Running Programs Backwards:
The Logical Inversion of Imperative Computation

BRIAN J. ROSS
Brock University
Department of Computer Science
St. Catharines, Ontario, Canada L2S 3A1
phone: (905) 688-5550 ext. 4284
fax: (905) 688-3255
email: bross@sandbanks.cosc.brocku.ca

Submitted for publication.
February 2, 1998

Abstract

The feasibility of inverting imperative computations using logic programming technology is informally explored. Program inversion is the process of determining the possible input values that will produce a given output for a program. The relational semantics of imperative computations treats programs as logical relations over the observable environment, which is usually taken to be the state of the variables in memory. The declarative semantics of program relations implicitly denote both forward and inverse computations. This view of computation has practical applications when logic programming technology is considered. An imperative program has a logic program derived for it which abductively describes its inverted behaviour. Depending on the inference strategy used, execution of this relational program can compute the inverse of the imperative program. It is shown how a number of nontrivial imperative computations can be inverted with minimal logic programming tools. The advantage of this approach is that nondeterministic inversions are possible, which permits sets of inputs to be computed for a particular output. This should prove to be a useful tool in imperative program testing and debugging.

Key words: inversion, abduction, reverse computation, program testing

1 Introduction

Program inversion refers to the computational process which determines the input values a program needs in order to generate a given set of output values. With respect to programs written in conventional imperative languages, a program inverse is usually conceived as being a program that computes the inverse computation of another program, where both programs are normally written in the same imperative language. A shortcoming of this approach is that imperative languages are ill-suited to computing generalised inverse computations. Firstly, the execution of a destructive assignment can cause irrevocable loss of information – the previous value stored before the assignment occurred is gone forever after an assignment. Secondly, imperative programming constructs are deterministic, whereas the the inversions of many algorithms are nondeterministic. Hence only a subset of imperative programs have inversions implementable in imperative code in a direct fashion.

This paper proposes that imperative programs are indeed conducive to inverse computation if we loosen the requirement that the source and inverse programs are written in the same imperative language. It is well known from work in logical program semantics (eg. [LS84, Heh84b]) that an imperative program is modellable by a logical relation which declaratively describes the computational behaviour with respect to the observable environment. If the Horn clause subset of first-order logic is used as the semantic language, then a program's logical semantics can be interpreted as a logic program [Llo87, CvE81]. Because logic programs have associated models of computation, the logic program description of the imperative program can have an inference strategy applied to it, resulting in a simple interpreter for the language. The relevance of this to the program inversion problem is that, if the logic program model and its corresponding inference strategy are appropriately chosen, the inverted computations of many non-trivial imperative algorithms are readily computable. This follows because logic programs are nondeterministic, and their inference strategies permit search for the environment values that are normally lost in a strictly imperative environment.

Besides being a fundamental theoretical concept, program inversion has practical applications in interactive program testing and debugging. Imperative programming languages such as C, Pascal, FORTRAN, and COBOL continue to dominate programming in industry, and there remains a pressing need to develop new methods of analyses for this class of programs. The inversion techniques proposed in this paper make it possible to model imperative programs by logical relations which, with the application of appropriate logic programming techniques, permit the computation of inverse computations. Performing computational inversions on forward-executing imperative code is a novel and powerful testing and debugging tool.

Section 2 addresses the inversion problem in more detail, and reviews work on the topic. Section 3 presents a logic–based inversion methodology that can compute imperative program inverses. Some example program inversions are given in section 4. A discussion concludes the paper in section 5.[1]

[1] A preliminary version of this work appeared in [RW89].

2 The inversion problem

2.1 General overview

Inversion is a fundamental concept in mathematics and theoretical computer science. *Function inversion* is the process of deriving for some $1:1$ function $f : X \rightarrow Y$ the inverse function $f^{-1} : Y \rightarrow X$ such that $f^{-1}(f(x)) = x$. For example, let $f(x) = x^2$ and $f^{-1}(x) = \sqrt{x}$. Then

$$f(f^{-1}(x)) \;=\; (\sqrt{x})^2 \;=\; x$$

There are more complex mathematical inversions, for example, the integration and differentiation of equations in calculus. Even though the existence of f^{-1} may be certain, however, the derivation of an effective computation procedure or equation for any general f^{-1} is difficult. The existence of a decision procedure for computing inversions would enable the solution of Fermat's last theorem.

Program inversion is similar to its mathematical counterpart. A conventional view of computation is for a program to first read some initial input values, do some computation on them, and produce output values as a solution for the given input. On von Neumann architectures, this scheme is inherently forward-directed due to destructive assignments and deterministic control constructs. Program inversion is concerned with backward-directed computation. Given a final set of output values of a forward-directed program, an inverted computation derives the input values that the program would use to compute the output. For example, as above, if we have a program that computes the square of an input integer n, then the inverse computation of this program is one that, given a final squared integer output m, finds its square roots. Note that there are two inputs ($\pm\sqrt{m}$) for the inversion to compute. This immediately points out one important difference between pure function inversions and general program inversions: an output result of a program can be produced by many possible inputs. Hence a computational inverse is not denoted as a function if the forward computation is many–to–one, but rather, it is modelled by a relation over multiple sets of input and output values.

McCarthy approaches generalised computational inversions from the standpoint of computing and testing Turing machine computations over the enumerated values of the program's data, in essence exhaustively searching for an inverse [McC56]. Although this technique will find an inverse if it exists, the Halting Problem implies that it is impossible for the program to determine if one does not exist. In addition, exhaustive enumeration is hopelessly inefficient as a practical tool. McCarthy suggests the use of problem–specific heuristics for improving the search for an inverse. As will be shown in the sequel, this will be the key for computing tractable inverses. In fact, the syntax of high–level programming languages contains enough information for deriving efficient inverses of many algorithms. Despite the computational limits of the generalised program inversion problem, the intractability of many algorithmic inversions should not prevent the study of the multitude of algorithms that have useful and tractable inversions.

Related theoretical work is that in reversible computations and cellular automata [Ben73, Ben82, TN90]. A reversible Turing machine is one in which the direction of computation can be reversed at any time. To do this, the Turing machine is constructed with an extra tape for retaining the step–by–step history of the computation. This tape records information that would normally have been destroyed in a forward execution. Hence each forward computation step has an inverse defined for it. Bennett makes a fascinating analogy between reversible computations and the reproductive activity of organic DNA in [Ben82], in which which forward and reverse DNA reactions are analogous to forward and backward computation steps.

Other work using program inversion includes the following. Broy uses program inverses to simplify the derivation of particular types of recursive structures during program synthesis [BKB80]. Program inversion is useful for enhancing the declarative power of functional programming languages, for example, to allow $(n:1)$ functions to be treated as inverse $(1:n)$ relations. To this end, program inversions have been introduced to Lisp [Kor81], fold and unfold transformation systems [Dar81], and Hope+ [HH86]. Programming environments often make use of "undo" operations, which represent the inverses of atomic operations [Lee86]. The inversions of imperative and logic programs are discussed in the following sections.

2.2 Imperative program inversion

Dijkstra [Dij82] and Gries [Gri81] introduced the step-wise derivation of inverted imperative programs from forward-executing algorithms, and others have investigated this idea further [CU90, vW91]. Consider a program that has a computationally useful inverse, and in which the problem is more intuitively specified and implemented in its forward-executing form. For example, symbolic differentiation is more conceptually intuitive for many people than symbolic integration. In such circumstances, the simpler program (differentiation) can be used for deriving its inverted equivalent (integration). When a deterministic conventional language is used to construct such an algorithm, it is usually implicitly assumed that the inverse algorithm likewise uses the same language as the source. This is a consequence of the fact that the formal programming calculi used in the above work is concerned with implementing programs in a specific imperative language with an associated refinement calculus. The inverse program is homomorphically refined from the forward program and its specification logic. A consequence of this is that the inverse computation must be a deterministic 1:1 function of output to input, in order to be implemented in the deterministic language used by these systems. This restricts the number of algorithms that may be inverted in this setting. A possible solution is to consider deriving a meta-program in the imperative language, which takes the source program and computes an inverse. However, the formal derivation of first-order programs is sufficiently challenging to suggest that higher-order program derivation is probably too complex for the immediate future.

A program S is composed of a sequence of mechanisms Γ which operate upon a program state σ:

$$S(\sigma):\ \{\ \Gamma_1;\ \ \Gamma_2;\ \ ...\ ;\ \ \Gamma_k\ \}$$

Imperative programs and their constituent Γ_i are functions over σ, and computations are characterised by the state of the store σ. Computations begin at an initial state σ_i, and are driven towards a solution state σ_f by many small cumulative changes to intermediate states by the program constructs:

$$\sigma_i\ \stackrel{\Gamma_1}{\rightarrow}\ \sigma_1\ \stackrel{\Gamma_2}{\rightarrow}\ \sigma_2\ \stackrel{\Gamma_3}{\rightarrow}\ ...\ \stackrel{\Gamma_k}{\rightarrow}\ \sigma_f$$

If the language used is side-effect free, then each Γ_i will change the store if it is an assignment. Higher-level language constructs such as tests and loop control the specific assignments that are applied during the computation.

The inverse of a forward algorithm can be step-wise derived from the program's code and specification. Von Wright suggests that program S^{-1} is the inverse of program S under precondition P if the following total correctness formula holds for all predicates Q [vW91]:

$$(P \wedge Q)\ [S;\ S^{-1}]\ Q$$

We can think of S^{-1} as reversing or undoing the computation done by S before it. As long as all the constituent Γ are invertible, so is S:

$$\begin{aligned} S^{-1}(\sigma) &\Leftrightarrow \{ \Gamma_1;\ \Gamma_2;\ \ldots;\ \Gamma_k \}^{-1} \\ &\Leftrightarrow \{ \Gamma_k^{-1};\ \Gamma_{k-1}^{-1};\ \ldots;\ \Gamma_1^{-1} \} \end{aligned}$$

Von Wright suggests a set of *converse commands* that can be symbolically applied to the constructs of a forward program to derive a specification requirement for the inverse. For example, as indicated above, the converse of sequential composition is

$$(\Gamma_1;\ \Gamma_2)^{-1} \ = \ \Gamma_2^{-1};\ \Gamma_1^{-1}$$

Likewise, the converse of a test is:

$$(\textbf{if}\ b_1 \to \Gamma_1 \ \Box\ b_2 \to \Gamma_2\ \textbf{fi})^{-1} \ = \ \Gamma_1^{-1};\ \{b_1\}\ \vee\ \Gamma_2^{-1}; \{b_2\}$$

This denotes the idea that, if the forward execution of a test along some branch is presumed to have succeeded, then its inverse execution must be such that the test condition must be true after the inverted branch executed has terminated.

Direct translation of any Γ_i into Γ_i^{-1} in the same imperative language is often impossible. Many Γ_i are not $(1:1)$ functions of input values to output values, and consequently their inverses are not directly translatable into imperative code. This is because imperative programming languages do not contain constructs which permit nondeterministic changes of state. An obvious example is the inversion of a single destructive assignment statement, which can yield an indefinite number of prior states for the variable's previous value. Von Wright models this with:

$$(u := e)^{-1} \ = \ \bigvee_d (\{u = e[d/u]\};\ < u := d >)$$

To invert an assignment to variable u, we need to insert into u the value d it had prior to being assigned the value of expression e. This is problematic because that prior value has been destroyed. The above logical expression uses *angelic nondeterminism* [vW91] to determine the prior value d that u contained before the assignment. Since d must exist, the expression essentially uses an oracle to select it. The inverted program will be *implementable* in the imperative code when the angelic nondeterminism operator $\vee$ is replaced with deterministic constructs from the program language itself – in other words, code that deterministically determines the value of d. However, it must be stressed that successfully refining this kind of nondeterministic selection into deterministic selection can be an extremely difficult theoretical problem for many algorithms, and probably impossible for NP-complete ones.

The need for the angelic nondeterminism operator can be seen when one considers the *AND/OR tree* model of computation [Har80]. The left-most diagram in figure 1 shows some computation paths for different inputs of a terminating imperative program. Each arc represents the modification of the computation state. Execution begins by starting at one of a number of initial input states $(i_1, \ldots, i_6)$ on the left of the tree, and deterministically changing the state until a final result o_1 or o_2 is obtained on the right. Hence each path taken from an i to an o represents a separate execution run of the program. The merging of states is a natural phenomenon of imperative languages, and represents the loss of information that occurs with destructive assignments. Worth noting is that the paths in this diagram are all deterministic – there is one and only one possible computation path from any input to an output. Therefore,

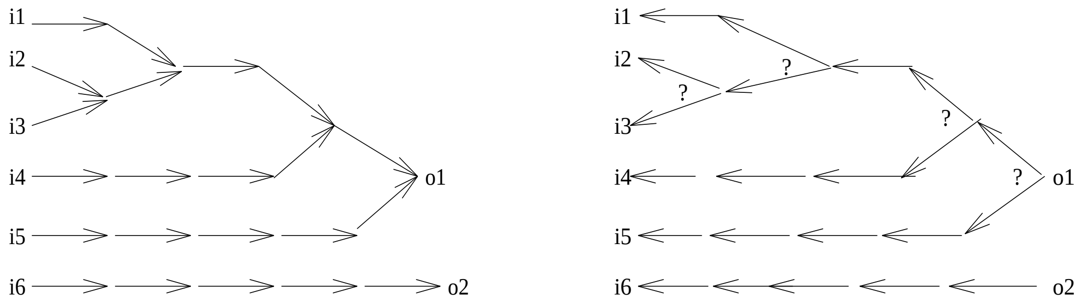

Figure 1: Forward and inverse computation trees

the AND/OR tree derivation for each computation takes the form of a single branching tree with only AND nodes.

The right-most diagram in figure 1 shows the inversions of the forward computations on the left. To perform an inversion, we start with a final value on the right, and follow branches until an initial input value is obtained. However, unlike the deterministic forward case, there is a nondeterministic choice of paths to take when inverting many computation steps. The AND/OR tree model ascribes OR nodes (labelled with "?") to nondeterministic choice points (those with multiple outgoing arcs), and AND nodes to deterministic points (those having single outgoing arcs). Inverse computation thus takes the form of a nondeterministic search on the AND/OR tree.

Imperative languages do not naturally accomodate the ability to choose a branches when executing inverse computations. Rather, the nature of imperative language constructs is such that each subsequent state must be deterministically created from the previous one. In other words, inverted program construction in an imperative language requires the refinement of angelic nondeterminism into deterministic code. Programs constructed this way have no OR nodes in their computation trees – all the branches at every OR node will have been pruned so that one deterministic choice is left. This naturally restricts all inversions to be deterministic. As we will see later in this paper, however, if a nondeterministic target language is used to implement the inverse computation, this inherent nondeterminism can be retained.

2.3 Logic program inversion

Program inversion is an intrinsic property of nondeterministic logic programs, and occurs as a natural result of applying sound and complete inference to a declarative logic program. A fundamental goal of logic programming research is the development of more powerful inference strategies, which in turn will contribute to the efficient inverted computation of logic programs.

A logic program *Prog* is a set of Horn clauses having the form $\forall\ (H \Leftarrow G_1 \wedge \cdots \wedge G_n)$ $(n \geq 0)$, where H, G_i are atoms of arity ≥ 0. The logic program syntax normally adopted for such clauses is "$H :- G_1, ..., G_n.$", where H is the clause head, and the goals G_i comprise the body [CM87]. Clauses with non-empty bodies are rules, while those with empty bodies are facts. A procedure or predicate is a collection of clauses labelled by the same head name (we assume the arity is consistent). A goal clause "$:- G_1, ..., G_m.$" represents a query Q for the logic program, and is solved by applying SLDNF resolution to refute $Prog \cup \{Q\}$ [Llo87].

The conceptual distinction between a program's declarative semantics and procedural semantics is one of the characterising features of the logic programming paradigm [Kow79]. A logic program has a *declarative semantics*, which is the program's relational interpretation when the program predicates are interpreted as comprising a first-order Horn clause theory. From the declarative perspective, a (pure) logic program describes *what* it computes – the program describes logical relations between first-order objects in some intended domain. Logic programs also have a *procedural semantics*, which is concerned with logic program behaviour with respect to a particular inference strategy. The procedural or operational perspective is concerned with *how* particular results are inferred from a particular logic program. Consequently, when incomplete inference strategies are to be used, procedural semantics will determine what subset of the declarative model is effectively computable. A single logic program can have an indeterminable number of computation trees, depending upon the the *computation rule* (or *goal selection rule*) used. For example, a logic program executed with standard Prolog's left-to-right computation rule may result in a finite computation tree, whereas the same program executed with a right-to-left goal selection can result in a tree with infinite branches. The manner in which these trees are searched depends on the *search rule*. Prolog's depth-first search rule, although efficient to implement, will blindly search infinite branches. A fair depth-first search rule will return all inferable computations in finite time; however, the resources required (exponential space) preclude the practicality of such strategies.

Logic program computations differ substantially from imperative computations from two perspectives. Firstly, as shown in figure 1, a (forward) imperative computation is defined by a single-branching AND/OR tree, where each state is deterministically derived from the previous. Given an imperative program and some input data, there is one and only one computation derivable. A single logic program can have a multitude of computations – the computation tree depends on the computation rule, and the computed results returned depends upon the search rule. This means that different computation trees can be inferred from a single logic program. Secondly, logic programming languages incorporate backtracking to search for multiple solutions in its computation tree. Backtracking represents the nondeterministic execution of alternate computation paths or branches in the tree. Hence, where imperative programs have one deterministic computation path, a logic program has multiple nondeterministic computation paths.

Consider the following declarative description of a sort program [CM87]:

$$sort(L1, L2)\ :-\ permutation(L1, L2), sorted(L2), !.$$

$$permutation([\ ], [\ ]).$$
$$permutation(L, [H|T])\ :-\ append(V, [J|U], L), append(V, U, W), permutation(W, T).$$

$$sorted([\ ]).$$
$$sorted([X]).$$
$$sorted([X, Y|L])\ :-\ X < Y,\ sorted([Y|L]).$$

This program is a high-level declarative specification of what it entails for a list of integers to be sorted. Although executable on a standard Prolog interpreter, this program is far too inefficient to be practical. However, there exist inference strategies that can execute this naive sort program with the same efficiency as a Quicksort or other efficient sorting algorithm. In other words, other inference strategies will result in much more efficient computation trees for this program. Of course, logic program interpreters are not yet sophisticated enough to automatically infer such efficient execution strategies for programs.

Pure logic programs quite readily support program inversion. Program predicates are considered to be declarative sentences in logic. The use of an appropriate inference procedure permits the determination of any relation represented within the program's ddeclarative semantics. Consider the append program:

$$append([\ ], A, A).$$
$$append([A|B], C, [A|D])\ :-\ append(B, C, D).$$

Interpreted declaratively, the relation defined by *append* implicitly represents a variety of computational directions. The goal $append([a], [b, c], Z)$ infers the third argument $Z = [a, b, c]$). On the other hand, the goal $append(X, [b, c], [a, b, c])$ infers the first list $A = [a]$. Hence these two goals are inverses of one another. Of course, a suitably written logic program permits a variety of different query forms as these, in fact, any query that can be instantiated by the program relation. Thus there is no inherent direction encoded within pure logic programs, and program inversions are computed as a natural by-product of using a sound inference strategy.

Actual implementations of logic programming languages such as Prolog do not use complete inference schemes. Rather, for efficiency's sake, they use much simpler control strategies, such as Prolog's left–to–right control rule and depth–first search rule. A result of this incompleteness is that a program's declarative interpretation is often a superset of what is actually computed by the interpreter. Therefore, a program must be written with a particular inference scheme in mind, so that the desired set of solutions can be generated. In the case of Prolog, programs are written to run effectively under left-to-right goal selection, depth-first search, and possibly extralogical control primitives such as the cut that are used to control search. The net effect of Prolog's control is that predicates are *directed* – they expect particular instantiation patterns of arguments. This is evident when one considers predicate *mode declarations*. A predicate $p(X, Y)$ with mode declaration $(+, -)$ means that the first argument must be instantiated for proper execution to occur. An inverted execution of p would use an instantiate mode of $(-, +)$. This is akin to the direction encapsulated in imperative program execution, although it is not as extreme, since a Prolog program normally has both deterministic and nondeterministic components. Truly nondeterministic predicates of course have no mode restrictions, but are too inefficient to be practical in many applications.

Shoham and McDermott experiment with inverting logic programs by deriving meta–interpreters incorporating a number of different control schemes [SM84]. Some logic programs require slightly modified inference strategies in order to infer their inverses. For example, the inverses of some logic programs is obtained by reversing the order of goal selection. A right–to–left computation rule interpreter is easily constructed, which is equivalent to transforming the ordering of goals in a clauses:

$$P(\tilde{t}_0)\ :-\ \ G_k(\tilde{t}_k),\ ...,\ G_2(\tilde{t}_2),\ ,\ G_1(\tilde{t}_1).$$

Although this suffices for a few logic program inversions, in many cases infinite computations arise from the left–recursion that is often introduced. This is due to the fact that predicates are expecting particular mode patterns consistent with the standard left–to–right rule. It turns out that increasingly sophisticated enhancements to the inference engine are required when considering different logic program inversions. Shohan and McDermott's general solution is to let the user interactively guide the inverted inference.

The inversion problem as it pertains to logic programming is one instance of a more fundamental goal of logic programming research – the invention of more efficient and powerful inference strategies. Because first-order Horn clauses are Turing machine powerful, there is not an effective decision procedure for generalised logic programs [Tar77]. Nevertheless, in many instances programs do allow the automatic deduction of useful control information. Recent enhanced implementations of Prolog have control mechanisms which conform to the dynamic state of the inference. For example, SICStus Prolog allows the delaying of goals whose arguments are not sufficiently instantiated [CW88]. This permits the execution of programs which would normally have required an altered computation rule, such as those studied by Shoham and McDermott. Many Prolog utilities, such as those for arithmetic, are strictly deterministic. SICStus Prolog contains built–in mathematical predicates which enable nondeterministic evaluation of mathematical expressions, and which automatically suspend evaluations when expressions are not instantiated enough to be evaluated. Work in abstract interpretation is also useful in this regards. The dataflow patterns inherent within a program can be ascertained, and the control scheme appropriately adapted. Sickel studies the modes of predicate arguments in order to determine the computability of particular inversions [Sic79].

3 Logically computing imperative inverses

This section gives an informal illustration of how logic programming can be used to invert imperative algorithms. Given an imperative program to be inverted, a logic program representing its relational semantics is automatically derivable for it. Inverted execution of this logic program generates inverted computations for the corresponding imperative source program. The technique described here is *not* computationally complete: the lack of a decision procedure for general logic programs, as well as the difficulty and intractability of deriving many algorithmic inversions, precludes an effective general inversion strategy. By using a few simple logic programming tools, however, the inversion of many nontrivial imperative programs is readily possible.

3.1 Imperative inversions are logical inversions

First-order predicate logic is a popular tool for modelling the semantics of imperative programming languages and the formal derivation of programs written in them. A host of different programming logics have been suggested, two famous examples being Hoare's axiomatic semantics [Hoa69] and Dijkstra's predicate transformers [Dij75]. A newer logic is Hehner's relational semantics [Heh84a, Heh84b]. Hehner's logic is appealing because it reduces the distinction between the programming and specification languages – both are directly translatable into one another. Hehner's logic treats all program language constructs as relations over the observable environment, which is usually taken to be the program variable values. A program P is therefore defined by a relation P over the initial store and final store:

$$\models P(\sigma_i,\ \sigma_f)$$

If this relational expression is logically satisfiable in the logical calculus, then it can be considered to be computationally implementable. On the other hand, if it is false, then it is not computationally feasible. The relation P is further defined according to how program P is implemented. Each language construct has a semantic expression indicating its effects on the store. Virtually

$$
\begin{array}{lcl}
\begin{array}{l}
mult([y, m, n], [\,]) : \{ \\
\quad y \ := \ 0; \\
\quad while \ (\ n \ > \ 0 \ ) \ \{ \\
\qquad n \ := \ n - 1; \\
\qquad y \ := \ y + m \\
\quad \} \\
\}
\end{array}
& \iff &
\begin{array}{l}
mult([Y_i, M_i, N_i], [Y_f, M_f, N_f]) \ :- \\
\quad Y_2 \ = * = \ 0, \\
\quad while([Y_2, M_i, N_i], [Y_f, M_f, N_f]). \\
\\
while([Y, M, N], [Y, M, N]) \ :- \\
\quad \neg N > 0. \\
\\
while([Y, M, N], Final) \ :- \\
\quad N > 0, \\
\quad N_2 \ = * = \ N - 1, \\
\quad Y_2 \ = * = \ Y + M, \\
\quad while([Y_2, M, N_2], Final).
\end{array}
\end{array}
$$

Figure 2: Integer multiplication and its (forward) semantics

all important computational concepts, including correctness, completeness, termination, program refinement, and program verification, are modellable in this logical system.

Clark and van Emden apply the Horn clause subset of first-order logic towards modelling imperative computations [CvE81]. This semantics is a relational semantics like Hehner's, but restricted to Horn clause logic. An advantage of restricting the logical model in this way is that logic programming models of computation can be applied to the semantic descriptions of programs. Ross applies this idea by using Horn clause logic in a source-to-source program transformation system [Ros89]. Imperative programs have Horn clause descriptions transliterated for them (see figure 2), which are treated as pure Prolog programs. Each predicate tuple has two arguments – the initial and final states as represented by variable store environment. An assignment

$$n := n - 1;$$

is modelled by an equality relation $= * =$ that updates the logic variable representing the variable n:

$$N2 \ = * = \ N1 - 1$$

Successive assignments to n cause successive instantiations to logic variables denoting variable n in the environment. While-loop iteration is modelled by a tail-recursive predicate that executes until the test condition is false. This type of semantics is essentially the same as that in the traditional logical semantics mentioned at the beginning of this section. There are a number of possible uses for this logic program model (besides its use in inversion to be shown in this paper). Firstly, the logic program can be immediately executed by a Prolog interpreter, which yields a fast means for prototyping and testing languages. In addition, using Hehner's predicative programming principles, by partially evaluating the logic program with a meta-interpreter, and then transforming the residual program back into the source language, source-to-source partial evaluation transformations are possible.

Consider the problem of deriving the inversion of the multiplication program in figure 2. The forward algorithm reads m and integer n, and computes $y = m * n$. One possible inverted computation is to read y and m, and determine n (ie. $n = y/m$). Disregarding the inference strategy used by the logic program interpreter, one forward execution of *mult* is denoted by the query

$$?- \ mult([,2,5], Final).$$

which computes $Final = [10, 2, 0]$. In other words, the relation $mult([\_, 2, 5], [10, 2, 0])$ exists in the declarative interpretation of the program. The inverted query

$$?- \ mult([Y, M, N], [10, 2, 0]).$$

is computable as well, since it is simply a more general instantiation of a relation known to exist. The forward and inverted computations are implicit in the declarative semantics for relation *mult*. If a powerful enough inference strategy were implemented by the logic program interpreter, both queries would be computable: the "inverse relation" is resident somewhere in the computation tree created by the particular inference strategy employed. In fact, the notion of "inverse" is ill-posed when discussing pure logic programs, given that there is no inherent direction of computation: the computability of any program relation is only limited by the power of the inference engine. The precise computation tree obtained for the logic program depends upon the control strategy the interpreter uses to execute it.

Because logic languages such as Prolog use restricted inference strategies, the *mult* predicate in figure 2 implicitly expects a forward-directed instantiation pattern ([-,+,+],[-,-,-]). When executed under Prolog (with a suitable user-definition for $= * =$) forward computations are readily inferred. The inverted query fails, however, because the expected mode instantiations for the built-in relational and arithmetic operators are violated. For example, the test "$\neg N > 0$" cannot execute, since N is uninstantiated. A more sophisticated inference strategy will be needed to execute the inverse of this particular algorithm.

3.2 A logical semantics for inversion

Given an imperative algorithm, the computability of its inverse is affected by the following:

1. Expression invertability: All mathematical (and other) primitives must have supplied for them effective means of computing their inverses. The invertibility of an expression requires an adequate definition of its component primitive operators, as well as the availability of data arguments.

2. Control: Inverting the iterative structure of iterative loops requires that the least fixpoints of the loops are computed. In other words, non-terminating computations cannot be inverted.

3. Algorithm properties: This is related to the previous points. An algorithm's inverse is ultimately algorithm dependent. Many algorithms have tractable inversions. A decision procedure for general inversions does not exist, however.

The key to performing imperative inversions is that, even though individual program statements might not be inverted directly, *it is possible to deduce the inverse of particular program mechanisms by their context with respect to the program state.* What makes many imperative programs prone to inversion is that variable values are recoverable from the state, because all variables contribute to either program control and/or the valuation of some store value. A variable's earlier value "destroyed" by an assignment is latent somewhere in the current state, it having contributed in some way to program control or the value of some variable at that point or in the future.

We will soon introduce a logic programming technique for inverting imperative computations. Before proceeding, however, it is important that we qualify exactly what we are proposing to accomplish. Our thesis is that the inversions of many imperative algorithms, including ones with nondeterministic inverses, are computable by giving its relational Horn clause semantics to an appropriate inference procedure. Since the logical semantics declaratively denotes the relation between the input and output states, forward and inverted computations are implicit within it. The ability to infer a relation for a particular query rests on the inference procedure chosen to infer solutions from the logic program's computation tree. We propose that the inference procedure required for computing the inversions of *many* imperative programs can use straight-forward control mechanisms found in currently available logic programming languages. We are not suggesting, however, that our approach will effectively compute the inverse of any general inference problem.

(i) Programs:

$$Progname(\bar{a}, \bar{e}) : \ local\ \bar{x}\ \{\ Q\ \} \quad \stackrel{\mathrm{def}}{=} \quad \models \quad Progname(<\bar{a}_i, \bar{e}_i>, <\bar{a}_f, \bar{e}_i>) \Leftarrow Q(<\bar{a}_i, \bar{e}_i, \bar{x}_i>, <\bar{a}_f, \bar{e}_f, \bar{x}_f>)$$

(ii) Chains:

$$Q \equiv S_1; S_2; ...; S_k \quad \stackrel{\mathrm{def}}{=} \quad \models \quad Q(\sigma_i, \sigma_f) \equiv S_1(\sigma_i, \sigma_2) \wedge S_2(\sigma_2, \sigma_3) \wedge ... \wedge S_k(\sigma_k, \sigma_f)$$

(iii) Assignment:

$$x := e \quad \stackrel{\mathrm{def}}{=} \quad \models \quad x_f \ = * = \ e(\sigma_i)$$

(iv) Tests:

$$\begin{array}{lcll} if\ (b)\ then\ \{\ Q_1\ \} & \stackrel{\mathrm{def}}{=} & \models & if_j(\sigma_i, \sigma_f) \Leftarrow b \wedge Q_1(\sigma_i, \sigma_f) \\ \qquad else\ \{\ Q_2\ \} & & \wedge & if_j(\sigma_i, \sigma_f) \Leftarrow \neg b \wedge Q_2(\sigma_i, \sigma_f) \end{array}$$

(v) While loops:

$$\begin{array}{lcll} while\ (b)\ \{\ Q\ \} & \stackrel{\mathrm{def}}{=} & \models & (\ while_j(\sigma_i, \sigma_f) \Leftarrow \neg b(\sigma_f) \wedge s_j(\sigma_f, \sigma_i)\ ) \\ & & \wedge & s_j(\sigma, \sigma) \\ & & \wedge & (\ s_j(\sigma_f, \sigma_i) \Leftarrow Q(\sigma_f, \sigma_1) \wedge b(\sigma_1) \wedge s_j(\sigma_1, \sigma_i)\ ) \end{array}$$

(vi) Do loops:

$$\begin{array}{lcll} do\ \ x = 1\ \ to\ \ n\ \{\ Q\ \} & \stackrel{\mathrm{def}}{=} & \models & (\ do_j(\sigma_i, \sigma_f) \Leftarrow \ (x_f \ = * = \ n+1) \ \wedge s_j(\sigma_f, \sigma_i)\ ) \\ & & \wedge & s_j(\sigma, \sigma) \\ & & \wedge & (\ s_j(\sigma_f, \sigma_i) \Leftarrow \ (x_f \ = * = \ x_1 + 1) \ \wedge \ Q(\sigma_f, \sigma_1) \\ & & & \qquad \wedge\ (n \geq x_1 \geq 0) \ \wedge s_j(\sigma_1, \sigma_i)\ ) \end{array}$$

Figure 3: Inverted relational semantics (no arrays)

Figure 3 contains a relational semantics for inverting imperative computations. With this semantics, *inverted logical semantics* are automatically transliterated for source programs. Program invocation is defined in (i). The $\overline{a}$ and $\overline{e}$ arguments are call by reference and call by value respectively. Local variables $\overline{x}$ are local variables, and are hidden by not including them in the predicate header. Chains (sequences of program constructs) are modelled in (ii) by conjunctions of input–output relations. States are passed between relations using the input–output arguments of the sequenced relations. Tests are modelled in (iv) using clauses for each of the possible results of the test outcome. Each test in the program has a uniquely named predicate (as is done for loop relations in (v) and (vi)).

In (iii), assignments are modelled by an equality operator $= * =$, which solves simple arithmetic equalities as in [SM84, CW88]. In "$x_f = * = e(\sigma_i)$", the expression e is converted to a logical form in which variables refer to values in the pre-assignment state σ_i, and the logic variable x_f represents the value of the assigned variable referred to in the post-assignment state. This equality operator is defined with as much information about the arithmetic and other primitives as possible, in order to invert given expressions. When an arithmetic expression cannot be solved, =*= will delay itself until the arithmetic expressions are sufficiently instantiated. The =*= operator is straight-forwardly implemented with the delayable arithmetic primitives found in such languages as SICStus Prolog (see appendix A). For example, consider the following assignment and its relation:

$$y := x * 2; \quad \Leftrightarrow \quad Y \ = * = \ X * 2$$

If $X = 5$, then $Y = * = 5 * 2$ infers $Y = 10$. If $Y = 10$, then $10 = * = X * 2$ infers $X = 5$. And if both $Y = 10$ and $X = 2$, then $10 = * = 5 * 2$ infers *true*. In addition, for the expression $y := x * x$, when inverting $25 = * = X * X$, both the values $X = 5$ and $X = -5$ are inferred. Such an inversion is therefore a bottom-level source of nondeterminism.

The most interesting semantic expressions are those that model inverted *while* and *do* loops in (v) and (vi). To understand them, lets first consider the "forward" relational semantics for a while loop:

$$\begin{array}{lcll} \mathit{while}\ (b)\ \{\ Q\ \} & \stackrel{\text{def}}{=} & \models & \mathit{while}(\sigma_i, \sigma_f) \Leftarrow b(\sigma_i) \wedge Q(\sigma_i, \sigma_j) \wedge \mathit{while}(\sigma_j, \sigma_f) \\ & & & \vee\ \mathit{while}(\sigma_i, \sigma_i) \Leftarrow \neg b(\sigma_i) \end{array}$$

The first disjunct is a recursive representation of the loop's forward execution, and the second is the termination case. The b test in each clause is a guard that permits only one clause to be executed for a particular state. The predicate will recurse until the termination test succeeds in the exit case.

Consider inverting the execution of this predicate. We first assume that we are inverting a terminated forward computation. Consequently, we can presume that the loop's exit test must have succeeded ($\models \neg b$) for the loop to have quit. If the final state σ_f supplied to the inverted computation triggers the exit test, then this represents the fact that the loop did indeed terminate in state σ_f. We might therefore initially presume that the loop body itself was never executed. Perhaps the loop test failed right away, and the loop was skipped. If this is the case, the final state must be equivalent to the initial state, and we have inferred one inverted computation path for the loop. The legality of this inversion will depend upon whether inverted mechanisms before the *while* loop can succeed with this final state.

This is not the only possible result for the loop. If we presume that the loop body was indeed executed one or more times, then what we must do is compute each inverse iteration of the loop. Each iteration of the loop body can be performed so long as the positive loop test evaluates to true ($\models b$) and the inverted body execution is satisfiable. Each "unwinding" of the loop is computed in this fashion, and every successful inverted loop body computation and loop test represents another nondeterministic choice point for the inverted computation. Note that performing an iteration doesn't negate the exit test we first tested. We retain the exit test at all times, since we know it must have triggered the loop's termination. Should the positive loop test or loop body fail during loop unwinding, then the possible inverse iterations for the loop have been exhausted.

This reasoning strategy is an example of *abduction* [HHNT86]. Abduction is a type of logical inference which first conjectures a particular hypothesis, and then attempts to establish its premises. This is precisely what happens when inverting a while loop (or inverting the entire program for that matter). We conjecture that the loop has terminated. We then establish that it has iterated zero or more times. Each inverted iteration of the loop body that is preceded by a positive loop test establishes the premise that the loop iterated. This line of reasoning essentially works backwards in time with respect to the computation steps of the forward execution. It is admittedly quite a challenging mental exercise for those used to thinking about imperative computations in the "normal" forward direction of time/computation, especially given that inverted computation often branches into multiple directions representing prior states that result in one given terminal state. The inverses of imperative computations are nondeterministic, and as such are much different than the deterministic forward computation.

An abductive inference strategy could be implemented in a specialised logic program interpreter. However, a better approach is to encode abduction in the inverted logical semantics itself. Brough and Hogger describe a logic program transformation which parallels the Greibach-Foster transformation used in formal grammar theory [BH86]. Given a logic program which matches the schema on the left, they propose the following logically sound transformation:

$$\begin{array}{lll} P & :- & T. \\ P & :- & P,\ R. \end{array} \quad \Longrightarrow \quad \begin{array}{lll} P' & :- & T,\ S. \\ S. & & \\ S & :- & R,\ S. \end{array} \qquad [1]$$

Presuming standard Prolog's left-first-depth-first control, the idea is that, for P on the left to successfully terminate, the goal T *must* succeed at some time. In addition, each recursive call to P means that the corresponding goal R must succeed. The transformed P' abductively assumes both of these facts *a priori*, and imposes their contribution to the computation *immediately*. Encoding these inference requirements in the program logic simplifies the inference strategy, and a standard Prolog interpreter can be used. Besides encoding abductive reasoning, a practical use of this transformation is to remove left–recursive looping – a phenomena which often results when a right–to–left goal reordering is attempted (this problem was often encountered by Shoham and McDermott [SM84]).

After applying the Griebach-Foster transformation on the forward semantics above, we get the inverted loop semantics in (v) of figure 3. With respect to the transformation scheme in [1], T is mapped to the test $\neg b$, and R to the test b and the body Q of the loop. This transformed predicate produces abductive behaviour for the logical inference, done entirely through logically transforming the program predicates. A loop's exiting condition ($\neg b(\sigma_f)$ for *while* loops) is given initial priority, which introduces a termination condition for the recursion. If goal delaying is used, the termination condition will delay until its expression is instantiated enough for evaluation. Each successive call to the loop body (second clause of s_j) is an inverted iteration of the loop. Loop iteration quits when the body clause cannot be satisfied. The *Do* loop semantics in (vi) is similar to the while loop, except that the loop counting mechanism and exit conditions are encoded in the predicate with their usual meaning.

The above semantics gives a general means for transliterating a source program into a logical expression conducive to inverted computation. Unfortunately, there is still a pernacious problem with data dependencies that arise when supplying an interpreter with an inverted program query. In order to invert a mathematical, relational, or some other such expression with the $=*=$ operator, expressions must have enough data available in order to invert uninstantiated argument(s). Data instantiation patterns can vary greatly as the inverted computation proceeds. In a few programs with deterministic inverses, predicate goals can be reordered with respect to the data flow to permit particular goals to execute. This will not necessarily work with all inversions, however, since data instantiation will often change dynamically as a nondeterministic inversion computation space is explored. A better solution is to use a coroutining or goal-blocking mechanism, such as that in SICStus Prolog [CW88]. The code

$$:-\ block\ \ do\_sum(?,-,-), do\_sum(-,?,-), do\_sum(-,-,?).$$

$$\begin{aligned}
do_sum(X,A,B)\ &:-\ var(X), nonvar(A), nonvar(B), !, X\ is\ A+B.\\
do_sum(X,A,B)\ &:-\ nonvar(X), nonvar(A), nonvar(B), !, X =:= A+B.\\
do_sum(X,A,B)\ &:-\ nonvar(X), var(A), nonvar(B), !, A\ is\ X-B.\\
do_sum(X,A,B)\ &:-\ nonvar(X), nonvar(A), var(B), !, B\ is\ X-A.
\end{aligned}$$

defines a predicate *do_sum* that can compute any one of the arguments so long as the other two are instantiated. Otherwise, it blocks so long as two or more arguments remain uninstantiated. This type of control mechanism represents the use of a goal selection rule that changes according to the dynamic state of the environment. We therefore presume that the arithmetic and relational expressions used by $=*=$ and other relational tests will be encoded with logical primitives that are self-invertible, and will delay if required by the data flow.

3.3 Inverted inference

An inverted logic program is transliterated for an imperative source program using the semantics of the previous section. When suitably executed, this logic program computes the inversion of the source program. Multiple initial states for inverted computations are reflected by multiple successful solution paths in the computation tree. Likewise, infinite inverted computations will have infinite computation trees. The practicality of computing these inversions depends upon the effectiveness and efficiency of the inference scheme used to search these (possibly infinite) computation trees.

Many imperative algorithms are well–suited to inversion, in fact, more so than logic program equivalents. Examples of such programs are in section 4. One assumption that can always be made about imperative computations is that *variables are always initialised before their use in expressions.* Therefore, the logic programs representing inverted imperative programs are atypical of most "natural" logic programs, and permit the following simple albeit incomplete inversion strategy:

1. A potentially valid computation tree is abductively constructed. Depending on the dataflow, some branches of the tree can be fully executed, while other branches suspended according to their dataflow requirements. The suspended branches represent symbolic reconstructions of potential forward computations.

2. The (logic) variables are set to some initial state.

3. Delayed goals are unblocked and executed as delayed variable values become available.

4. If the entire path succeeds, another inverse computation path has been found. If it fails, backtracking will construct another path, and the above steps recurse.

This exploits the forward, deterministic nature of imperative computations. The state drives the search in that branches are explored as long as they satisfy the program state/variable values. Failure to satisfy the state means that new branches of the search tree are immediately explored. This control presupposes the use of a logic program interpreter with goal delaying or coroutining, such as in [CW88], which allows control to be determined by the dataflow patterns of the program. Of course, this does not mean that infinite or nonterminating branches can be avoided. Many inverted algorithms have exponential or infinitely branching computation trees. This technique is therefore effective with well-behaved inverted algorithms.

Nondeterminism is introduced into imperative inversions from two sources. The first is the nondeterministic nature of assignments and the mathematical and other primitives used in them. For example, inverting the integer assignment "$25 := N * N$" has two solutions, 5 and -5, each of which represents a choice point in the inverted computation tree. The other source of nondeterminism is inverted loop iteration: inverted loops may have zero, finite , or infinite number of inverted solutions. Each backwards–iterated execution of a loop computed by the Brough–Hogger predicate schema is another potential choice point.

A feel for the above inversion scheme will now be conveyed through some manual inversions. We will invert a chain of assignments and an *if* statement. Section 4 gives further examples that include iterations. First, consider the following chain of assignments, along with its logical translation:,

$$\begin{array}{l} n \ := \ n - 1; \\ y \ := \ y + n; \\ z \ := \ y * y; \end{array} \quad \Longleftrightarrow \quad \models \ (N2 = * = N - 1) \wedge (Y2 = * = Y + N2) \wedge (Z2 = * = Y2 * Y2)$$

If the initial store is [n=3,y=2,z=6], then the final store will be [n=2,y=4,z=16]. Now consider inverting the program from this final store. We take the logical expression and substitute these final values into the logical variables:

$$\models \ (2 = * = N - 1) \wedge (4 = * = Y + 2) \wedge (16 = * = 4 * 4)$$

Each of these goals is directly inferable from the information at hand, and therefore any computation rule can be used to infer the initial store [n=3,y=2,z=_]. Note that the initial z is undeterminable, since it does not contribute to the final state in any way.

Let us invert this same code, but from the final store [n=2,y2=_,z=16] instead, in which y is undetermined. Again, we substitute these values into the logical expression:

$$\models \ (2 = * = N - 1) \wedge (Y2 = * = Y + 2) \wedge (16 = * = Y2 * Y2)$$

At this point, the first and last goals can be inferred. The middle goal cannot, however, because there is too little information at hand to infer values for Y or $Y2$ from this expression alone. Should this goal have been selected by some inference strategy, the interpreter would have blocked this goal until either Y or $Y2$ became instantiated. The simplified expression is:

$$\models \ (Y2 = * = Y + 2) \wedge (16 = * = Y2 * Y2) \qquad [N = 3] \qquad (\dagger)$$

At this point, the only possibility is to simplify the last goal. Because we are dealing with an integer squaring operator, there are two possible results, and each represents a different nondeterministic path. The first possible simplification is for $Y2 = 4$, giving:

$$\models \ (4 =*= Y+2) \qquad [N=3, Y2=4]$$

which then becomes:

$$\models \ true \qquad [N=3, Y2=4, Y=2]$$

This denotes the possible computation that begins with the store [n=3,y=2,z=_], and ends in the store [n=2,y=4,z=16]. The second simplification from step (†) is if $Y2 = -4$, yielding:

$$\models \ (-4 =*= Y+2) \qquad [N=3, Y2=-4]$$

which gives

$$\models \ true \qquad [N=3, Y2=4, Y=-6]$$

The other inverted computation is therefore the one with initial store [n=3,y=-6,z=_] and final store [n=2,y=-4,z=16].

Next, consider inverting the following test (clause heads are omitted from the logic):

$$\begin{array}{l} if\ (\ n=3\ )\ \{ \\ \quad n\ :=\ n+1; \\ \quad z\ :=\ y+y; \\ else\ \{ \\ \quad n\ :=\ n-1; \\ \quad y\ :=\ y+n; \\ \quad z\ :=\ y*y; \\ \} \end{array} \quad \Longleftrightarrow \quad \begin{array}{l} \models\ (N =*= 3) \wedge (N2 =*= N+1) \wedge (Z2 =*= Y+Y) \\ \quad \vee\ (\neg N =*= 3) \wedge (N2 =*= N-1) \wedge (Y2 =*= Y+N2) \\ \quad \wedge (Z2 =*= Y2 * Y2) \end{array}$$

Note that the *else* block contains the three assignments we inverted above. The procedure for inverting this code is similar to the previous example. We begin with a desired final state, and the infer possible initial state(s). The only difference is that there are two clauses to check – one for each possible test result. Let the final state be the state [n=2,y=4,z=16] which we used for the previous example. We begin by inverting the positive *if* case:

$$\models \ (N =*= 3) \wedge (2 =*= N+1) \wedge (16 =*= 4+4)$$

We immediately disregard this clause, because the final test case $16 =*= 4+4$ fails. The second *if* case is:

$$\models \ (\neg N =*= 3) \wedge (2 =*= N-1) \wedge (4 =*= Y+2) \wedge (16 =*= 4*4)$$

As before, the final three goals are refined, until we get

$$if([N, Y, Z], [2, 4, 16]) \Leftarrow (\neg 3 =*= 3) \qquad [N=3, Y=2]$$

However, this test condition fails, because the *else* case is only valid for $n \neq 3$. We therefore infer that the final store [n=2,y=4,z=16] is not a possible final state for this program segment. If this construct were part of a larger program, then the likely scenario would be that the final state is an inferred inverted state for program constructs following this *if* statement. Should the inversion of the *if* fail as we have found, then backtracking would enable the interpreter to infer another nondeterministic inversion for the subsequent commands. This should continue until the *if* is indeed satisfied, upon which time its own inverted states would be used to invert any program segment that precedes it in the source program.

Note that the use of a logic program interpreter alleviates all of the bookkeeping we did in the above examples. The binding of variables, selection of goals, and maintenance of nondeterministic choice points during clause selection are all automatically maintained by the interpreter. This highlights the fact that logic program interpreters are theorem provers, and that the computation of program inversions is equivalent to inferring the satisfaction of logical theorems describing the source program.

4 Examples

4.1 Deterministic inversion: integer multiplication

$mult([y,x,n])$: {
   y := 0;
   $while$ ($n > 0$) {
     $n := n - 1$;
     $y := y + x$
   }
 }
	$\Longleftrightarrow$	$mult([Y,X,N], FinalEnv)$:−
		   $while([Y2,X,N], FinalEnv)$,
		   $Y2 = * = 0$.
		
		 $while(StartEnv, [Y,X,N])$:−
		   $\neg N > 0$,
		   $s([Y,X,N], StartEnv)$.
		
		 $s(Env, Env)$.
		 $s([Y2,X,N2], Env)$:−
		   $Y2 = * = Y + X$,
		   $N2 = * = N - 1$,
		   $N > 0$,
		   $s([Y,X,N], Env)$.

Figure 4: Integer multiplication and its inverted predicate

Figure 4 shows the multiplication program of figure 2 along with its inverted predicate. Executing the query

$$? - \; mult([\_, 2, 3], Final).$$

with the forward logic in figure 2 generates $Final = [6, 2, 0]$ (ie. $y = 6$). We try the inverted program with the query

$$? - mult(Start, [6, 2, 0]).$$

Running the query with a standard Prolog interpreter returns a first solution of $Start = [Y, 2, 3]$, which is indeed the inverted computation, having determined that $n = 3$. Of interest is the uninstantiated variable Y. This indicates that, because y is immediately assigned 0 at the start, any value can be passed as y with no effect on the computation. The classical interpretation of logic variables within logical expressions conveys useful information during inversion: a logical variable in an inverted answer means that the result holds for *any* initial value of that variable. Note that this inversion can execute with a standard Prolog interpreter. Enhanced control such as coroutining is not required, because each assignment is directly invertible from the information at hand.

Figure 5 shows the computation tree for the above inversion query when Prolog's standard left-first-depth-first inference is used. The initial iterations of the loop satisfy the while loop test; however, each of these iterations does not satisfy the initial assignment $y := 0$. Abduction applies successive inverse iterations to the loop, until this condition is satisfied and a result is obtained. The loop is essentially rewound until $Y = * = 0$ is satisfied. Each invocation of the recursive s clause represents another rewinding of the loop. The inversion of the loop is best understood by looking at the relations for *while* that are inferred during the inversion, which are obtained by applying the computed answer substitutions to the *while* atom:

$$\begin{array}{lll}
\models & while([6,2,0],[6,2,0]) & : 0 \;\; iterations \\
\models & while([4,2,1],[6,2,0]) & : 1 \;\; iteration \\
\models & while([2,2,2],[6,2,0]) & : 2 \;\; iterations \\
\models & while([0,2,3],[6,2,0]) & : 3 \;\; iterations \\
\models & while([-2,2,4],[6,2,0]) & : 4 \;\; iterations \\
\ldots & &
\end{array}$$

The inverted computation state (the first argument of *while*) represents a store configuration that results in the final given state [y=6,x=2,n=0]. The subsequent computed state corresponds to the previous state required to execute the loop one additional iteration.

Another query is possible:

$$?- \;\; mult(E, [6,2,1]).$$

asks for an inverted computation in which the final n is 1 rather than 0. The immediate answer from the interpreter is *no*. This confirms that the abductive reasoning mechanism is working. It is impossible for the loop to terminate if the final value of n is 1, since the loop exit test cannot be satisfied.

After the first inverse result above was successfully obtained, another request for a solution (perhaps to test if the program's inverse is nondeterministic) results in a search down a nonterminating branch. This is due to the loop's continuing inversion over higher initial n values, all of which must fail because y becomes negative. A discussion of this phenomenon is given in section 5. One solution to this is to consider y as being a data type that ranges over positive integers (including 0). If this is done, then $= * =$ should be defined so that expressions can never infer a negative value for y. If this enhancement is incorporated in the logic program, an interesting variety of general queries are possible. The query

$$?- \;\; mult(E, [7,2,0]).$$

can be given, which gives *no* as a result, since this final store is inconsistent with any forward computation of the program. Likewise, the query

$$?- \;\; mult([Y,3,N], [6,2,0]).$$

asks for an inverse computation in which x is initially 3, but terminates with the value 2. Again, the interpreter replies *no*. Since x is not assigned to anywhere in the program, its initial and final values must be the same.

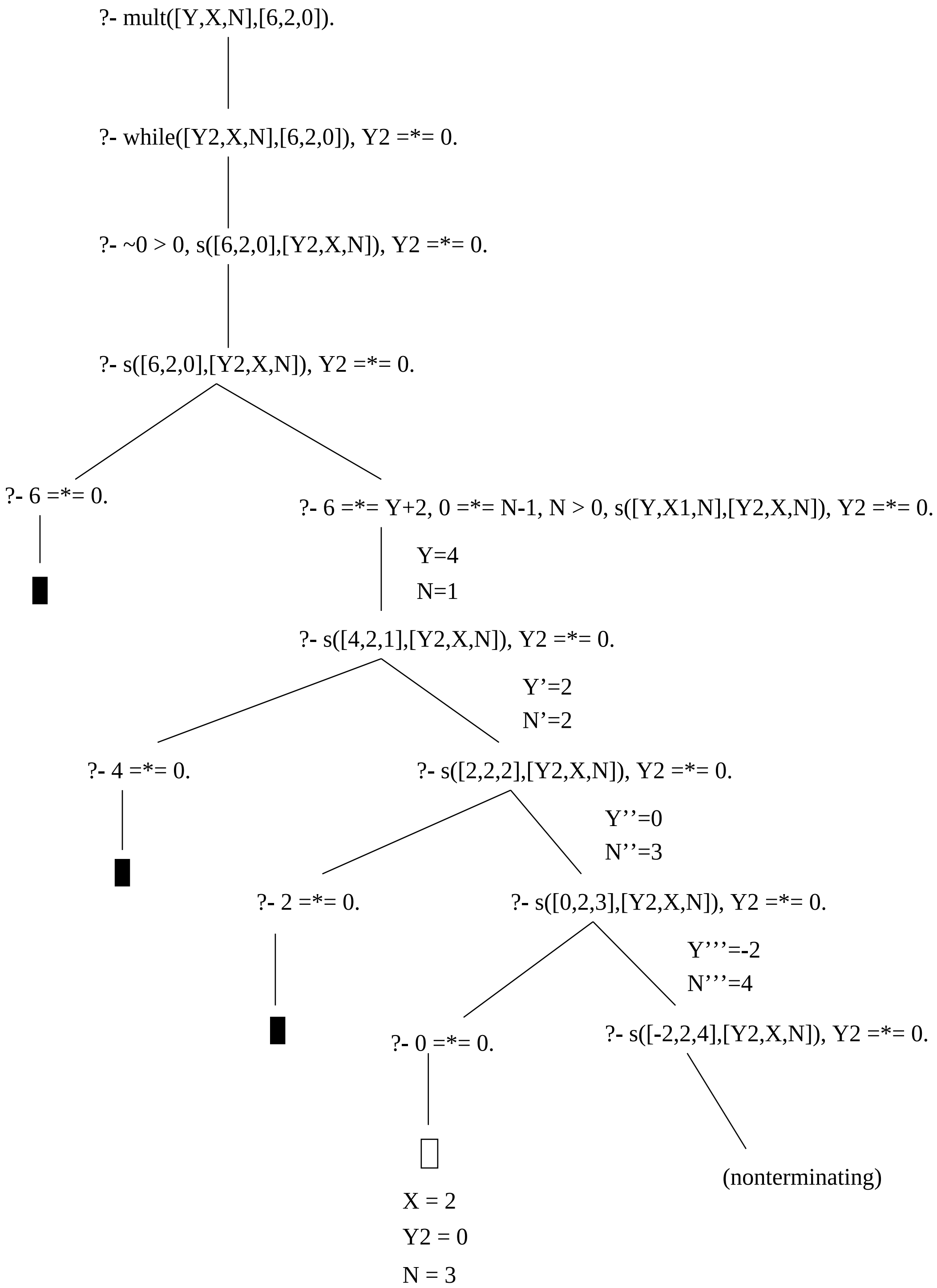

Figure 5: Computation tree for mult inversion

4.2 Nondeterministic inversion: binary powers algorithm

$$
\begin{array}{lcl}
\begin{array}{l}
power([y,x,n]): \{ \\
\quad y := 1; \\
\quad while\ (\ n > 0\ )\ \{ \\
\quad\quad while\ (\ n//2*2 =:= n\ )\ \{ \\
\quad\quad\quad n := n//2; \\
\quad\quad\quad x := x*x \\
\quad\quad \}; \\
\quad\quad n := n-1; \\
\quad\quad y := y*x \\
\quad \} \\
\}
\end{array}
& \Longleftrightarrow &
\begin{array}{l}
power([Y1,X1,N1], FinalEnv)\ :- \\
\quad while2([Y3,X1,N1], FinalEnv), \\
\quad Y3\ =*=\ 1. \\
\\
while2(StartEnv, [Y5,X5,N5])\ :- \\
\quad \neg N5 > 0, \\
\quad s2([Y5,X5,N5], StartEnv). \\
\\
s2(Env, Env). \\
s2([Y4,X3,N4], Env)\ :- \\
\quad Y4\ =*=\ Y1*X3, \\
\quad N4\ =*=\ N3-1, \\
\quad while1([X1,N1],[X3,N3]), \\
\quad N1 > 0, \\
\quad s2([Y1,X1,N1], Env). \\
\\
while1(StartEnv, [X5,N5])\ :- \\
\quad \neg N5//2*2\ =:=\ N5, \\
\quad s1([X5,N5], StartEnv). \\
\\
s1(Env, Env). \\
s1([X3,N3], Env)\ :- \\
\quad X3\ =*=\ X1*X1, \\
\quad N3\ =*=\ N1//2, \\
\quad N1//2*2\ =:=\ N1, \\
\quad s1([X1,N1], Env).
\end{array}
\end{array}
$$

Figure 6: Binary powers

Consider the program *power* and its inverted predicate in figure 6. *Power* is an imperative implementation of a binary powers algorithm which, given the integer inputs x and n, efficiently computes $y = x^n$. Forward execution with the input $x = 2$ and $n = 6$ computes a final state $[y = 64,\ x = 16,\ n = 0]$. Using a standard Prolog interpreter, the inverted query

$$?-\ power(Startstate, [64, 16, 0]).$$

produces three results:

$$
\begin{array}{rll}
(i) & Startenv = [Y, 4, 3] & : 64 = 4^3 \\
(ii) & Startenv = [Y, 2, 6] & : 64 = 2^6 \\
(iii) & Startenv = [Y, -2, 6] & : 64 = (-2)^6
\end{array}
$$

These are the three initial states which produce the given final state $[64, 16, 0]$. This is therefore an example of a nondeterministic inversion. The nondeterminism results from squaring integers in the inner loop assignment ($=*=$ computes both positive and negative integral square roots), as well as repeated abductive unwinding of the loops.

4.3 Coroutining: Fibonacci numbers

$$
\begin{array}{lcl}
\begin{array}{l}
fib([x,n]):\ local\ y,z;\ \{ \\
\quad x\ :=\ 0; \\
\quad z\ :=\ 0; \\
\quad while\ (\ n>0\ )\ \{ \\
\qquad y\ :=\ z; \\
\qquad z\ :=\ x; \\
\qquad x\ :=\ y+z; \\
\qquad n\ :=\ n-1; \\
\quad \} \\
\}
\end{array}
& \Longleftrightarrow &
\begin{array}{l}
fib([X,N],[X3,N3])\ :-\\
\quad X2\ =*=\ 1,\\
\quad Z\ =*=\ 0,\\
\quad N2\ =*=\ N-2,\\
\quad while([X2,N2,Y,Z],[X3,N3,Y3,Z3]).\\
\\
while(StartEnv,[X,N,Y,Z])\ :-\\
\quad \neg\ N>0,\\
\quad s([X,N,Y,Z],StartEnv).\\
\\
s(Env,Env).\\
s([X2,N2,Y2,Z2],Env)\ :-\\
\quad Y2\ =*=\ Z,\\
\quad Z2\ =*=\ X,\\
\quad X2\ =*=\ Y2+Z2,\\
\quad N2\ =*=\ N-1,\\
\quad N>0,\\
\quad s([X,N,Y,Z],Env).
\end{array}
\end{array}
$$

Figure 7: Fibonacci numbers

This example shows an inversion which requires a more general control scheme than the previous examples. Figure 7 shows a simple Fibonacci program with its inverted semantics. The forward program takes an integer n, and calculates the n^{th} Fibonacci number. Executing the inverted query

$$?-\ fib(E,[144,0]).$$

returns the solution $E = [\_,\ 13]$, meaning that 144 is the 13^{th} Fibonacci number. This logic program does not execute under standard Prolog control. A problem arises with the goal $N2 =*= N-2$. Both $N2$ and N are uninstantiated for the inverted query above, and there is too little information available for $=*=$ to execute. If coroutining is used, however, this goal can block until either $N2$ or N get a value, at which time it can be unblocked and resolved.

4.4 A final example that uses arrays: Algorithm P

This example is used to illustrate how conventional data structures such as arrays can be used within logical inversions. The program in figure 8 is an implementation of Knuth's Algorithm P [Knu81]. Given a permutation of n different integers, the forward program calculates a unique number k ($0 \leq k < n!$) based on the relative ordering of integers in the permutation. Executing algorithm P on the permutation $[1,2,4,3]$ produces the permutation number $f = 17$ that is uniquely calculated for this particular permuted configuration of integers. Note that the final state for this terminating computation sets $r = 0$, $s = 1$, and $u = [1,2,3,4]$ (the permutation is sorted as a side effect).

We will invert the program to compute the inverse problem of generating the permutation configuration that corresponds to a given permutation number. This inverted problem shows how

$$
\begin{array}{ll}
\begin{array}{l}
algp([u,t,f]):\ local\ [r,s]\ \{\\
\quad r\ :=\ t-1;\\
\quad f\ :=\ 0;\\
\quad while\ (\ r>0\ )\ \{\\
\quad\quad (*\ s\ is\ max.\ element\\
\quad\quad\quad between\ u[0]\ and\ u[r]\ *)\\
\quad\quad call\ maxarr(u,r,s);\\
\quad\quad f\ :=\ r*f+f+s;\\
\quad\quad t1\ :=\ u[r];\\
\quad\quad u[r]\ :=\ u[s];\\
\quad\quad u[s]\ :=\ t1;\\
\quad\quad r\ :=\ r-1\\
\quad \}\\
\}
\end{array}
& \Longleftrightarrow \quad
\begin{array}{l}
algp([U1,T1,F1|\_],Env)\ :-\\
\quad while([U1,T1,F2,R2,S1],Env),\\
\quad F2\ =*=\ 0,\\
\quad R2\ =*=\ T1-1.\\
\\
while(Startenv,[U4,T4,F4,R4,S4])\ :-\\
\quad \neg\ R4>0,\\
\quad s([U4,T4,F4,R4,S4],Startenv).\\
\\
s(Env,Env).\\
s([U3,T1,F4,R2,S2],Env)\ :-\\
\quad R2\ =*=\ R1-1,\\
\quad arrasgn(U2,S2,Tmp1,U3),\\
\quad arrasgn(U1,R1,Tmp2,U2),\\
\quad arrval(U1,S2,Tmp2),\\
\quad arrval(U1,R1,Tmp1),\\
\quad F4\ =*=\ F3+S2,\\
\quad F3\ =*=\ F2+F1,\\
\quad F2\ =*=\ F1*R1,\\
\quad maxarr(U1,R1,S2),\\
\quad R1\ >\ 0,\\
\quad s([U1,T1,F1,R1,S1],Env).
\end{array}
\end{array}
$$

Figure 8: Algorithm P

the inversion of a forward algorithm can be computationally interesting from a problem solving perspective. The logic program introduces some array relations. In $arrval(A,B,C)$, C is the value of the B^{th} entry of array A. $arrasgn(A_1,B,C,A_2)$ places value C into the B^{th} slot of array A_1, resulting in array A_2. $maxarr(A,B,C)$ is satisfied if C is the maximum array element in U between $A[0]$ and $A[B]$. The query

$$?-algp([Perm|\_],[[1,2,3,4],4,17,0,1]).$$

queries for the permutation matching permutation number 17. The inverted result $Perm = [1,2,4,3]$ is returned.

The power of logical inversion is readily seen by considering more general mixed-mode queries for the inverted program. This permits many of the final store values from the inverted query to inferred as well. A query

$$?-\ algp([Perm|\_],[[1,2,3,4],4,F|\_]).$$

results in the generation of a table of all possible permutations $Perm$ and their corresponding permutation numbers F. This effectively permits the source program to be exhaustively tested for all possible permutations of 4 elements.

5 Conclusion

The feasibility of computing the inversions of imperative programs using logic programming techniques has been shown. Features of this approach are:

- Correctness: The computation of inversions can be considered a purely logical derivation based on the program's formal axiomatic semantics. The set of program axioms define a theory of the program's behaviour, and the inference rule used is SLDNF resolution. The logic program declaratively describes an inverted imperative computation.
- Nondeterminism: The logic programming computation model permits sets of inputs to be obtained for a given output. By using a nondeterministic programming language to compute inversions, many more algorithms can be inverted than if a deterministic imperative language is used as the target for implementing the inverse algorithm. This is an entirely novel way of analysing deterministic imperative computations.
- Semi–automation: The logic programming inference engine searches the (possibly infinite) search space. The examples in the paper execute under a standard Prolog interpreter with coroutining.

More work in logic programming languages is required before the technique proposed in this paper can be practical in a production environment. The completeness of the inference strategy determines whether general inversions can be effectively computed. The control scheme used here is not complete, and thus cannot compute all solutions in all instances. Our use of coroutining was one attempt to generalise control. In any case, the ability to compute an inverse depends on the sophistication of the inference engine for the language. The introduction of abstract interpretation, dynamic control mechanisms such as intelligent backtracking, and search heuristics, will greatly aid program inversions, both for imperative programs and logic programs. Even if strong inference schemes are available, program inversion is still ultimately algorithm dependent. As mentioned in section 2, while some programs have tractable inversions, other algorithmic inversions are unknown (Fermat's equation), while others are impossible (Halting Problem). The complexity of programming language mechanisms and encoded mathematical expressions also determines the computability of inversions. For example, higher-order polynomial mathematical expressions are inherently more difficult to invert than the simple arithmetic used in the examples in section 4.

Although a transliterated logic program faithfully models the declarative semantics of its source program, it does not model *deep knowledge* about the nature of the algorithm. This points out the limited algorithmic information encoded within an imperative program's logical semantics: these logical semantics are far more shallow in content than what is found within program specification logic as used in [Hoa69, Dij75, Heh84b]. For example , consider the infinite computation tree in figure 5. The logical relation for *mult* does not encode the fact that y can never be negative in this program. The deduction of this simple fact is not within the inferential power of SLDNF resolution. That having been said, it is important to realise that the inverted predicates are automatically compiled, rather than meticulously derived as is the case for program specifications. More significantly, the limited algorithmic content encoded within them is adequate for computing many inversions.

Since generalised inference and inversion are computationally difficult tasks, it is reasonable for the logic program and/or inference system to be supplemented with appropriate problem-specific information if it helps with the computation of inversions. In a program such as *mult*, if a program tester knows a fact about the algorithm, for example, the invariant $\{Y >= 0\}$, then he or she is free to enhance the inverted relation with this information. This strategy is suggested by Vasak and Potter, in which additional termination conditions are encoded within logic programs to stop execution down fruitless paths of the computation [TJ85]. Of course, knowing how to control inference in such a manner is not always feasible – knowing such heuristics for particular algorithms would resolve many outstanding unsolved mathematical and computational problems (and paradoxes). But should an algorithm have a well-defined specification and well-behaved inversion properties, then information from the specification can be included within the logical relation to aid in its inference. As theorem proving and logic programming technology advances, the need for such programmer intervention will diminish.

Advanced logic programming languages can directly contribute to the inversion problem in both imperative and logic programming domains. The use of constraint logic programming (CLP) [vH90] for inversion is a very promising approach currently under investigation. CLP permits arithmetic reasoning to be performed on contraints, which are inequalities over arithmetic and other domains. An example of the type of analysis possible with CLP is the following. Consider the imperative program,

$$
\begin{array}{l}
b\ :=\ 0;\ \ c\ :=\ 0; \\
if\ \ (\ \ a \geq 0\ \ )\ \ \{\ \ b\ :=\ 1\ \ \} \\
if\ \ (\ \ a \leq 0\ \ )\ \ \{\ \ c\ :=\ 1\ \ \}
\end{array}
$$

The tests in the *if* statements can be considered to be constraints in a CLP semantics for the program. Then, given a final state of $b = c = 1$, both the test constraints $a \geq 0$ and $a \leq 0$ will be valid. A CLP interpreter can then deduce that these constraints reduce to $a = 0$.

Bennett discusses logical reversibility of computation in [Ben73], in which he compares biochemical DNA reactions with reversible computations:

> [...] we can see that here the reactions form a branching structure, with a main trunk corresponding to the desired computation path, and side branches corresponding to incorrect or "extraneous" reverse computations. [...] A few such extraneous states would pose no problem – a small driving force would still suffice to push the system into the desired final state. Temporary backward excursions onto the side branches would occur, but would not lead to errors, contrary to what one might expect. Since no state of a deterministic computer can have more than one logical successor, the erroneously reversed operations would be corrected as soon as the computation proceeded forward again, and the desired path would be rejoined. The real problem comes from the enormous number of extraneous predecessors; typically they outnumber the states on the intended computation path by hundreds of orders of magnitude. This is because, in irreversibly programmed computations, one can usually proceed backward along an extraneous path for many steps, making further wrong choices along the way, before arriving at a state that has no predecessors.

Although the problem domain to which Bennett refers is somewhat different in purpose to imperative inversions, it is valuable to note that the computation space and inference mechanism he describes is identical to the computation trees used in the logical inversions in this paper. A logical computation space can have applied to it more intelligent inference decisions, unlike the reversible Turing machine automatons considered by Bennett. Extraneous paths can be discounted in logical inversions when finite failure is encountered. In addition, many extraneous paths are very useful, since imperative inversions can be nondeterministic.

Inversion as a program testing paradigm has not been extensively explored. The power of logical inversion is that it permits imperative programs with destructive assignments to be treated as relations. Having an inverse computable for a program allows the program's input–output relation to be tested. This type of testing lets one see whether any undesirable final store values can be computed, and what input causes their computation. Catastrophic error checking can be done by inspecting inverted computations for undesirable initial states. The modular nature of logic programs is well–suited to imperative program debugging, as modular debugging is the only feasible testing methodology for large–scale software systems.

In conclusion, it is worth exploring what other ways logic programming can be applied to the large body of work in the logical semantics of conventional programming languages. The sharing of technology between these two different paradigms will be productive.

Acknowledgement: The author is supported by NSERC Operating Grant 0138467.

References

[Ben73] C.H. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 17:525–532, 1973.

[Ben82] C.H. Bennett. The thermodynamics of computation – a review. *International Journal of Theoretical Physics*, 21(12):905–940, 1982.

[BH86] D. Brough and C.J. Hogger. The Treatment of Loops in Logic Programming. Technical Report 86/16, Imperial College, 1986.

[BKB80] M. Broy and B. Krieg-Bruckner. Derivation of invariant assertions during program development by transformation. *TOPLAS*, 2(3):321–337, July 1980.

[CM87] W.F. Clocksin and C.S. Mellish. *Programming in Prolog (3rd ed).* Springer-Verlag, 1987.

[CU90] W. Chen and J.T. Udding. Program inversion: more than fun! *Science of Computer Programming*, 15:1–13, 1990.

[CvE81] K.L. Clark and M.H. van Emden. Consequence verification of flowcharts. *IEEE Transactions on Software Engineering*, SE-7(1):52–60, January 1981.

[CW88] M. Carlsson and J. Widen. *SICStus Prolog User's Manual*, October 1988.

[Dar81] J. Darlington. An experimental program transformation and synthesis system. *Artificial Intelligence*, 16:1–46, 1981.

[Dij75] E.W. Dijkstra. Guarded commands, nondeterminacy, and the formal derivation of programs. *Communications of the ACM*, 18:453–457, August 1975.

[Dij82] E.W. Dijkstra. EWD671: Program inversion. In *Selected Writings on Computing: A Personal Perspective*, pages 351–354. Springer-Verlag, 1982.

[Gri81] D. Gries. *The Science of Programming.* Springer-Verlag, 1981.

[Har80] David Harel. And/Or programs: a new approach to structured programming. *ACM Transactions on Programming Languages and Systems*, 2(1):1–17, January 1980.

[Heh84a] E.C.R. Hehner. Predicative Programming Part I. *Communications of the ACM*, 27(2):134–143, February 1984.

[Heh84b] E.C.R. Hehner. *The Logic of Programming.* Prentice-Hall, 1984.

[HH86] P.G. Harrison and H.Khoshnevisan. On the Synthesis of Function Inverses. Technical report, Imperial College, 1986.

[HHNT86] J.H. Holland, K.J. Holyoak, R.E. Nisbett, and P.R. Thagard. *Induction.* MIT Press, Cambridge, Mass., 1986.

[Hoa69] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[Knu81] Donald E. Knuth. *Seminumerical Algorithms.* Addison-Wesley, 2nd edition, 1981.

[Kor81] R.E. Korf. Inversion of applicative programs. In *IJCAI*, pages 1007–1009, 1981.

[Kow79] R. Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436, July 1979.

[Lee86] G.B. Leeman. A formal approach to undo operations in programming languages. *Journal of the ACM*, 8(1):50–87, January 1986.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming (2nd ed).* Springer-Verlag, 1987.

[LS84] J. Loeckx and K. Sieber. *The Foundations of Program Verification.* Wiley-Teubner, 1984.

[McC56] J. McCarthy. The inversion of functions defined by turing machines. In C.E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 177–181. Princeton University Press, 1956.

[Ros89] B.J. Ross. The partial evaluation of imperative programs using prolog. In *Meta–programming in Logic Programming*, pages 341–363. MIT Press, 1989.

[RW89] B.J. Ross and P.F. Wilk. The Logical Inversion of Imperative Computations. Technical Report DAI 414, Dept. of AI, University of Edinburgh, 1989.

[Sic79] S. Sickel. Invertibility of logic programs. In *4th Workshop on Automated Deduction*, pages 103–109, Austin, Texas, February 1-3 1979.

[SM84] Y. Shoham and D.V. McDermott. Directed relations and inversion of prolog programs. In *International Conference on Fifth Generation Computer Systems*, pages 307–316, 1984.

[Tar77] S. Tarnlund. Horn Clause Computability. *BIT*, 17:215–226, 1977.

[TJ85] T.Vasak and J.Potter. Metalogical control for logic programs. *Journal of Logic Programming*, 2(3):203–220, October 1985.

[TN90] T. Toffoli and N.Margolus. Invertible cellular automata: a review. *Physica D, Nonlinear Phenomena*, 45(1-3), 1990.

[vH90] P. van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1990.

[vW91] J. von Wright. Program inversion in the refinement calculus. *Information Processing letters*, 37:95–100, January 1991.

A Appendix: Implementation of $=*=$

The following code is a coroutined implementation of $=*=$ that was used for the examples in this paper. It is implemented in SICStus Prolog [CW88].

```
?- op(700,xfx,'=*=').

A =*= B :- var(A), nonvar(B), !, do(A, B).
A =*= B :- var(B), nonvar(A), !, do(B, A).
A =*= B :- nonvar(A), nonvar(B), !, do(C,A), do(C,B).
A =*= B :- var(A), var(B), !, A=B.

do(X, A+B) :- !, do_sum(X,A,B).
do(X, A-B) :- !, do_sub(X,A,B).
do(X, A*B) :- A==B, !, do_square(X,A).
do(X, A*B) :- !, do_mult(X,A,B).
do(X, A//B) :- !, do_div(X,A,B).
do(X, X).

:- block do_sum(?,-,-), do_sum(-,?,-), do_sum(-,-,?).

do_sum(X,A,B) :- var(X), nonvar(A), nonvar(B), !, X is A+B.
do_sum(X,A,B) :- nonvar(X), nonvar(A), nonvar(B), !, X =:= A+B.
do_sum(X,A,B) :- nonvar(X), var(A), nonvar(B), !, A is X-B.
do_sum(X,A,B) :- nonvar(X), nonvar(A), var(B), !, B is X-A.

:- block do_sub(?,-,-), do_sub(-,?,-), do_sub(-,-,?).

do_sub(X,A,B) :- var(X), nonvar(A), nonvar(B), !, X is A-B.
do_sub(X,A,B) :- nonvar(X), nonvar(A), nonvar(B), !, X =:= A-B.
do_sub(X,A,B) :- nonvar(X), var(A), nonvar(B), !, A is X+B.
do_sub(X,A,B) :- nonvar(X), nonvar(A), var(B), !, B is A-X.

:- block do_square(-,-).

do_square(X,A) :- nonvar(X), X < 0, !, fail.
do_square(X,A) :- var(X), nonvar(A), !, X is A*A.
do_square(X,A) :- nonvar(X), nonvar(A), !, X =:= A*A.
do_square(X,A) :- !, D is sqrt(X), C is integer(D), X is C*C,
                  (A is C; A is -C).
do_square(X,A) :- !, fail.

:- block do_mult(?,-,-), do_mult(-,?,-), do_mult(-,-,?).

do_mult(X,A,B) :- var(X), nonvar(A), nonvar(B), !, X is A*B.
do_mult(X,A,B) :- nonvar(X), nonvar(A), nonvar(B), !, X =:= A*B.
do_mult(X,A,B) :- nonvar(X), var(A), nonvar(B), !, A is X//B, A*B =:= X.
```

```
do_mult(X,A,B) :- nonvar(X), nonvar(A), var(B), !, B is X//A, A*B =:= X.

:- block do_div(?,-,-), do_div(-,?,-), do_div(-,-,?).

do_div(X,A,B) :- var(X), nonvar(A), nonvar(B), !, X is A//B.
do_div(X,A,B) :- nonvar(X), nonvar(A), nonvar(B), !, X =:= A//B.
do_div(X,A,B) :- nonvar(X), var(A), nonvar(B), !, A is X*B.
do_div(X,A,B) :- nonvar(X), nonvar(A), var(B), !, B is A//X.
```