11pt
15pt 15pt

Formal Aspects of Computing (2003) 3: 2–??
© 2003 BCS

Running Programs Backwards: the Logical Inversion of Imperative Computation

Brian J. Ross
Brock University, Department of Computer Science, St Catharines, Ontario, L2S 3A1, Canada

Keywords: inversion, abduction, reverse computation, program testing, logic programming

Abstract. Imperative programs can be inverted directly from their forward–directed program code with the use of logical inference. The relational semantics of imperative computations treats programs as logical relations over the observable state of the environment, which is taken to be the state of the variables in memory. Program relations denote both forward and backward computations, and the direction of the computation depends upon the instantiation pattern of arguments in the relation. This view of inversion has practical applications when the relational semantics is treated as a logic program. Depending on the logic programming inference scheme used, execution of this relational program can compute the inverse of the imperative program. A number of nontrivial imperative computations can be inverted with minimal logic programming tools.

1. Introduction

This paper proposes that the inverse computations of imperative programs can be obtained directly from their forward–directed source code with the use of logical inference. It is well established in work in program semantics and verification that an imperative program can be modeled by a logical relation that declaratively describes the computational behaviour with respect to the observable environment. If the Horn clause subset of first-order logic is used as the relational logic language, then a program's logical semantics can be interpreted as a logic program [Llo87, CvE81, Ros89]. Because logic programs have associated models of computation, the logic program model of the imperative program may have an inference strategy applied to it, resulting in a simple interpreter

Correspondence and offprint requests to: Brian J. Ross, Brock University, Department of Computer Science, St Catharines, Ontario, L2S 3A1, Canada. bross@cosc.brocku.ca for the language. The relevance of this to program inversion is that, if an appropriate inference strategy is chosen, many non-trivial imperative algorithms are readily inverted via inference on their relational semantics. Additionally, because logic programming semantics are inherently nondeterministic in nature, nondeterministic inversions are handled as well.

A review of inversion, as well as relevant logic programming topics, is given in section 2. Section 3 presents a means for computing imperative inversions using logical programming inference. An example program inversion is given in section 4. A discussion concludes the paper in section 5.

2. Background

2.1. The inversion problem

Function inversion is the process of deriving for some one-to-one function $f : X \rightarrow Y$ the inverse function $f^{-1} : Y \rightarrow X$ such that $f^{-1}(f(x)) = x$. For example, if $f(x) = x^2$, then $f^{-1}(x) = \sqrt{x}$. Because there are two values $(\pm\sqrt{x})$ in this inverse, the inverse function is best denoted by a relation over the domain and range. Even though the existence of f^{-1} may be certain, the derivation of an effective computation procedure for any general f^{-1} can be difficult to determine. The existence of a decision procedure for computing mathematical inverses would enable the solution of many unsolved problems in mathematics.

Program inversion is similar to function inversion. Conventional computation involves the calculation of result values σ_f given some initial data σ_i using a program P:

$$P(\sigma_i) \rightarrow \sigma_f$$

We call such computations *forward–directed. Computational inversion* or *reverse computation* is backward-directed computation.

Definition 2.1. Given a forward-directed program P such that $P(\sigma_i) \rightarrow \sigma_f$, an *inverted program* P^{-1} is one that computes the inverse of P:

$$P^{-1}(\sigma_f) \rightarrow \sigma_i$$

P^{-1}'s computation is called an *inverse computation* of P.

Program inversion refers to computing inverse computations as well as to deriving inverted programs from forward–directed source programs (ie. how to derive P^{-1} from a given P).

Work has been done on the formal derivation of inverted programs from forward–directed programs [Dij82, Gri81, CU90, GvdS90, vW91, vdS93]. For example, Von Wright suggests a set of *converse commands* that can be symbolically applied to the constructs of a program to derive a specification requirement for its inverse [vW91]. An example is the inversion of a single destructive assignment statement which, if viewed outside of the context of other program components, yields an indefinite number of prior states for the variable's previous value. Von Wright models this with:

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

The value d that u contained prior to the assignment is determined using *angelic* *nondeterminism* [Hoa85]. As discussed above, because e may be a generalized mathematical expression, computing its inverse may be difficult. The inverted program is only implementable in a deterministic language when angelic nondeterminism is replaced with deterministic constructs, which can be a significant challenge, and probably impossible for NP–complete problem instances. Since most contemporary imperative languages used in industry (C, Fortran, Cobol) are strictly deterministic, inverses of programs written in these languages using the techniques of [Dij82, Gri81] *et al* are implementable in these same languages only when the inverses are likewise deterministic.

Program version is an important and useful concept within different software disciplines. Some representative theoretical papers on Turing machine inversion and reversible cellular automata are [McC56, Ben73, Ben82, TN90]. Broy uses program inverses to simplify the derivation of particular types of recursive structures during program synthesis [BKB80]. Inversion is used in functional programming environments, such as Lisp [Kor81], fold and unfold transformation systems [Dar81], and Hope+ [HH86]. Programming environments make use of "undo" operations, which are inversions of atomic operations [Lee86].

2.2. Logic programming

We presume familiarity with classical first–order logic. A first–order theory is defined by an alphabet, a first–order language, axioms, and inference rules. The first–order theory of *logic programming* is defined as follows. See [Llo87] for a detailed treatment.

The alphabet consists of variables, constants, function symbols of arity ≥ 1 (those of arity 0 are considered to be constants), predicate symbols of arity ≥ 0, the standard logical connectives $\wedge$, $\vee$, $\leftarrow$ and $\neg$, logical quantifiers $\forall$ and $\exists$, and punctuation symbols "(", ")", "." and ",". Terms are inductively defined: a variable is a term; a constant is a term; and if f is a function of arity n and t_1,...,t_n are terms, then $f(t_1, ..., t_n)$ is a term. An atomic formula or *atom* is a formula $p(t_1, ..., t_n)$ where p is a predicate symbol of arity n and t_1,...,t_n are terms.

The language of logic programs uses the Horn clause subset of first–order logic, which is defined as follows.

Definition 2.2. A *definite program clause* is a first-order sentence of the form

$$\forall(A \vee \neg B_1 \vee \cdots \vee \neg B_n)$$

where A and B_i are atoms, $n \geq 0$, and all the variables in the expression are universally quantified. A is called the *head* and the B_i are the *body.* This is also denoted by "$A \leftarrow B_1, \cdots, B_n.$".

Definition 2.3. A *unit clause* is a clause of the form "$A \leftarrow$", in which the body is empty. This is also denoted by "$A.$".

Definition 2.4. All the program clauses with the sam head M and arity n are said to comprise *predicate* M of arity n.

Definition 2.5. A *definite (program) query* or *definite goal* is a clause of the form

$$\forall(\neg B_1 \vee \cdots \vee \neg B_n) \quad \equiv \quad \neg\exists(B_1 \wedge \cdots \wedge B_n)$$

It is also denoted by "$\leftarrow B_1, \cdots, B_n$.".

Definition 2.6. The *empty clause*, denoted by ϵ, has an empty head and body. It represents a contradiction.

Definition 2.7. A *Horn clause* is either a definite program clause or a definite goal.

Definition 2.8. A *logic program* is a set of Horn clauses.

Any goal clauses in a logic program represent queries to verify with respect to the program theorems.

Definition 2.9. A *variant* of a clause C is a new clause C' obtained by renaming all the variables in C. In logic programming, an infinite set of variable names is presumed to exist for renaming purposes.

Definition 2.10. A *ground term (atom)* is a term that does not contain variables.

Definition 2.11. The *Herbrand universe* U_P of a logic program P is the set of all ground terms that can be formed from the constants and function symbols appearing in P.

Definition 2.12. The *Herbrand base* B_P of a logic program P is the set of all ground atoms that can be formed from the predicate symbols in P with ground terms from U_P as arguments.

Definition 2.13. The *Herbrand interpretation* $\mathcal{I}_P$ for a logic program P is a logical interpretation in which: the domain is U_P; constants in P are assigned themselves in U_P; for each n-ary function f in P, the mapping from $(U_L)^n$ to U_L defined by $(t_1, ..., t_n) \rightarrow f(t_1, ..., t_n)$ is assigned to f.

Definition 2.14. (*SLD–resolution*) Let the query G be $\leftarrow A_1, ..., A_m, ..., A_k$ and program clause C be $A \leftarrow B_1, ..., B_q$. The *resolvent* G' is derived from G and C using mgu θ if the following hold:

1. A_m is designated the *selected atom* in G.
2. θ is the *most general unifier (mgu)* of A_m and A: it is a substitution of the variables in A_m and A such that $A_m\theta$ and $A\theta$ are equivalent. In addition, for all such unifiers σ of A_m and A, there exists a substitution γ such that $\sigma = \theta\gamma$.
3. G' is the goal $\leftarrow (A_1, ..., A_{m-1}, B_1, ..., B_q, A_{m+1}, ..., A_k)\theta$.

Definition 2.15. An *SLD–derivation* of program P and goal G is a finite or infinite sequence $(G = G_0, G_1, ...,)$ of goals derived through SLD–resolution: a sequence $C_1, C_2, ...$ of variants of clauses from P, and a sequence $(\theta_1, \theta_2, ...)$ of mgu's such that G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1}. The variants are created so that clause C_i does not have any variables already used in the derivation up to G_{i-1}.

Definition 2.16. An *SLD–refutation* of P and G is a finite SLD–derivation of P and G that has the empty clause ϵ as the last goal in the derivation.

The premise behind SLD–derivations is the following. We wish to know the validity of a conjunction of atoms $\exists(A_1 \wedge ... \wedge A_k)$. If SLD–derivation performed on the negation of this expression derives ϵ, then it is equivalent to *false*, and so the original expression $\exists(A_1 \wedge ... \wedge A_k)$ is valid. which implies that there is a logical contradiction, and so the original query $\exists(A_1 \wedge ... \wedge A_k)$ must be true. Hence, this approach to inference is also termed *SLD–refutation.* The power of logic programming is that, not only can the refutation be performed on the original query, but the final mgu unifier θ of variable values for which the query is true is computed as as well. Notationally,

$$P \cup G \vdash_{SLD} \epsilon \quad \text{iff} \quad P \models G\theta$$

for some computed answer substitution θ. The intended interpretation $\mathcal{I}$ for the right–hand expression is taken to be U_P.

Theorem 2.1. (Soundness of SLD–resolution) If P is a logic program and G a goal, then every computed answer substitution θ using SLD–resolution is a correct answer for P and G.

Proof. See [Cla79, Llo87].

Theorem 2.2. (Completeness of SLD–resolution) For every correct answer for $P \cup \{G\}$ there exists a computed answer for it via SLD–resolution.

Proof. See [Cla79, Llo87].

Theorem 2.3. (Semi–decidability of SLD–resolution) There is no decision procedure for obtaining terminating inferences for general program goals and programs.

Proof. Horn clause logic is Turing powerful. (See [Hog90].)

It is convenient to denote logic programs directly within predicate logic derivations, without explicitly using the notation for SLD–resolution. The following definition denotes logic programs in such a form.

Definition 2.17. (program completion) The *completion Comp(P)* of a logic program P is one in which:

1. Every clause $p(t_1, ..., t_n) \leftarrow L_1, ..., L_m$ is transformed into
$$p(X_1, ..., X_n) \leftarrow \exists Y_1 ... \exists Y_d : X_1 = t_1, ..., X_n = t_n, L_1, ..., L_m$$
where X_i are new logical variables, Y_i are logical variables local to the body goals $L_1, ...L_m$, and $=$ is an equality theory equivalent to that which yields mgu's under SLD–resolution (see [Llo87]).
2. For all clauses of a predicate,
$$p(X_1, ..., X_n) \leftarrow E_1.$$
$$\cdots$$
$$p(X_1, ..., X_n) \leftarrow E_k.$$
transformed as in step 1 above, then the expression denoting this predicate is
$$\forall X_1 ... \forall X_n : (p(X_1, ..., X_n) \leftrightarrow E_1 \wedge ... \wedge E_k)$$

The notation $Comp(P)$, for example $Comp(while)$, refers to the completed transformation of predicate P. It will usually be used when performing logical substitutions in derivations. Quantification and argument unification goals will be relaxed if no confusion will arise.

Logic programming languages have characteristic inference strategies that are determined by the following definitions.

Definition 2.18. The *computation rule* (or *selection rule*) is the strategy used for selecting the next goal within a query for resolution.

The *independence of the computation rule* says that all possible answers for program P and goal G are obtainable using any computation rule.

Definition 2.19. The *search rule* is the criteria used for selecting the clause to resolve with the selected goal.

Practically speaking, the search rule prioritizes the clauses to use during resolution, which determines the ordering of answers obtained. Since some derivations may be infinite, the search rule determines when non–terminating derivations are encountered.

Definition 2.20. *Backtracking* is a mechanism by which the inference system can revert back to a previous point in the SLD–derivation and perform another alternative resolution step.

Actual implementations of logic programming languages such as Prolog [CM87] use a restricted inference strategy that is efficiently implemented on conventional hardware. Given a query "$\leftarrow B_1, \cdots, B_n$", Prolog's computation rule selects the first goal B_1. Prolog's search rule is to use the first clause that unifies with the goal, where the order is determined by the textual order within the program file. Prolog's backtracking strategy is to revert to the last point in the derivation in which a clause selection was made by the search rule and to try the *next* subsequent clause that unifies with the selected goal. This backtracking is applied exhaustively through the inference. The net effect of Prolog's control strategy is that the computation tree is searched depth–first and exhaustively. A disadvantage is that it is an incomplete strategy — inferences can easily dive down nonterminating branches of the tree.

2.2.1. Logic program inversion

Logic programs inherently support inversion [Sic79, SM84]. The use of an appropriate inference procedure permits the determination of any relation represented within the program's declarative logical semantics. Consider the Prolog *append* predicate:

$$\begin{array}{l} append([\ ], A, A). \\ append([A|B], C, [A|D]) \ \leftarrow \ append(B, C, D). \end{array}$$

The goal

$$\leftarrow append([a], [b, c], Z)$$

infers $Z = [a, b, c]$, while the goal

$$\leftarrow append(X, [b, c], [a, b, c])$$

infers $A = [a]$. Therefore the inferences of these two goals represent inverses of one another. 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 no inherent direction is encoded within pure logic programs, and program inversions are computed as a natural by–product of SLD–resolution's soundness and completeness.

As mentioned earlier, logic programming language implementations use incomplete inference schemes, such as Prolog's left–to–right depth–first control:

not all valid solutions are necessarily computable, due to non–termination down infinite branches of the computation tree. Consequently, Prolog predicates are often *directed* — they expect particular instantiation patterns of arguments in order to terminate, as well as to execute built–in predicates correctly. This is akin to the direction encapsulated in imperative computations, although it is not as acute, since a Prolog program usually has both deterministic and nondeterministic components. Shoham and McDermott made the notion of direction more precise with the following definitions [SM84].

Definition 2.21. Let P be a logic program with Herbrand interpretation $\mathcal{I}$, *R(*X_1*,...,*X_n*)* be a predicate of arity n from P, and let V be $\cup X_i$ of variables X_i. R is a *function from* $V1$ *to* $V2$ if $< V1, V2 >$ is a partition of V, and for all instantiations of V_1, interpretating R with Prolog's inference strategy will generate all the solutions $V2$ consistent with relation R wrt $\mathcal{I}$.

Definition 2.22. A Prolog predicate R is *D–directed* if D is a set of variable partition tuples $\{< V1_i, V2_i >\}$ for R, and R is a function from $V1_i$ to $V2_i$ for all i.

Definition 2.23. A Prolog predicate R is *complete* if it is D–directed for D, the set of all variable partitions for R.

A complete predicate is therefore one that is fully invertible. Although pure logic programs are conceptually complete in this sense, many predicates typically are not when restricted inference strategies such as Prolog's are considered.

2.3. Abductive Reasoning in Logic Programs

Abduction is a style of logical inference that first conjectures a particular hypothesis and then attempts to establish its premises [HHNT86]. Abductive reasoning can be applied to the program inversion problem if one considers inverted computation in the following way. Consider execution of a program P:

$$\sigma_i = \sigma_1 \rightarrow \sigma_2 \rightarrow \sigma_3 \rightarrow \cdots \rightarrow \sigma_k = \sigma_f$$

Each transition represents an atomic alteration of the environment or store σ with respect to the execution of P, and the final state σ_f is found after $k-1$ transitions. To invert such a computation, the inverted program P^{-1} must treat each σ_i as a state that must be logically consistent as a state following σ_{i-1} with respect to P. To invert P, an abductive approach would be to assert that some σ_k is a conjectured final state and then establish the premise that σ_{k-1} is a valid state preceding σ_k. If this reasoning is repeatedly applied to all the intermediate states $\sigma_1, ..., \sigma_{k-1}$ leading up to σ_k, then an inversion has been determined for that σ_k.

Section 3 will use abduction for inverting the *while*–loops of an imperative language. The logical construction to be used was suggested by Brough and Hogger [BH91]; it is based on the Greibach-Foster grammatical transformation. Consider the following logic program predicate schema:

$$P(X) \leftarrow R(X).$$
$$P(X) \leftarrow P(Y), Q(Z).$$

where X, Y and Z are arbitrary argument tuples. When executed with Prolog's left–to–right control rule, the goal "$\leftarrow P(X)$." will often fall into an infinite loop with the second clause (and assuredly so if X are distinct variables). This is because $P(Y)$ recurses before $Q(Z)$ has a chance to establish additional computational constraints on the inference. This is known as *left–recursion* in logic programming. Brough and Hogger suggest the following transformation to resolve this problem.

Definition 2.24. (Forward–simulation transform, or FST) Given a predicate matching the schema P,

$$\begin{array}{ll} P_1: & P(X) \leftarrow R(X). \\ P_2: & P(X) \leftarrow P(Y), Q(Z). \end{array}$$

the FST for P, or FST(P), is

$$\begin{array}{ll} C_1: & P'(T) \leftarrow R(X'), s(X', T). \\ C_2: & S(T, T). \\ C_3: & S(Y, T) \leftarrow Q(Z), S(X, T). \end{array}$$

where X, Y and Z are identical to those in P, while S' and T are mutually-exclusive vectors of new and distinct variables.

The abductive reasoning behind the FST is as follows. In order for P to have inferred a terminating solution, the goal $R(X)$ must be applied as the final goal: clause P_1 is the only means for the inference of P to terminate. The FST immediately establishes that $R(X)$ is required as first premise in $P'(X)$ in clause C_1. Secondly, each recursive call to P in clause P_2 must involve a successful resolution of Q; should any call to Q fail, then the whole clause fails. The FST makes this explicit by presupposing these successful resolutions of R *before* P is recursively invoked (in clause C_3). Finally, clause C_2 denotes a termination of the loop. The FST results in an inverted execution of P, by abductively starting with the conditions necessary for a termination of the left–recursive loop *a priori* and then iteratively unwrapping the loop until the loop is successfully inferred. Note that the FST is generalizable by replacing R and Q with multiple conjunctions of goals.

Theorem 2.4. (Correctness of FST) For predicates P and $P' = FST(P)$, the relations defined by each are equivalent:

$$\models \ \forall (P \ \leftrightarrow \ P')$$

Proof. Induction on length of inference of P and P' (see [BH91]).

3. A Relational Semantics of Inverted Imperative Computation

A simple imperative *while*–language $\mathcal{L}$ will be used (see Figure 1), due to its conventionality with those in industry, as well as its computational equivalence with primitive–recursive functions and hence Turing machines [HU79]. An operational semantics following one in [SA91] follows; see that source for additional descriptions of this style of semantics.

Definition 3.1. (Operational semantics of $\mathcal{L}$) Let $\alpha \in \mathcal{L}$ be a program, and σ be a state consisting of the valuations of all the program variables. An interpreter function $I_{\mathcal{L}} : (\alpha, \sigma) \rightarrow (\alpha', \sigma')$ is defined as follows:

$$\begin{array}{lcl}
\textit{Prog} & ::= & \textit{program}\ ([\bar{x}]) \text{: } \{\textit{Statement}\} \\
\textit{Statement} & ::= & \textit{Statement}\ ;\ \textit{Statement} \\
 & | & \text{x} := e \\
 & | & \text{if } (e_b) \text{ then } \{\textit{Statement}\} \text{ else } \{\textit{Statement}\} \\
 & | & \text{while } (e_b)\ \{\textit{Statement}\} \\
 & | & \epsilon
\end{array}$$

where e is an arithmetic expression
e_b is a Boolean expression
$\bar{x}$ is a list of program variables

Fig. 1. Syntax of $\mathcal{L}$

$$I_{\mathcal{L}}(program(\bar{x}) : \{A\},\ \sigma)\ =\ I_{\mathcal{L}}(A,\ \sigma)$$

$$I_{\mathcal{L}}(\epsilon,\ \sigma)\ =\ (\epsilon,\ \sigma)$$

$$I_{\mathcal{L}}(x := t; A_2; ...; A_n,\ \sigma)\ =\ I_{\mathcal{L}}(A_2; ...; A_n,\ \sigma(x/Val_\sigma(t)))$$

$$I_{\mathcal{L}}(if(e_b)then\{A_1\}else\{A_2\}; A_3; ...; A_n,\ \sigma) = \begin{cases} I_{\mathcal{L}}(A_1; A_3; ...; A_n,\ \sigma) & : if\ \ Val_\sigma(e_b) = true \\ I_{\mathcal{L}}(A_2; A_3; ...; A_n,\ \sigma) & : if\ \ Val_\sigma(e_b) = false \end{cases}$$

$$I_{\mathcal{L}}(while(e_b)\{A_1\}; A_2; ...; A_n,\ \sigma) = \begin{cases} I_{\mathcal{L}}(A_1; while(e_b)\{A_1\}; A_2; ...; A_n,\ \sigma) & : if\ \ Val_\sigma(e_b) = true \\ I_{\mathcal{L}}(A_2; ...; A_n,\ \sigma) & : if\ \ Val_\sigma(e_b) = false \end{cases}$$

In the above, function Val_σ evaluates $\mathcal{L}$–terms with respect to the current state σ, and A_i is a statement of $\mathcal{L}$. The notation $x/Val_\sigma(t)$ denotes a new state σ' in which variable x's value is replaced with the valuation of t wrt σ.

The operational semantics takes the form of a *stack machine*, in which program code is saved in the first argument of the tuple $(A,\ \sigma)$ and the current variable values are in the second argument. It is assumed that all expressions in assignments and boolean tests are *well–defined* — there is a computable solution for every expression and state. Note that nontermination may arise with *while* statements, while a terminated computation occurs only when the code argument reduces to ϵ.

A translation between $\mathcal{L}$ source code and its relational semantics $\mathcal{R}$ is in figure 2. The $\mathcal{R}$ semantics defines a logic program. The same set of logical connectives and syntax as described in section 2.2 is used. Function terms are translated in an obvious way from the arithmetic and boolean expressions used in P. All the predicates are of arity 2, and the predicate names used in Figure 2 are freely indexed so that each program construct is modeled by a unique predicate. (Identical program constructs can be modeled by the same predicate if desired). A relation *asgn_eq* is also introduced, and is described below.

$\mathcal{R}$ defines input–output relations over the environment σ for all program constructs. For all relations $R(\sigma_i, \sigma_f)$ for $\mathcal{L}$ constructs A (other than s used in (vi)), the initial state upon executing A is σ_i, and the state upon terminating execution is σ_f. As with the operational semantics, all assignment and boolean expressions are presumed to be well–defined. Briefly, (ii) defines the main predicate for a program. Sequenced statements are modeled in (iii) by linking the input–output

(i) Empty:

$$\epsilon \overset{\text{def}}{=} \models \sigma_i = \sigma_f \quad (\equiv true)$$

(ii) Programs:

$$program(\bar{x}) : \{ A \} \overset{\text{def}}{=} \models program([\bar{x}_i], [\bar{x}_f]) \leftarrow A([\bar{x}_i], [\bar{x}_f]).$$

(iii) Chains:

$$S_1; S_2; ...; S_k \;\; (\equiv A) \overset{\text{def}}{=} \models S_1(\sigma_i, \sigma_2) \wedge S_2(\sigma_2, \sigma_3) \wedge ... \wedge S_k(\sigma_k, \sigma_f) \\ (\equiv A(\sigma_i, \sigma_f))$$

(iv) Assignment:

$$x := e \overset{\text{def}}{=} \models asgn\_eq(\sigma_i, x_f, e, \sigma_f)$$

(v) Tests:

$$\begin{array}{lcl} if\ (e_b)\ then\ \{ A_1 \} & \overset{\text{def}}{=} & \models \quad if_j(\sigma_i, \sigma_f) \leftarrow e_b \wedge A_1(\sigma_i, \sigma_f). \\ \quad else\ \{ A_2 \} & & \wedge \quad if_j(\sigma_i, \sigma_f) \leftarrow \neg e_b \wedge A_2(\sigma_i, \sigma_f). \end{array}$$

(vi) While loops:

$$\begin{array}{lcl} while\ (e_b)\ \{ A \} & \overset{\text{def}}{=} & \models (while_j(\sigma_i, \sigma_f) \leftarrow \neg e_b(\sigma_f) \wedge s_j(\sigma_f, \sigma_i)). \\ & & \wedge \quad s_j(\sigma_i, \sigma_f) \leftarrow \sigma_i = \sigma_f. \\ & & \wedge(\;\; s_j(\sigma_f, \sigma_i) \leftarrow A(\sigma_1, \sigma_f) \wedge e_b(\sigma_1) \wedge s_j(\sigma_1, \sigma_i)). \end{array}$$

Fig. 2. Relational semantics $\mathcal{R}$

relations for the chained statement relations: the output state for a statement's relation is the input state for the following statement's relation. A test is modeled in (v) by a unique predicate with two clauses, where each clause accounts for the separate test cases.

In (iv), *asgn_eq* is an arithmetic equality relation with respect to the intended Herbrand interpretation $\mathcal{I}_P$ being used by the logic program. In *asgn_eq(σ_i, x_f, e, σ_f)*, the final value $x_f \in \sigma_f$ of variable x is the value of expression e with respect to $\mathcal{I}$ and the state σ_i at the commencement of the assignment. The reference to x's initial value x_i, lost after the assignment, is referenced within σ_i, and possibly within e and other relations within the predicate in question. It is important to note that the *invertibility* of a given assignment expression is not guaranteed: although *asgn_eq* may denote it, its computability might not be possible during an inference proof (see section 2.1). The computability of all expression inverses are dependent upon the problem at hand.

Finally, *while*–loops are modeled in (vi). This predicate uses the forward–simulation transform of section 2.3, applied to a predicate modeling forward–directed *while*–loop "*while* (e_b) { A }":

$$\begin{array}{ll} \models & while_j(\sigma_i, \sigma_f) \leftarrow e_b(\sigma_i) \wedge A(\sigma_i, \sigma_2) \wedge while_j(\sigma_2, \sigma_f). \\ & \wedge \;\; while_j(\sigma_i, \sigma_i) \leftarrow \neg e_b(\sigma_i). \end{array}$$

This relation matches the schema required by the FST, and by theorem 2.4, it is logically equivalent to the transformation used in (vi) in Figure 2. The rationale for using the FST is an operational one: the control introduced by the FST abductively inverts forward–directed *while*–loops. Basically, the inverted *while* presumes that the loop terminates (the negated test) and then iteratively inverts the loop body's execution with the s predicate. Note that the first abductive s clause can be abbreviated "$s_j(\sigma,\sigma)$.", denoting that the initial and final states are identical. We call $s(\sigma,\sigma)$ the *base clause*, and the other the *iterative clause*.

In logical derivations to follow, each step's comments describe the operation performed to derive that exprssion. If necessary, the term to be expanded in the next step is underlined. It is assumed that $\models$ is with respect to the Herbrand interpretation of the logic program P, and $\vdash$ is with respect to logic program P. $\vdash_{SLD}$ denotes the use of SLD–resolution as the inference rule, while $\vdash$ alone denotes a general logical derivation step.

Theorem 3.1. (Equivalence of relational and operational semantics) Let P be an $\mathcal{L}$–program, and $\mathcal{R}(P) = P(\sigma_i,\sigma_f)$ its relational semantics. Then for all states σ_i:

1. If $\models \exists\sigma_f : P(\sigma_i,\sigma_f)$, then $I_{\mathcal{L}}(P,\sigma_i)$ derives (ϵ,σ_f) in a finite number of steps.
2. If $\not\models \exists\sigma_f : P(\sigma_i,\sigma_f)$, then $I_{\mathcal{L}}(P,\sigma_i)$ never derives a pair (ϵ,σ_x) for any σ_x.

Proof. 1. The correspondence between $\mathcal{R}(P)$ and $I_{\mathcal{L}}(P,\sigma)$ is straight–forward for program headers (which define the initial state) and assignments. For *while* loops, the proof uses induction on the size of the inference corresponding to completed resolutions of *while* loop iterations.

The base case is when the loop test is initially false, and 0 iterations occur of the loop body. The relational derivation of this is as follows.

$$\begin{array}{ll}
\models \exists\sigma_f : \underline{while(\sigma_i,\sigma_f)} & : assumption \\
\vdash \exists\sigma_f : \neg e(\sigma_f) \wedge \underline{s(\sigma_f,\sigma_i)} & : subst.\ Comp(while) \\
\vdash \exists\sigma_f : \neg e(\sigma_f) \wedge (\underline{\sigma_i = \sigma_f} \vee (A(\sigma_1,\sigma_f) & \\
\qquad \wedge e(\sigma_1) \wedge s(\sigma_1,\sigma_i))) & : subst.\ Comp(s) \\
\vdash \exists\sigma_f : \neg e(\sigma_f) \wedge (\underline{\sigma_i = \sigma_f} \vee false) & : assume\ 0\ iterations,\ simplify \\
\vdash \neg e(\sigma_i) & : simplify
\end{array}$$

The corresponding derivation of $I_{\mathcal{L}}$ is:

$$I_{\mathcal{L}}(while(e)\{A\},\sigma_i) \ = \ (\epsilon,\sigma_i)$$

where $Val_{\sigma_i}(e) = false$ in order for 0 iterations of the loop. This correlates with the relational semantics, and is derived in 1 step.

Assuming the hypothesis holds for $k-1$ iterations of the loop ($k > 1$), then for k iterations, the relational inference is:

$$\begin{array}{lll}
\models & \exists\sigma_f : \underline{while(\sigma_i,\sigma_f)} & \\
\vdash & \exists\sigma_f : \neg e(\sigma_f) \wedge A(\sigma_1,\sigma_f) \wedge e(\sigma_1) \wedge \underline{s(\sigma_1,\sigma_i)} & : (1) \\
\vdash & \ldots & \\
\vdash & \exists\sigma_f : \neg e(\sigma_f) \wedge A(\sigma_1,\sigma_f) \wedge e(\sigma_1) \wedge A(\sigma_2,\sigma_1) \wedge e(\sigma_2) \wedge ... & \\
 & \quad \wedge e(\sigma_{k-1}) \wedge A(\sigma_k,\sigma_{k-1}) \wedge e(\sigma_k) \wedge \underline{s(\sigma_k,\sigma_i)} & : (2) \\
\vdash & \exists\sigma_f : \neg e(\sigma_f) \wedge ... \wedge A(\sigma_k,\sigma_{k-1}) \wedge e(\sigma_k) \wedge \underline{\sigma_k = \sigma_i} & : (3) \\
\vdash & \exists\sigma_f : \underline{\neg e(\sigma_f) \wedge A(\sigma_1,\sigma_f) \wedge e(\sigma_1) \wedge ...} & \\
 & \quad \underline{\wedge A(\sigma_{k-1},\sigma_{k-2}) \wedge e(\sigma_{k-1})} \wedge A(\sigma_i,\sigma_{k-1}) \wedge e(\sigma_i) & : (4) \\
\vdash & \exists\sigma_f : \underbrace{\underline{while(\sigma_{k-1},\sigma_f)}}_{(\dagger)} \wedge \underbrace{\underline{A(\sigma_i,\sigma_{k-1}) \wedge e(\sigma_i)}}_{(\ddagger)} & : (5)
\end{array}$$

Descriptions of the steps in the above proof are as follows.

(1) Substitute $Comp(while)$, simplify using iterative clause, since $k > 0$.
(2) Repeat above k times.
(3) Substitute base clause of Comp(s).
(4) Simplify.
(5) Fold with *while*.

$I_\mathcal{L}$ derives the following:

$$\begin{array}{ll} I_\mathcal{L}(while(e)\{A\}, \sigma_i) & \\ = \ I_\mathcal{L}(A; while(e)\{A\}, \sigma_i) & : presumption \ \ k > 0, \ \ so \ \ Val_{\sigma_i}(e) = true \\ = \ I_\mathcal{L}(while(e)\{A\}, \sigma_{k-1}) & : I_\mathcal{L}(A, \sigma_i) = (\epsilon, \sigma_{k-1}) \ \ by \ \ ind. \ \ hyp. \ and \ (\ddagger) \\ = \ (\epsilon, \sigma_f) & : ind. \ \ hyp. \ \ and \ \ (\dagger) \end{array}$$

By the induction hypothesis, each of $I_\mathcal{L}(A, \sigma_i)$ and $I_\mathcal{L}(while(e)\{A\}, \sigma_{k-1})$ are derived in a finite number of steps, and hence so is the entire derivation.

Similar reasoning applies to chains and test statements, and is omitted.

2. Since all assignment and boolean expressions are well–defined, the only way in which an $\mathcal{L}$–relation can be undefined is if a non–terminating *while*–loop is being modelled. Consider the relation for a non–satisfiable *while* relation:

$$\begin{array}{lll} \not\models & \forall\sigma_i\exists\sigma_f : while(\sigma_i, \sigma_f) & \\ \vdash & \neg\forall\sigma_i\exists\sigma_f : while(\sigma_i, \sigma_f) & \\ \vdash & \neg\forall\sigma_i\exists\sigma_f : (\neg e(\sigma_f) \wedge s(\sigma_f, \sigma_i)) & : subst. \ \ Comp(while) \\ \vdash & \exists\sigma_i\forall\sigma_f : \neg(\neg e(\sigma_f) \wedge s(\sigma_f, \sigma_i)) & : distr. \ \neg \ \ through \ \forall, \ \exists \\ \vdash & \exists\sigma_i\forall\sigma_f : (e(\sigma_f) \vee \neg s(\sigma_f, \sigma_i)) & : distr. \ \neg \\ \vdash & \exists\sigma_i : (\forall\sigma_f : e(\sigma_f)) \vee (\forall\sigma_f : \neg s(\sigma_f, \sigma_i)) & : distr. \ \forall \\ \vdash & (\forall\sigma_f : e(\sigma_f)) \vee (\exists\sigma_i\forall\sigma_f : \neg s(\sigma_f, \sigma_i)) & : distr. \ \exists, \ \ simplify \\ \vdash & (\forall\sigma_f : e(\sigma_f)) \vee false & : since \ \ \models \forall\sigma_f : s(\sigma_f, \sigma_f) \\ & & \ \ using \ \ base \ \ clause \ \ of \ \ s \\ \vdash & \forall\sigma_f : e(\sigma_f) & : simplify \end{array}$$

Next, assume that $I_\mathcal{L}(while(e)\{A\}, \sigma_i)$ derives (ϵ, σ_f) in a finite number of steps. To have terminated with this result, the step $I_\mathcal{L}(while(e)\{A\}, \sigma_f) = (\epsilon, \sigma_f)$ must have been taken, implying that $Val_{\sigma_f}(e) = false$ was determined. This contradicts the above relational result, and so $I_\mathcal{L}$ cannot terminate.

Theorem 3.2. (Computability of $\mathcal{L}$ inversions) Let P be an $\mathcal{L}$–program, and $\mathcal{R}(P) = P(\sigma_i, \sigma_f)$. For some final state σ_f, if $\models \exists\sigma_i : P(\sigma_i, \sigma_f)$, then $\vdash_{SLD} \leftarrow P(\sigma_i, \sigma_f)$.

Proof. Theorem 2.2.

Theorem 3.2 states that if an inverse exists for a particular program with a given final state, then SLD–resolution will infer it from the programs relational semantics. This is due to the completeness of SLD–resolution.

Theorem 3.3. (Verifiability of $\mathcal{R}$ relations)

$$\models P(\sigma_i, \sigma_f) \quad iff \quad \vdash_{SLD} \leftarrow P(\sigma_i, \sigma_f)$$

Proof. Theorems 2.1 and 2.2.

Theorem 3.3 is a very powerful result, as will be seen in section 4. It is essentially a product of the central tenet of the theory of logic programming. It is important from the context of imperative program analysis, however, since it implies that SLD–resolution can be used to verify the input–output relations of imperative programs.

Finally, the following theorem is a reminder that, although the previous results have practical importance in inferring imperative program inversions, we must nevertheless reconcile ourselves with fundamental computability limitations.

Theorem 3.4. (Undecidability of inferring $\mathcal{L}$ inversions) There is no decision procedure for solving generalized $\vdash_{SLD} \leftarrow P(\sigma_i, \sigma_f)$.

Proof. Theorem 2.3 states that SLD–resolution is semi–decidable. This carries over to the restricted logic programs that arise with the $\mathcal{R}$ semantics of $\mathcal{L}$ programs, because the *while* programs encodable in $\mathcal{L}$ are Turing powerful [HU79].

4. Example Inversion

```
$power([y, x, n])$ : {
    $y := 1$;
    $while$ ( $n > 0$ ) {
        $while$ ( $even(n)$ ) {
            $n := n/2$;
            $x := x * x$
        };
        $n := n - 1$;
        $y := y * x$
    }
}
```

Fig. 3. Binary powers

Consider program *power* in Figure 3, and its inverse relation in Figure 4. *Power* uses a binary powers algorithm which, given integer inputs x and n, efficiently computes $y = x^n$. Most of the code is straight forward. Forward execution with the input $[y = ?,\ x = 2,\ n = 2]$ computes a final state $[y = 4,\ x = 4,\ n = 0]$.

The relational semantics of *power* in Figure 3 correspond to the imperative source code and are automatically compilable from the source. The state comprises the values of the three variables y, x and n. Using standard Prolog notation [CM87], constants are denoted by lower–case literals, while logical variables are upper–case. Variable Env conveniently denotes an entire state in which individual state values are not referenced. There are two unique $while$ predicates corresponding to each $while$–loop. The chains in Figure 3 exploit the transitivity of conjunction by reversing the order of goals from their corresponding statements in the source program. Although any permutation of conjunctions is logically equivalent, this particular ordering will have ramifications on Prolog interpretation, as will be discussed below.

Before proceeding with some SLD inferences, it must be pointed out that expressions used within the *asgn_eq* relation in this example are assumed to be invertible. The integer arithmetic expressions used are: (i) $N2 = N/2$; (ii) $X2 = X * X$; (iii) $N2 = N - 1$; and (iv) $Y2 = Y * X$. These expressions are fully invertible: (i) is invertible for N when $N2$ contains a value, and similarly

$$
\begin{array}{ll}
c_1: & power([Y1, X1, N1],\ Env) \leftarrow \\
 & \quad while2([Y2, X1, N1],\ Env), \\
 & \quad asgn\_eq([Y1, X1, N1],\ Y2,\ 1,\ [Y2, X1, N1]). \\
 & \\
c_2: & while2(Env,\ [Y2, X2, N2]) \leftarrow \\
 & \quad \neg N2 > 0, \\
 & \quad s2([Y2, X2, N2], Env). \\
 & \\
c_3: & s2(Env,\ Env). \\
c_4: & s2([Y3, X2, N3],\ Env) \leftarrow \\
 & \quad asgn\_eq([Y2, X2, N3],\ Y3,\ Y2 * X2,\ [Y3, X2, N3]), \\
 & \quad asgn\_eq([Y2, X2, N2],\ N3,\ N2 - 1,\ [Y2, X2, N3]), \\
 & \quad while1([Y1, X1, N1],\ [Y2, X2, N2]), \\
 & \quad N1 > 0, \\
 & \quad s2([Y1, X1, N1],\ Env). \\
 & \\
c_5: & while1(Env,\ [Y2, X2, N2]) \leftarrow \\
 & \quad \neg even(N2), \\
 & \quad s1([Y2, X2, N2],\ Env). \\
 & \\
c_6: & s1(Env,\ Env). \\
c_7: & s1([Y1, X2, N2],\ Env) \leftarrow \\
 & \quad asgn\_eq([Y1, X1, N2],\ X2,\ X1 * X1,\ [Y1, X2, N2]), \\
 & \quad asgn\_eq([Y1, X1, N1],\ N2,\ N1/2,\ [Y1, X1, N2]), \\
 & \quad even(N1), \\
 & \quad s1([Y1, X1, N1],\ Env).
\end{array}
$$

Fig. 4. Binary powers inverse relation

for (ii) and (iii); and (iv) is invertible for either Y (or X) when $Y2$ and X (or $Y2$ and Y) have values. The arithmetic theories and corresponding computation schemes used to compute these inversions should be obvious, and details are omitted without distracting from the main focus of interest — the inversion of imperative control.

One SLD–inference is the following. Each line of the inference represents the current goal. To simplify the derivation, fresh logical variables are indexed as needed, and unifying substitutions are applied immediately. Underbraced letters temporarily denote terms.

$$\begin{array}{rcll}
1: & \leftarrow & power(Start,\ [4,4,0]). & : initial\ \ query \\
2: & \leftarrow & while2([Y_2,X,N],[4,4,0]), & \\
 & & \underbrace{asgn\_eq([Y,X,N],Y_2,1,[Y_2,X,N])}_{A}. & : c_1, Start=[Y,X,N], \\
 & & & \ Env=[4,4,0] \\
3: & \leftarrow & \neg 0>0, s_2([4,4,0],Env_2),\ A. & : c_2, Env_2=[Y_2,X,N] \\
4: & \leftarrow & s_2([4,4,0],Env_2),\ A. & : \neg 0>0 \vdash true \\
5: & \leftarrow & asgn\_eq([Y_4,4,0],4,Y_4*4,[4,4,0]), & \\
 & & asgn\_eq([Y_4,4,N_4]0,N_4-1,[Y_4,4,0]), & \\
 & & while1([Y_5,X_5,N_5],[Y_4,4,N_4]), & \\
 & & \underbrace{N_5>0, s2([Y_5,X_5,N_5],Env2)}_{B},\ A. & : c_4 \\
6: & \leftarrow & while1([Y_5,X_5,N_5],[Y_4,4,N_4]),\ B,\ A. & : 4=Y_4*4 \vdash Y_4=1, \\
 & & & \ 0=N_4-1 \vdash N_4=1 \\
7: & \leftarrow & \neg even(1), s1([1,4,1],Env_3),\ B,\ A. & : c_5, Env_3=[Y_5,X_5,N_5] \\
8: & \leftarrow & s1([1,4,1],Env_3),\ B,\ A. & : \neg even(1) \vdash true \\
9: & \leftarrow & 1>0, s2([1,4,1],Env2),\ A. & : c_6,\ Env_3=[1,4,1] \\
10: & \leftarrow & s2([1,4,1],Env2),\ A. & : 1>0 \vdash true \\
11: & \leftarrow & asgn\_eq([Y,4,1],1,1,[1,X,N]). & : c_3,\ Env_2=[1,4,1] \\
12: & \leftarrow & \epsilon & : 1=1 \vdash true
\end{array}$$

Carrying through the unifying substitutions to the original argument, the result $Start=[Y,4,1]$, or $x^n=4^1=4$, is inferred.

Another inference is possible. Continuing the above derivation from step 8:

$$\begin{array}{rcll}
8: & \leftarrow & s1([1,4,1],Env_3),\ B,\ A. & : from\ \ above \\
9': & \leftarrow & asgn\_eq([1,X_6,1],4,X_6*X_6,[1,4,1]), & \\
 & & asgn\_eq([1,X_6,N_6],1,N_6/2,[1,X_6,1]), & \\
 & & even(N_6), s1([1,X_6,N_6],Env_3),\ B,\ A. & : c_7 \\
10': & \leftarrow & even(2), s1([1,2,2],Env_3),\ B,\ A. & \\
 & & \multicolumn{2}{r}{: 4=X_6*X_6 \vdash X_6=2,\ 1=N_6/2 \vdash N_6=2} \\
11': & \leftarrow & s1([1,2,2],Env_3),\ B,\ A. & : even(2) \vdash true \\
12': & \leftarrow & 2>0,\ s2([1,2,2],Env2),\ A. & : c_6,\ Env_3=[1,2,2] \\
13': & \leftarrow & s2([1,2,2],Env2),\ A. & : 2>0 \vdash true \\
14': & \leftarrow & asgn\_eq([Y,2,2],1,1,[1,2,2]). & : c_3,\ Env_2=[1,2,2] \\
14': & \leftarrow & \epsilon & : 1=1 \vdash true
\end{array}$$

This infers the inverted computation $Start=[Y,2,2]$, or $x^n=2^2=4$.

Yet one more inference is possible from step $10'$ above, when $4=X_6*X_6 \vdash X_6=-2$ is inferred. The rest of this inference (omitted) computes the inverted computation $Start=[Y,-2,2]$, or $x^n=(-2)^2=4$.

Note that the above derivations imitate Prolog's computation strategy. Clauses are selected for unification in the order they reside in the translation in Fig. 3, and goals are selected for resolution from left to right. Indeed, the above inversions are automatically obtained using a standard Prolog interpreter [CM87]. The ordering of goals in the logic program in Fig. 3 permits inverted control and the *asgn_eq* and boolean test relations to be executed under Prolog's depth–first–left–first control strategy. The query "$\leftarrow power(StartEnv,[4,4,0])$" automatically infers the above three inverted results. Prolog's backtracking mechanism recovers from failed branches of the inference tree. Backtracking searches alternative nondeterministic branches as generated by the nondeterministic computations from *asgn_eq*, which yields both positive and negative integral square roots, as well as abductive unwinding of the loops. In addition, the query "$\leftarrow$ *power*$([Y,2,2],\ [4,4,0])$" returns *true* from the interpreter, confirming that this is a valid relation for *power*. Prolog can also perform some tests of negative relations. For example, "$\leftarrow$ *power*$(StartEnv,\ [4,4,1])$" returns *false*: abductive reasoning for the outer *while*–loop stipulates that $\neg N > 0$ in order for the loop to have terminated, which $N = 1$ in this query clearly violates. Finally, the query "$\leftarrow$ *power*$([Y,X,3],\ [4,4,0])$" also returns *false*, as an initial value of $N = 3$ will not correspond to the final values $Y = 4$ and $X = 4$ in this program relation.

5. Conclusion

The feasibility of computing the inversions of imperative programs using logic programming techniques has been shown. Correctness of inversions is guaranteed. Assuming that the logical semantics for the imperative language are sound, the translational semantics of the source program defines a theory of its behaviour. Since SLD resolution is sound, the inversions are sound logical inferences of the source program's logical semantics. Another advantage is the ability to obtain nondeterministic inversions. The logic programming model permits multiple initial values to be obtained for a given final value. Mixed forward and inverted computation is also conceivable. The approach is conducive to semi–automation, since logical predicates are compiled directly from the source program. Inferring an inverse from these predicates is the responsibility of the logic programming system. Finally, the approach permits interactive program analysis to be performed. An inverse relation permits interactive inspection of the input–output behaviour of the source program. As shown in section 4, this is a powerful analytical tool.

This paper's inversion technique has some advantages over formal derivation techniques in [Dij82, Gri81]. Logically inferred inversions are compiled automatically from the source program, and complex derivation proofs are unnecessary. These inversions are not restricted to deterministic target languages (although Dijkstra's guarded language has nondeterministic constructs). Unlike logical inversions, derived inverted programs are usually guaranteed to terminate. Establishing this, however, requires significant effort during their derivation. The completeness of our inversions depends entirely on the robustness of the inference system.

The tools used to invert the example in section 4 — standard Prolog control, abduction, and arithmetic equality — are capable of inverting some imperative algorithms, but are inadequate for most general cases. In fact, the abductive modeling of *while*–loops is only an aid in inverting loops and is not strictly required. Other imperative algorithms, including Fibonacci numbers, bubble sort, and Knuth's Algorithm P [Knu81], have been successfully inverted with the use of more advanced logic programming control strategies. Abstract interpretation, dynamic control mechanisms such as coroutining and intelligent backtracking, and search heuristics are very useful in this regard.

One promising avenue being investigated is the application of constraint logic programming (CLP) [vH90] towards imperative program inversion. Constraints are inequalities over domains such as arithmetic. This paper's *asgn_eq* relation is a rudimentary equality constraint. CLP permits arithmetic reasoning on constraints — reasoning that is not at all incorporated into the basic logic programming paradigm. In the program

```
$b$ := 0; $c$ := 0;
$if$ ( $a \geq 0$ ) $b$ := 1;
$if$ ( $a \leq 0$ ) $c$ := 1;
```

the tests in the *if* statements are regarded as constraints in a CLP system. Given a final state with $b = c = 1$, both constraints $a \geq 0$ and $a \leq 0$ will be valid, and the constraint solver deduces that $a = 0$.

Inversion computability is not guaranteed, even if sophisticated inference systems are used. Some inversions are unknown, while others are undecidable (Halting Problem). The complexity of mathematical primitives also influences computability. For example, higher-order polynomials are more difficult to invert than the simple arithmetic used in this paper. These realities shouldn't discourage the analysis of tractable inversions. One reasonable strategy to consider is the supplementing of inverted predicates with problem-specific information [TJ85]. Invariant relations can be added to program predicates to prevent inference down fruitless, nonterminating directions. As theorem proving and logic programming advance, the need for programmer intervention will diminish.

In conclusion, the application of logic programming technology towards the analysis of imperative computations is worth further investigation. Other work in this direction includes consequence verification of imperative control [CvE81] and partial evaluation transformations [Ros89].

Acknowledgements: Thanks to Harvey Abramson for his supervision of my MSc thesis, of which this work is a direct extension; Rajiv Trehan for his advice on CLP languages; Jamie Andrews, Rob Scott and Paul Wilk for their input; and anonymous referees for their advice. 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.

[BH91] D.R. Brough and C.J. Hogger. Grammar-related transformations of logic programs. *New Generation Computing*, 9:115–134, 1991.

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

[Cla79] K.L. Clark. Predicate Logic as a Computational Formalism. Technical Report 79/59, Imperial College, December 1979.

[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.

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

[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.

[GvdS90] D. Gries and J.L.A. van de Snepscheut. Inorder traversal of a binary tree and its inversion. In E.W. Dijkstra, editor, *Formal Development of Programs and Proofs*, pages 37–42. Addison Wesley, 1990.

[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.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice–Hall, 1985.

[Hog90] C.J. Hogger. *Essentials of Logic Programming.* Oxford University Press, 1990.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 1979.

[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.

[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.

[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.

[SA91] V. Sperschneider and G. Antoniou. *Logic: A Foundation for Computer Science.* Addison Wesley, 1991.

[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.

[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.

[vdS93] J.L.A. van de Snepscheut. *What Computing is All About.* Springer-Verlag, New York, 1993.

[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.