}$$

$$2^{2*x} - 5 * 2^{x+1} + 16 = 0 \tag{3}$$

$$\cos x * (\cos(2 * x) - 2 * \sin x) = 0 \tag{4}$$

$$\cos(2 * 5^{3*x+6}) - 2 * \sin(5^{3*x+6}) = 0 \tag{5}$$

The last two of them used the improvements discussed in the subsections above. We had an average increase in performance of 6.7%. Sterling and Shapiro [7] point out that this number could be as high as 35%. We credited this difference to the simplicity of our tests, since our implementation of *free_of/2* is not far from the one presented by Sterling and Codish in [9]. It should be noted, however, that this result of 6.7% is based on experiments that involved calls to the predicate *solve_equation/3*, which means that it represents a gain in performance for the system as a whole. Counting exclusively the number of predicates involved in the execution of *free_of/3* this result would be bigger.

5 Hybrid approach: symbolic-numerical integration with Prolog

In this section we describe the two forms of interaction between Prolog and C that we used in our work. Both use primitives of the GNU Prolog Compiler developed by Daniel Diaz and a more detailed description of all the primitives presented below can be found in his work [10].

5.1 Solving cubic polynomial equations

The most simple interaction happens when a Prolog program makes calls to routines written in C, a resource that we used in order to solve cubic polynomial equations. We have implemented symbolic predicates only to solve quadratic equations, so, to solve a cubic equation, the numeric module must be called. We then use Newton's iterative method to find a real root in the range [-10;10]. Once this root has been found, we use polynomial division to reduce the problem to a quadratic equation. This quadratic equation is then solved by symbolic predicates and we obtain the other two roots (real or complex) with infinite precision. The advantage of this approach is that we can guarantee that at least two of the three roots will have very good precision.

It should be noted that we could have used a purely symbolic technique to solve the third degree equation as well. But our goal is just to demonstrate a potential use for a hybrid approach and not to present a solver for third degree equations. Later in this paper, we indicate how the resolution of polynomial equations with greater degree could be accomplished using this technique.

This module was built in several steps. First, we added heuristics that can detect if the third degree polynomial . Then, we included the following

predicate to execute division of polynomials:

```

divide_polynomials([(C1,N1)|P1], [(C2,N2)|P2], [(C,N)|P], R) :-
    N1 >= N2, !, C is C1/C2, N is N1 - N2,
    multiply_single([(C2,N2)|P2], (C,N), P3),
    subtract_polynomials([(C1,N1)|P1], P3, P4),
    remove_zero_terms(P4,P5),
    divide_polynomials(P5, [(C2,N2)|P2], P, R).
divide_polynomials([(C1,N1)|P1], [(C2,N2)|P2], P, [(C1,N1)|P1]) :-
    P = [(0,0)], N1 < N2, !.
divide_polynomials([], P2, [], [(0,0)]).

```

The predicate *divide_polynomials*(*P1*, *P2*, *P*, *R*) is successful if *P* is the result of the division of *P1* by *P2* and *R* is the rest. Using a procedural reading [6], the predicates *multiply_single/3*, *subtract_polynomials/3* and *remove_zero_terms/2* perform, respectively: a multiplication of a polynomial by another with only one term, a subtraction of two polynomials and the purge of terms with zero coefficients.

Whenever it is guaranteed that a fourth degree polynomial equation has a real root, a technique similar to the above described can be applied. Two roots are obtained numerically and polynomial division reduces the problem to a quadratic equation whose solution is performed symbolically. In this way, at least two roots can be obtained with very high precision.

5.2 Preserving precision in the solution of transcendental equations

Our system uses meta-level inference to manage a solver with numerical features added to a symbolic module. Specifically, there is a predicate *foreign/2* in the Prolog source that serves as a directive which declares Newton's method, implemented as a routine in C [11], to be a foreign function: `:- foreign(newton(+term, -float)).` The C routine must possess the following signature: `Bool newton(P1Term equation, double *root).`

The C-routine contains Newton's method for root finding in a problem defined as $f(x) = 0$ and needs to execute several calls to the definition of the function. Each new call to the function involves the three steps. First a copy of the term must be created, so that its original value can be used in future calls. Using *Copy_Term/2*, *equation* is copied to another variable. After that, using the primitive *Rd.Compound_Check/3*, available in `gprolog.h`, the variable can be recovered to be unified with the numerical value. Unification will cause each occurrence of the variable in the equation to be automatically

substituted by the numerical value. For last the routine *Math_Load_Value/2* computes the value $f(x)$, which is represented in the copy of *equation*.

These steps describe the Prolog consult below, considering that *Equation* is instantiated:

```
?- equation(Equation, X), X = number, Y is Equation.
```

Note that, in the procedure described above, numeric processing is performed by the appropriated routine, but the calls are left to Prolog. Prolog treats equations as primitive types, allowing them to be used as simple parameters.

The utility of this hybrid approach can be best illustrated with the resolution of certain types of transcendental equations, namely, those that can be put in the form of a polynomial times another (simpler) transcendental function. For example, in the solution of the equation: $(x^3 - 2 * x^2 - 5 * x + 2) * (x \ln x - 1) = 0$ the meta-level creates two new problems: $x^3 - 2 * x^2 - 5 * x + 2 = 0$ and $x \ln x - 1 = 0$

The first equation will be solved as described above and the second by Newton's method in the numeric module, which will be called when all symbolic options fail. This procedure leads to at least two roots determined with very high precision, because they will be the result of symbolic computation. On the other hand, this allows all roots to be found, what would not be possible with the sole application of the symbolic approach to the same problem.

6 Conclusions

This paper described an equation solver written in Prolog that integrates symbolic and numeric modules with the aid of inference at the meta-level [3]. The symbolic part is a Prolog module that accounts for algebraic manipulation while the numeric part is a routine in C that can perform Newton's iterative method for root finding over any equation in one variable. We show the details of the interaction between Prolog and C that allow, for example, the equation to be passed to the numeric module as a symbolic parameter by the solver. The solver contains clauses at the meta-level which guide the resolution process so as to assure that each side of the interface between C and Prolog is responsible for doing what it is best prepared to do. As a result of this approach, exact (symbolic) methods are used as much as possible during a resolution process, leading to an increase in the overall precision of the equation resolution process. As a final remark, it should be noted that, due to the characteristics of Prolog, as soon as a result is obtained, it can

easily be recorded to aid in future computations, in a kind of learning process that can improve accuracy even further.

References

- [1] Malpas, J.: Prolog: A Relational Language and its Applications. Prentice-Hall Inc. (1987)
- [2] Cheney, W., Kincaid, D.: Numerical Mathematics and Computing. Brooks/Cole Publishing Co. (1994)
- [3] Bundy, A., Welham, B.: Using meta-level inference for selective application of multiple rewrite rule sets in algebraic manipulation. *Artificial Intelligence* **16** (1981) 189–212
- [4] Kocbach, L., Liska, R.: Generation and verification of algorithms for symbolic-numeric processing. *Journal of Symbolic Computation* **25** (1998) 367–382
- [5] Casanova, M.A., Giorno, F.A.C., Furtado, A.L.: Programação em Lógica e a Linguagem PROLOG. Editora Edgard Blücher Ltda (1987)
- [6] Kowalski, R.A.: The early years of logic programming. *Communications of the ACM* **31** (1988) 38–43
- [7] Sterling, L., Shapiro, E.: *The Art of Prolog: advanced programming techniques*. 2 edn. MIT Press series in logic programming. The MIT Press (1999)
- [8] Moses, J.: Algebraic simplification, a guide for the perplexed. 2nd Symposium on Symbolic Manipulation (1971) 282–304
- [9] Sterling, L., Codish, M.: Pressing for parallelism: A prolog program made concurrent. *Journal of Logic Programming* **3** (1986) 75–92
- [10] Diaz, D.: GNU PROLOG: A Native Prolog Compiler with Constraint Solving over Finite Domains. 1.7 edn. (2002)
- [11] Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: *Numerical Recipes in C: The Art of Scientific Computing*. 2 edn. Cambridge University Press (1992)