Verification of Tail-Recursive Logic Programs and Subgoal Induction

David A. Rosenblueth   Carlos Velarde V.

Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas
Universidad Nacional Autónoma de México
Apdo. 20-726
01000 México D.F.
{drosenbl, velarde}@servidor.unam.mx

Abstract

Kowalski has recently drawn attention to a gap existing between the procedural and declarative styles of logic programming. Such a gap results from the procedural style emphasizing the efficiency of a program, while the declarative style emphasizing the correctness of a program. In trying to bridge this gap, Kowalski has advocated, for some applications, first writing a procedural version of a logic program (as is often done), but then reading the program declaratively for verification. If we follow Kowalski's suggestion, we are likely to have to verify tail-recursive logic programs, because such programs are efficiently executed by Prolog.

We observe that the verification of tail-recursive logic programs coincides with an existing verification method for state-oriented (imperative) programs known as "subgoal induction," which is in a sense the converse of Floyd's inductive-assertions method. As a consequence, subgoal induction will play an important role in the declarative reading of efficiently executable logic programs. We establish our result by formalizing with unfold/fold operations a compilation of tail-recursive logic programs into a nondeterministic generalization of flowcharts. Because such a generalization includes subroutines, the connection with subgoal induction extends to clauses in which the last call is not recursive, provided that we give temporal precedence to the subgoals of each clause.

1 Introduction

Verification of tail-recursive logic programs. A logic programmer can write statements that are true about the relations of interest, concentrating only on a *declarative* description of such relations. Often, however, the execution of a program written in this manner is inefficient. Hence, logic programmers frequently take into account the *procedural* knowledge about a problem. When a logic program represents this kind of knowledge, the programmer gives temporal precedence to the subgoals of each clause. "Tail-recursive" definitions, in which a recursive procedure call occurs last in the definition of a predicate, are particularly important, because of being efficiently executable by Prolog. Kowalski has observed [12] that the declarative aspect of a logic program is useful even for a programmer who thinks procedurally, since programs so written can be *verified* by reading them declaratively. The purpose of this paper is to follow Kowalski's suggestion by studying the verification of tail-recursive logic programs.

Declarative and procedural knowledge. Logic programming is a radically new approach to program development. As remarked by Kowalski [12], logic programming combines declarative and procedural knowledge in the same representation. To illustrate the relationship between both these kinds of knowledge and their representation in logic programming, Kowalski uses in [12] the sorting problem. A top-level, declarative definition of this problem would relate two lists as one list being an ordered permutation of the other list:

$$\mathit{sort}(X, Y) \leftarrow \mathit{permutation}(X, Y),\ \ \mathit{ordered}(Y)$$

The Prolog execution of this program, however, may be extremely inefficient.

As a result of inefficiencies arising from programs such as this one, Prolog programmers often write programs such that, when given to Prolog, efficient algorithms are executed. In the sorting problem, for example, it is possible to make Prolog execute the quicksort algorithm with a program such as:

$$\begin{aligned} &\mathit{quicksort}([\,],[\,]) \leftarrow && (1)\\ &\mathit{quicksort}([X|Xs], Z) \leftarrow \mathit{part}(X, Xs, \mathit{Small}, \mathit{Big}), &&\\ &\qquad\qquad \mathit{quicksort}(\mathit{Small}, S), &&\\ &\qquad\qquad \mathit{quicksort}(\mathit{Big}, B), &&\\ &\qquad\qquad \mathit{append}(S, [X|B], Z) && (2) \end{aligned}$$

If read as a procedure, clause (2), for instance, would be interpreted as: To sort a nonempty list of values, first partition the tail of that list with respect to its head X, into a list having the values smaller than or equal to X and a list having the values bigger than X, then sort these two lists, and concatenate the sorted lists along with X.

Even more efficiency can be obtained with "difference lists" [21] as follows:

$$qs([\,], Zs, Zs) \leftarrow \tag{3}$$

$$\begin{aligned} qs([X|Xs], Ys, Zs) \leftarrow\ & part(X, Xs, Small, Big), \\ & qs(Small, Ys, [X|Ws]), \\ & qs(Big, Ws, Zs) \end{aligned} \tag{4}$$

not only avoiding the use of *append* but also making the program tail-recursive.

Kowalski argues that the logical basis of this program is useful if we can prove that the specification is a model of the program.

Flowchart verification. Applying tail-recursion optimization to a logic program makes such a program be executed as a flowchart. If this translation process is correct, then the logic program must have the same meaning as the corresponding flowchart. This indicates the possibility of using verification methods for flowcharts in the verification of tail-recursive logic programs.

Perhaps the best known method for verifying flowcharts is Floyd's [6]. However, there are other methods, such as "subgoal induction" [16]. Thus, two questions arise. First, which flowchart-verification method are we going to use? And second, given a tail-recursive logic program, which flowchart are we going to verify?

Clark and van Emden's work. Our starting point in trying to answer these questions is Clark and van Emden's work [2] relating flowchart verification with logic programs. These researchers exhibited a connection between Floyd's inductive-assertions verification method [6] and clauses of the form:

$$q(X, Z) \leftarrow p(X, Y),\ c(Y, Z) \tag{5}$$

where p and q are associated with points of the flowchart and c is in turn associated with a primitive command going from p to q. In addition, these authors found a similar connection between Morris and Wegbreit's subgoal-induction verification method [16] and clauses of the form:

$$p(X, Z) \leftarrow c(X, Y),\ q(Y, Z) \tag{6}$$

Observe that if p and q are associated with the same point of the flowchart, then (6) becomes a tail-recursive clause. This suggests considering subgoal induction, rather than Floyd's method, for verifying tail-recursive logic programs. We must, however, extend the connection between logic programs and subgoal induction to tail-recursive clauses which may not be exactly of the form (6).

Arbitrary tail-recursive clauses. First, we will generalize (6) to the case in which c may not necessarily represent a primitive command, but may in turn be associated with a flowchart. This generalization amounts to a verification of a system of mutually recursive flowcharts. Such a verification is studied by Morris and Wegbreit in [16], and we will confirm that it can also be described with clauses similar to (6).

Next, we will treat tail-recursive clauses having terms other than variables. We will give a transformation taking a tail-recursive definite clause and producing clauses of the form (6), having variables as arguments. The central idea of this transformation is the addition of a list that acts like a stack, and stores values in a similar way as procedures are compiled in state-oriented languages, such as Pascal. This transformation can be viewed as a logical reconstruction of a compilation process of a tail-recursive clause.

2 Verification of tail-recursive logic programs

We devote this section to some general ideas about tail recursion and program verification. First we define "tail-recursive logic programs," and then we introduce both logic program and flowchart verification.

2.1 Tail-recursive logic programs

Tail-recursion optimization. An important contribution to the execution of programs was the observation that just as the last call of a procedure can be replaced by a *goto* [7, 9], a *goto* can be eliminated by calling a procedure which calls other procedures as its last action [10]. "Tail-recursion optimization" is a technique for executing the last call of a procedure as a *goto* if it is recursive.[1]

Prolog's tail-recursion optimization. Tail-recursion optimization is common in Prolog systems. However, this technique is in a sense incompatible with Prolog's backtracking.

Consider the following logic program for computing the quotient and the remainder resulting from dividing an integer by possibly another one:

$$qr(U_0, V_0, K_1, U_1) \leftarrow qr'(0, U_0, V_0, K_1, U_1) \tag{7}$$

$$qr'(K_0, U_0, V_0, K_0, U_0) \leftarrow U_0 < V_0 \tag{8}$$

$$\begin{aligned} qr'(K_0, U_0, V_0, K_2, U_2) \leftarrow {} & U_0 \geq V_0,\ U_1 \ \mathit{is}\ U_0 - V_0,\ K_1 \ \mathit{is}\ K_0 + 1, \\ & qr'(K_1, U_1, V_0, K_2, U_2) \end{aligned} \tag{9}$$

[1] Sometimes, this tactic can be used in the last call of a procedure, even if it is not recursive. Hence, it is also known as "last-call optimization."

A goal would be, for instance:

$$\leftarrow qr(7, 2, K, U)$$

If this program is compiled using tail-recursion optimization, it will be executed in essentially the same way as the flowchart in Figure 1.

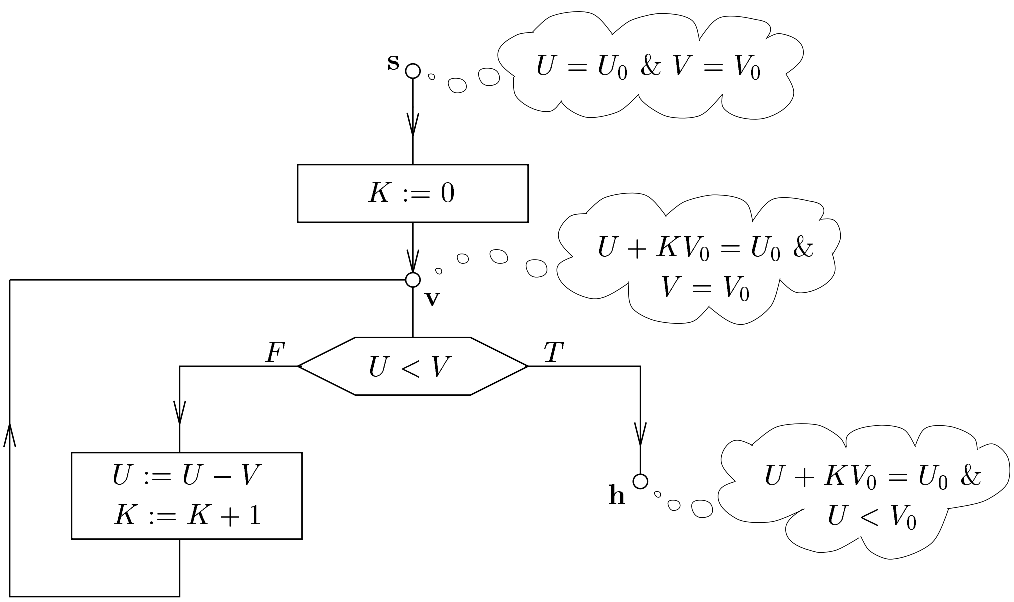


Figure 1: *A flowchart for computing quotient and remainder.*

Suppose, however, that any of the first three subgoals in clause (9) succeeds nondeterministically (i.e. leaving choice points). Then the last subgoal cannot be treated as a *goto*, because the information needed to reconstruct Prolog's state in case of backtracking would be lost. (We refer the reader to [15, 21] for a detailed explanation of this phenomenon.) A condition for the application of tail-recursion optimization in a clause is that all subgoals which are not the last one in that clause succeed deterministically.

Similarly, if clause (9) is written before clause (8), tail-recursion optimization cannot be applied without destroying a pointer to clause (9) necessary for execution in case of backtracking. (See [15, 21].) Thus, such an optimization is typically only applied to the last subgoal of the last clause in the definition of a predicate.

Tail-recursive logic programs. In this paper, we will not be concerned with whether or not these conditions are met by a particular program. The satisfaction of such (and other, similar) conditions in fact depend on the Prolog implementation, and here we are interested in connections between logic programs and flowcharts from a *verification* point of view. Thus, rather than dealing with tail-recursive Prolog programs, we prefer to use the term *tail-recursive logic program* to refer to a logic program having a clause in which the last subgoal has the same predicate symbol as the head. We will see (Section 6) that our verification techniques are applicable to tail-recursive logic programs even if they correspond to Prolog programs which do not satisfy the above conditions.

In addition, we will remark that the last call need not be recursive to apply our verification results. For didactical purposes, we think that it is advantageous to consider only tail-recursive logic programs for now, but later (Section 6) we will remove the recursivity condition.

2.2 Verification

Logic program verification. By "verifying" a logic program, we will understand proving that the specification is a model of the program (providing also a meaning of the auxiliary predicates) [11]. We can do so, by first assuming that the body of each clause holds with respect to the specification, and then proving that the head of the same clause also holds with respect to the specification. The fact that the quotient program can be compiled into a flowchart suggests considering existing verification methods for flowcharts, such as Floyd's [6].

Inductive-assertions method. In Floyd's method for proving partial correctness of a flowchart, an assertion is attached at each point of the flowchart. "Verifying" the flowchart consists in proving that each assertion holds if the assertion at the preceding point holds, except for the one associated with the start point. Given a computation, Floyd proves by induction on the length of the computation that if the initial assertion holds, then the assertions at all points of the computation also hold. Computations of verified flowcharts may not terminate; hence the qualification "partial." Termination is proved separately, by showing that an entity cannot decrease indefinitely in a computation.

Figure 1 appears with assertions allowing a verification of the flowchart with Floyd's method. Take for instance the assertion of point **v**. Assuming that this assertion holds, if we go around the loop once, we get:

$$(U - V) + (K + 1)V_0 = U_0 \ \& \ V = V_0$$

equivalently:

$$U - V + KV_0 + V_0 = U_0 \ \& \ V = V_0$$

which becomes the same assertion.

At first sight, it might seem that this assertion would help to verify the clauses (8) and (9). However, such clauses are not true of this assertion. To see this, observe that

Floyd's assertion refers to a predicate different from $qr'(K_1, U_1, V_0, K_2, U_2)$. Floyd's assertion relates the *initial* values of the variables (i.e. those of V_0 and U_0) with the values that change during the computation (i.e. those of unsubscripted U and K). By contrast, the $qr'(K_1, U_1, V_0, K_2, U_2)$ predicate relates the values of the variables that change during the computation (the first two argument places) with the *final* values of K and U (i.e. K_2 and U_2), together with the value of V_0.

Logical descriptions of flowcharts. Clark and van Emden [2] have given two alternative descriptions of flowcharts with logic programs. One of these descriptions defines "forward" computations by relating the initial values of the variables with those changing during the computation, and is essentially the same as the set of Floyd's verification conditions. The other one describes "backward" computations by relating the values of the variables which change as the computation proceeds with their final values, and coincides with the conditions to be proved by Morris and Wegbreit's "subgoal induction" verification method. This suggests connections between tail-recursive logic programs and subgoal induction.

3 An extension of Clark and van Emden's descriptions of flowcharts in logic

In this section, we review Clark and van Emden's logical descriptions of flowcharts [2] and extend such descriptions to recursive systems of flowcharts. As a convenient step for this generalization, we present a reformulation of some of the concepts and results given in [2], with certain changes in terminology and presentation. In [2], flowcharts are represented as "flowgraphs," a generalization allowing tests and actions of a flowchart to be treated uniformly as binary relations. Here, for the same purpose we use "active graphs," a slight generalization of the flowgraph concept. (This term was coined by van Emden in [23].) Also, we put emphasis on fixpoint semantics and we limit ourselves to logic program verification in the sense of Kowalski [11] (i.e. proving that the specification is a model of the program). Although we are mainly interested in the "backward" description of recursive systems of flowcharts, we also present the "forward" description of non-recursive flowcharts, for the sake of comparison and because it is more intuitive.

3.1 Active graphs

Active graphs. Informally, an active graph is a directed graph with its arcs labeled by binary relations over a given set of "states," and having two distinguished nodes sometimes called "input" and "output" nodes, respectively. The minor difference with respect to the concept of flowgraph presented in [2] is that for an active graph we do not impose the condition that the input node has no incoming arc nor the condition that the output node has no outgoing arc. Thus, flowgraphs are active graphs, and both can be used to model conventional flowcharts, as explained in [2] for flowgraphs. An active graph also can be viewed as a nondeterministic Ianov scheme under a general interpretation [5, pages 41–43], with only one final node, and the formalization presented next runs along these lines. We define an *active graph* as composed of:

1. a finite automaton $\langle \Sigma, \mathcal{N}, \mathbf{s}, \mathbf{h}, \mathcal{A} \rangle$, where Σ is an alphabet, $\mathcal{N}$ is a finite set of *nodes* in which we distinguish a *start node* $\mathbf{s}$ and a *halt node* $\mathbf{h}$, and $\mathcal{A} \subseteq \mathcal{N} \times \Sigma \times \mathcal{N}$ is a set of *arcs*; and

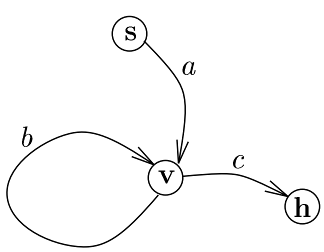
2. an interpretation of the elements of Σ as binary relations on a set D, i.e. a function $\mathcal{I} : \Sigma \to 2^{D \times D}$.

We denote an active graph as a 7-tuple $\langle \Sigma, \mathcal{N}, \mathbf{s}, \mathbf{h}, \mathcal{A}, \mathcal{I}, D \rangle$. The elements of D will be called *states*. The elements of Σ will be called *command symbols*, and the binary relations associated with them by $\mathcal{I}$ will be called *commands*.

Let us see how a flowchart can be modeled by an active graph. For example, following [2] but using a slightly different notation, to the flowchart of Figure 1 corresponds the active graph:

$$\langle \Sigma, \mathcal{N}, \mathbf{s}, \mathbf{h}, \mathcal{A}, \mathcal{I}, D \rangle = \langle \{a, b, c\}, \{\mathbf{s}, \mathbf{v}, \mathbf{h}\}, \mathbf{s}, \mathbf{h}, \{(\mathbf{s}, a, \mathbf{v}), (\mathbf{v}, b, \mathbf{v}), (\mathbf{v}, c, \mathbf{h})\}, \mathcal{I}, D \rangle$$

illustrated in Figure 2, with set of states D and interpretation $\mathcal{I}$ as explained next.



$$\begin{aligned}
a &\mapsto \{(\langle K, U, V\rangle, \langle 0, U, V\rangle) \mid K, U, V \in \mathbb{Z}\} \\
b &\mapsto \{(\langle K, U, V\rangle, \langle K, U, V\rangle) \mid U \geq V;\ K, U, V \in \mathbb{Z}\} \\
&\qquad \{(\langle K, U, V\rangle, \langle K+1, U-V, V\rangle) \mid K, U, V \in \mathbb{Z}\} \\
c &\mapsto \{(\langle K, U, V\rangle, \langle K, U, V\rangle) \mid U < V;\ K, U, V \in \mathbb{Z}\}
\end{aligned}$$

Figure 2: *Active graph for the flowchart of Figure 1.*

Each state $\sigma \in D$ is a function $\sigma : \{K, U, V\} \to \mathbb{Z}$ that maps each variable symbol occurring in the flowchart into an integer value. Because the set of variables is finite, it can be represented as a tuple, so that a state can be represented concisely as a tuple of the corresponding values. So, if in our example we take $\langle K, U, V \rangle$ as the tuple of variables, then a triple of integers as $\langle 0, 4, 3 \rangle$ represents the state $\{(K, 0), (U, 4), (V, 3)\}$. The interpretation $\mathcal{I}$ maps the symbols of the alphabet to commands as shown on the right-hand side of Figure 2. In the second formula the juxtaposition of binary relations denotes their composition in the order in which they appear. In the same way, from here on RS will denote the composition of R followed by S, for binary relations R and S. Also, as we consider only binary relations on D, the qualification frequently will be omitted.

Computations in active graphs. An active graph can be viewed as the definition of a binary relation —the "input-output" relation of the graph— in terms of the commands of the graph, combining these relations by means of the composition and union operations, in a way determined by the structure of the graph. For example, if we see Figure 2 as a domain-codomain diagram, as those used elsewhere in mathematics to represent compositions of functions (a particular kind of relations), we can view each node as the set D and each command symbol as a name for its associated command. The diagram then indicates the application of a, followed by zero or more applications of b (that is, followed by $b^* = I \cup b \cup b^2 \cup \ldots$, observing that an application of the identity relation I has the same effect as zero applications of b), followed by an application of c. So, in this simple case the input-output relation is ab^*c. As in this example, we will sometimes use a command symbol to represent its associated command, instead of the complete notation $\mathcal{I}(a)$.

There are several equivalent ways to formally define the input-output relation of an active graph. For example, following the intuitive idea just presented, this relation could be defined as the union of all the compositions each of which is determined by a path from the start node to the halt node of the graph. Although this definition is conceptually simple, from a computational point of view it is preferable to provide an inductive definition, or one that facilitates the inductive characterization of the input-output relation. This is the case of the next definition in terms of "computations," from which two inductive characterizations are readily obtained [2], which in turn lead to a corresponding pair of logic programs semantically related with the original flowchart.

Let us consider an active graph $G = \langle \Sigma, \mathcal{N}, \mathbf{s}, \mathbf{h}, \mathcal{A}, \mathcal{I}, D \rangle$. A *computation* is a nonempty and possibly infinite sequence of (node, state)-pairs

$$\ldots, (n_i, \sigma_i), \ldots \qquad i = \ldots, -2, -1, 0, 1, 2, \ldots$$

such that σ_i and σ_{i+1} are related by a command c_i labeling an arc from n_i to n_{i+1} in G, for all i in the sequence. When the sequence is finite, $(n_j, \sigma_j), \ldots, (n_k, \sigma_k)$, we say that it is a *computation from n_j to n_k*. The relation whose elements are the pairs (σ_j, σ_k) for which there exists such a computation is called the *computed relation from n_j to n_k*, and when these nodes are, respectively, the start and halt nodes, it is also called the *input-output relation* of the active graph.

Forward relations and forward description. A *forward computation* is a computation from the start node to any node n. As in [2], the set of forward computations can be inductively defined as the smallest set $\mathcal{F}$ such that:

1. $\mathcal{F}$ includes all the computations $(\mathbf{s}, \sigma)$ and
2. if $\mathcal{F}$ includes the computation $(\mathbf{s}, \sigma), \ldots, (m, \mu)$, then $\mathcal{F}$ also includes all the computations $(\mathbf{s}, \sigma), \ldots, (m, \mu), (n, \nu)$ such that $(\mu, \nu) \in a$ for some arc (m, a, n).

The computed relation from $\mathbf{s}$ to n will be called *forward relation* to n and will be denoted f_n. Observe that $f_{\mathbf{h}}$ is the input-output relation of the active graph. It follows directly from the definitions that for the forward relations we have:

$$\begin{aligned} &f_{\mathrm{s}} \supseteq I \\ &f_n \supseteq f_m a \qquad (m, a, n) \in \mathcal{A} \end{aligned} \tag{10}$$

With a slight abuse of notation, reading each symbol f_n as a variable over $2^{D \times D}$, we will call (10) the *system of forward inclusions* for the active graph. So, in its original reading, (10) asserts that the forward relations constitute a solution to the system of forward inclusions. It is convenient to represent this solution as a tuple $(f_{\mathrm{s}}, \ldots, f_n, \ldots, f_{\mathrm{h}})$ with one component per node. Let us prove that this solution is (componentwise) included in any other solution to (10).

Theorem 3.1 *The forward relations constitute the least solution to the system of forward inclusions.*

Proof Let $(f_{\mathrm{s}}, \ldots, f_n, \ldots, f_{\mathrm{h}})$ be the solution to (10) constituted by the forward relations and let $(\varphi_{\mathrm{s}}, \ldots, \varphi_n, \ldots, \varphi_{\mathrm{h}})$ be any other solution to that system. We must prove that $f_n \subseteq \varphi_n$ for each $n \in \mathcal{N}$. From the definitions, we have that $(\chi, \zeta) \in f_n$ iff there exists a forward computation $(\mathbf{s}, \chi), \ldots, (n, \zeta)$. So, we must prove that for any such computation it follows that $(\chi, \zeta) \in \varphi_n$. This will be done by an induction on the length of the computation. For the base case, let $(\mathbf{s}, \chi)$ be any forward computation of length one; then $(\chi, \chi) \in \varphi_{\mathrm{s}}$ because, by hypothesis, φ_{s} satisfies the first inclusion in (10). For the induction step, let $(\mathbf{s}, \chi), \ldots, (m, \gamma), (n, \zeta)$ be any forward computation of length greater than one. As $(\mathbf{s}, \chi), \ldots, (m, \gamma)$ is also a forward computation, by the induction hypothesis we have that $(\chi, \gamma) \in \varphi_m$. From this inclusion and the fact that $(\gamma, \zeta) \in a$ for some arc (m, a, n), it follows that $(\chi, \zeta) \in \varphi_m a$. Then $(\chi, \zeta) \in \varphi_n$ because $\varphi_n \supseteq \varphi_m a$ holds, as this inclusion is an instance of one of the forward inclusions (10), and we assumed that φ_m and φ_n satisfy (10). $\square$

In predicate notation, as used in logic programming, the system (10) can be written as the following set of implications, known as the *forward description* [2] of the active graph in question:

$$\begin{aligned} &f_{\mathrm{s}}(X, X) \leftarrow \\ &f_n(X, Z) \leftarrow f_m(X, Y),\ a(Y, Z) \qquad (m, a, n) \in \mathcal{A} \end{aligned} \tag{11}$$

Observe that the first formula is simpler than but equivalent to $f_{\mathrm{s}}(X,Y) \leftarrow I(X,Y)$, the direct translation of the first forward inclusion, as a consequence of the I definition.

As an example, for the active graph of Figure 2 the system of forward inclusions and the forward description are shown next to the left and to the right, respectively:

$$\begin{array}{ll} f_{\mathrm{s}} \supseteq I & f_{\mathrm{s}}(X,X) \leftarrow \\ f_{\mathrm{v}} \supseteq f_{\mathrm{s}}\ a & f_{\mathrm{v}}(X,Z) \leftarrow f_{\mathrm{s}}(X,Y),\ a(Y,Z) \\ f_{\mathrm{v}} \supseteq f_{\mathrm{v}}\ b & f_{\mathrm{v}}(X,Z) \leftarrow f_{\mathrm{v}}(X,Y),\ b(Y,Z) \\ f_{\mathrm{h}} \supseteq f_{\mathrm{v}}\ c & f_{\mathrm{h}}(X,Z) \leftarrow f_{\mathrm{v}}(X,Y),\ c(Y,Z) \end{array} \tag{12}$$

The forward description (11) is a set of formulas of a first-order language. Any solution $(\varphi_{\mathrm{s}}, \ldots, \varphi_n, \ldots, \varphi_{\mathrm{h}})$ to (10), together with the commands, constitutes a model of (11). Because the meaning of the command symbols is fixed in advance, we will call *model w.r.t. the commands* any model of this kind. Using these terms, Theorem 3.1 can be reformulated in the following way:

Theorem 3.2 *The forward relations constitute the least model w.r.t. the commands of the forward description.*

Now let us say some words about computability. Following Apt [1, pages 523–528], let L be a language with finitely many but at least one constant symbol, finitely many but at least one function symbol, infinitely many predicate symbols in every arity, and including the language associated with the logic program (11). In an Herbrand interpretation for L, the set of states D can be identified with an appropriate set of tuples of ground terms of L (each tuple of the length required to represent a state.) Given D in this way, Theorem 3.2 says that the forward relations constitute the least Herbrand model (w.r.t. the commands) of the forward description.

Suppose that the commands are recursively enumerable. Then, by Corollary 4.5 in [1] we have that for each command there is a corresponding program which computes the command. If we now put together (11) and the programs corresponding to the commands, we obtain a program whose least Herbrand model is formed by the forward relations together with the commands. This program computes the forward relations and, by the same corollary, these relations are recursively enumerable.

Forward description and Floyd's method. We say that an active graph is *(partially) correct* w.r.t. a relation Ψ (the *specification*) if the input-output relation f_{h} of the graph is included in Ψ. (A more familiar definition of correctness could be: an active graph is correct w.r.t. a unary predicate $\psi(x)$ and a binary predicate $\Psi'(x,z)$ —respectively the input and output specifications— if the implication $\psi(x) \wedge f_{\mathrm{h}}(x,z) \rightarrow \Psi'(x,z)$ holds [4, page 873]. But this formula is logically equivalent to $f_{\mathrm{h}}(x,z) \rightarrow (\psi(x) \rightarrow \Psi'(x,z))$, which is of the form $f_{\mathrm{h}}(x,z) \rightarrow \Psi(x,z)$, with $\Psi(x,z)$

defined as $\psi(x) \to \Psi'(x,z)$. So, this definition of correctness reduces to the former one.) A proof of $f_{\mathrm{h}} \subseteq \Psi$ is called a *verification* of the graph w.r.t. Ψ. One verification can be obtained by finding a solution $(\varphi_{\mathrm{s}}, \ldots, \varphi_{\mathrm{h}})$ to (10) such that $\varphi_{\mathrm{h}} \subseteq \Psi$. This is so because the least solution to (10) is included in any other solution, then in particular we have $f_{\mathrm{h}} \subseteq \varphi_{\mathrm{h}}$, which combined with the inclusion $\varphi_{\mathrm{h}} \subseteq \Psi$ leads to $f_{\mathrm{h}} \subseteq \Psi$. If the active graph models a flowchart, this proof method can be recognized as a relational variant of the Floyd's inductive-assertions verification method [4]. The φ's play the role of the inductive assertions and the implications (11) play the role of the verification conditions. So, we have a generalization of Floyd's method, to be applied to active-graph verification.

As we know, $(\varphi_{\mathrm{s}}, \ldots, \varphi_{\mathrm{h}})$ is a solution of the system of forward inclusions if and only if $(\varphi_{\mathrm{s}}, \ldots, \varphi_{\mathrm{h}})$ is a model (w.r.t. the commands) of the logic program (11). Then a verification of the logic program, which consists in a proof of the right-hand side of this biconditional [3], can be considered as an inductive-assertions verification of the active graph (w.r.t. φ_{h}, which is taken equal to Ψ).

Backward computations. Another logical description of an active graph can be given departing now from the *backward computations*, which are the computations from a node n to the halt node. The set $\mathcal{B}$ of backward computations can be inductively constructed in a similar way as was done for the set of forward computations. For any node n, let the computed relation from n to $\mathbf{h}$ be denoted by $\mathcal{b}_n$ and called *backward relation* from n. In particular, $\mathcal{b}_{\mathbf{s}}$ is the input-output relation of the graph. The *system of backward inclusions* is:

$$\begin{aligned} &\mathcal{b}_{\mathrm{h}} \supseteq I \\ &\mathcal{b}_m \supseteq a\ \mathcal{b}_n \qquad (m,a,n) \in \mathcal{A} \end{aligned} \tag{13}$$

The following result can be proved in essentially the same way as was done for Theorem 3.1:

Theorem 3.3 *The backward relations constitute the least solution to the system of backward inclusions.*

The *backward description* of the active graph is:

$$\begin{aligned} &\mathcal{b}_{\mathrm{h}}(Z,Z) \leftarrow \\ &\mathcal{b}_m(X,Z) \leftarrow a(X,Y),\ \mathcal{b}_n(Y,Z) \qquad (m,a,n) \in \mathcal{A} \end{aligned} \tag{14}$$

Reformulating Theorem 3.3 we have:

Theorem 3.4 *The backward relations constitute the least model w.r.t. the commands of the backward description.*

For the active graph of Figure 2 the system of backward inclusions is:

$$
\begin{aligned}
ƀ_{\mathrm{h}} &\supseteq I \\
ƀ_{\mathrm{v}} &\supseteq c\ ƀ_{\mathrm{h}} \\
ƀ_{\mathrm{v}} &\supseteq b\ ƀ_{\mathrm{v}} \\
ƀ_{\mathrm{s}} &\supseteq a\ ƀ_{\mathrm{v}}
\end{aligned}
\tag{15}
$$

and the backward description is:

$$
\begin{aligned}
&ƀ_{\mathrm{h}}(Z,Z) \leftarrow \\
&ƀ_{\mathrm{v}}(X,Z) \leftarrow c(X,Y),\ ƀ_{\mathrm{h}}(Y,Z) \\
&ƀ_{\mathrm{v}}(X,Z) \leftarrow b(X,Y),\ ƀ_{\mathrm{v}}(Y,Z) \\
&ƀ_{\mathrm{s}}(X,Z) \leftarrow a(X,Y),\ ƀ_{\mathrm{v}}(Y,Z)
\end{aligned}
\tag{16}
$$

Backward description and the subgoal induction method. A different verification of an active graph with respect to Ψ can be obtained by finding a solution $(\beta_{\mathrm{s}}, \ldots, \beta_{\mathrm{h}})$ to (13) such that $\beta_{\mathrm{s}} \subseteq \Psi$. The reason is that the least solution to (13) is included in any other solution. In particular we have $ƀ_{\mathrm{s}} \subseteq \beta_{\mathrm{s}}$, which combined with the inclusion $\beta_{\mathrm{s}} \subseteq \Psi$ leads to $ƀ_{\mathrm{s}} \subseteq \Psi$. When the active graph corresponds to a flowchart, this proof method can be recognized as the subgoal induction method [4], so we have an extension of this method to active graph verification.

As in the case of the forward relations, proving that $(\beta_{\mathrm{s}}, \ldots, \beta_{\mathrm{h}})$ is a solution of (13) is the same as proving that $(\beta_{\mathrm{s}}, \ldots, \beta_{\mathrm{h}})$ is a model (w.r.t. the commands) of (14), and this is a verification of (14). That is, a verification of (14) is a subgoal-induction verification (w.r.t. β_{s}, which is taken as Ψ) of the original active graph.

Backward relations as a limit point. Using the theory of complete partial orders (cpo's), the least solution of systems like (10) and (13) can be expressed as a limit point of an appropriate operator. Let us see the case of the system of backward relations. We may suppose that each backward relation $ƀ_i$ ($i \in \mathcal{N}$) appears as left-hand side of at least one inclusion of the system of backward inclusions for the active graph (if it is not the case, say for the relation $ƀ_i$, then we add the superfluous inclusion $ƀ_i \supseteq \emptyset$). If a relation $ƀ_i$ appears in more than one inclusion as left-hand side, these inclusions can be replaced by one which has $ƀ_i$ to the left and the union of the right-hand sides of the replaced inclusions to the right. For example, the second and third inclusions in (15) become the single inclusion:

$$ƀ_{\mathrm{v}} \supseteq c\ ƀ_{\mathrm{h}} \cup b\ ƀ_{\mathrm{v}}$$

This way, the following system of $|\mathcal{N}|$ inclusions is obtained:

$$ƀ_i \supseteq B_i(ƀ_{\mathrm{s}}, \ldots, ƀ_{\mathrm{h}}) \qquad i \in \mathcal{N} \tag{17}$$

where $B_i(\bar{b}_{\mathrm{s}}, \ldots, \bar{b}_{\mathrm{h}})$ denotes a relational expression in terms of compositions and unions of the $|\mathcal{N}|$ backward relations and the given commands of the active graph. It is easy to show that this and the original system have the same set of solutions; thus we say that these systems are equivalent. We will also refer to (17) as a *system of backward inclusions*.

Each right-hand side of the inclusions in (17) can be seen as defining value of a corresponding function $B_i : (2^{D\times D})^{|\mathcal{N}|} \to 2^{D\times D}$. Let us keep in mind that any Cartesian product set $(2^{D\times D})^k$ is a cpo, with the inclusion of sets between corresponding components as the order relation. It turns out that each function B_i is continuous, because it is defined as a combination of the continuous operators of union and composition of binary relations. Hence the continuity of the function $B : (2^{D\times D})^{|\mathcal{N}|} \to (2^{D\times D})^{|\mathcal{N}|}$, with $B = (B_{\mathrm{s}}, \ldots, B_{\mathrm{h}})$, is guaranteed. But then the function B has a least fixpoint $\mathit{lfp}(B)$ which is equal to the limit $\bigcup_{k\in\mathbb{N}} B^k(\emptyset, \ldots, \emptyset)$. Now, the Knaster/Tarski theorem [1, 22] states that this fixpoint is also the least pre-fixpoint of B, which means that $\mathit{lfp}(B)$ is the least solution of the system (17).

As an example, for the graph of Figure 2 we have $B(\bar{b}_{\mathrm{s}}, \bar{b}_{\mathrm{v}}, \bar{b}_{\mathrm{h}}) = (a\bar{b}_{\mathrm{v}}, c\bar{b}_{\mathrm{h}} \cup b\bar{b}_{\mathrm{v}}, I)$, and it can easily be proved that $\mathit{lfp}(B) = (ab^*c, b^*c, I)$.

3.2 Systems of active graphs

Now we proceed to generalize the active-graph definition to one for a "recursive system" of active graphs. Informally, such a system consists of a finite set of active graphs, but now to each graph a corresponding name is assigned and these names also can be used as arc labels. We define a *recursive system of active graphs* as composed of:

1. an alphabet $\Gamma = \{G_k \mid 1 \le k \le N\}$ of *graph names*;

2. for each graph name G_k a corresponding finite automaton $\langle \Sigma\cup\Gamma, \mathcal{N}_k, \mathbf{s}_k, \mathbf{h}_k, \mathcal{A}_k \rangle$, where Σ is an alphabet disjoint with Γ, $\mathcal{N}_k$ is a finite set of *nodes* in which we distinguish an start node $\mathbf{s}_k$ and a halt node $\mathbf{h}_k$, $\mathcal{N}_k$ is disjoint with $\mathcal{N}_j$ for each $j \neq k$, and $\mathcal{A}_k \subseteq \mathcal{N}_k \times (\Sigma\cup\Gamma) \times \mathcal{N}_k$ is a set of *arcs*; and

3. an interpretation of the elements of Σ as binary relations on a set D, i.e. a function $\mathcal{I} : \Sigma \to 2^{D\times D}$.

Each name of a graph will be associated with a binary relation on D, which will be called the "input-output" relation of the corresponding graph. These relations could in principle be defined in terms of computations, in a way similar to the one followed in the case of an active graph. But if such a way were chosen, first the concept of computation would have to be generalized in order to take care of recursive applications of relations, which can occur along the sequence of relation applications leading an initial state into a final state. An approach of this kind is followed in [5, pages 51–54], in defining the relation computed by a nondeterministic recursive system of Ianov schemes.

Instead of this approach, for our purposes here it is better the following shortcut, in which a statement like Theorem 3.3 is taken as definition of the backward relations associated with the system of graphs.

Each graph G_k of a system correspondingly determines the following set of inclusions similar to (13):

$$\begin{aligned} &\mathfrak{b}_{\mathrm{h}_k} \supseteq I \\ &\mathfrak{b}_m \supseteq a\ \mathfrak{b}_n \qquad (m,a,n)\in\mathcal{A}_k \end{aligned} \tag{18}$$

with each $\mathfrak{b}_n$ considered as a variable over the set of binary relations on D, and with every occurrence of a variable associated with an input node, $\mathfrak{b}_{\mathrm{s}_j}$, replaced by the name of the corresponding graph G_j, which in these inclusions is used as a variable of the same type as that of the replaced variable. The union of all the sets of inclusions in such a way associated with the different graphs will be called the system of *backward inclusions* for the system of graphs.

For example, consider the system of active graphs presented in Figure 3. In each graph, the leftmost node is the input node and the rightmost node is the output node. The system of backward inclusions for this system of graphs is the following, where the inclusions shown on the left correspond to the first graph and the inclusions shown on the right correspond to the second graph:

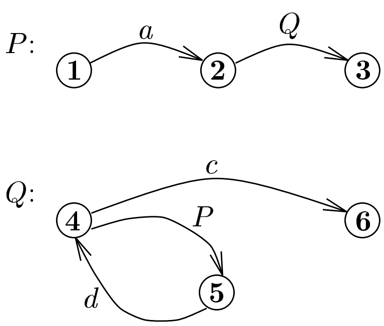


Figure 3: *A recursive system of active graphs.*

$$\begin{aligned} &\mathfrak{b}_3 \supseteq I &\qquad& \mathfrak{b}_6 \supseteq I \\ &\mathfrak{b}_2 \supseteq Q\ \mathfrak{b}_3 && Q \supseteq c\ \mathfrak{b}_6 \\ &P \supseteq a\ \mathfrak{b}_2 && Q \supseteq P\ \mathfrak{b}_5 \\ &&& \mathfrak{b}_5 \supseteq d\ Q \end{aligned} \tag{19}$$

This system can be rewritten in the next equivalent form, by following a similar algebraic manipulation to that used for obtaining (17) from (13):

$$\begin{aligned} &\mathfrak{b}_3 \supseteq I &\qquad& \mathfrak{b}_6 \supseteq I \\ &\mathfrak{b}_2 \supseteq Q\ \mathfrak{b}_3 && Q \supseteq c\ \mathfrak{b}_6 \cup P\ \mathfrak{b}_5 \\ &P \supseteq a\ \mathfrak{b}_2 && \mathfrak{b}_5 \supseteq d\ Q \end{aligned} \tag{20}$$

In general, for any system of graphs the system of backward inclusions written in a form like (20) has $|\mathcal{N}_1 \cup \cdots \cup \mathcal{N}_N|$ inclusions, with each variable appearing as left-hand side of a corresponding inclusion. The argument given to characterize the least solution to (17) as a least fixpoint also is valid here. We take the relations that constitute this least solution as the *backward relations* for the system of graphs. Each of these relations which names a graph (P and Q in the example) is called the *input-output relation* of such a graph.

As in the case of an active graph, writing (18) in predicate notation we obtain the backward description corresponding to the active graph G_k of a system:

$$
\begin{array}{ll}
ϐ_{\mathrm{h}_k}(Z,Z) \leftarrow & \\
ϐ_m(X,Z) \leftarrow a(X,Y),\ ϐ_n(Y,Z) \qquad & (m,a,n) \in \mathcal{A}_k
\end{array}
\tag{21}
$$

As the *backward description* of a system of active graphs we take the union of the backward descriptions corresponding to the graphs of the system.

For the system in Figure 3 the backward description is:

$$
\begin{array}{ll}
 & ϐ_6(Z,Z) \leftarrow \\
ϐ_3(Z,Z) \leftarrow & q(X,Z) \leftarrow c(X,U),\ ϐ_6(U,Z) \\
ϐ_2(Y,Z) \leftarrow q(Y,U),\ ϐ_3(U,Z) & q(X,Z) \leftarrow p(X,U),\ ϐ_5(U,Z) \\
p(X,Z) \leftarrow a(X,Y),\ ϐ_2(Y,Z) & ϐ_5(U,Z) \leftarrow d(U,V),\ q(V,Z)
\end{array}
\tag{22}
$$

Suppose that we have a system of active graphs for which the set of states D is chosen as we did right after Theorem 3.2, but here including in L the language associated with the backward description of the given system. Then, by the definition of the backward relations, these relations constitute the least Herbrand model (w.r.t. the commands) of the backward description. If in addition the commands are computable, so are the backward relations and, in particular, the input-output relations corresponding to the graphs.

The subgoal induction method and tail-recursive chain programs. The presented verification methods for active graphs can be readily extended to cover systems of active graphs. Next, we treat the case of the subgoal induction method. A system of N active graphs is said to be *(partially) correct* w.r.t. a tuple of relations $\Psi = (\Psi_1, \ldots, \Psi_N)$ (the *specification*) if the tuple $(G_1, \ldots, G_N)$ of input-output relations for the system of graphs is (component-wise) included in Ψ. A proof of this inclusion is called a *verification* of the system of graphs w.r.t. Ψ. A verification by the subgoal induction method consists in finding a solution $(\beta_{\mathrm{s}_1}, \ldots, \beta_{\mathrm{h}_1}, \ldots, \beta_{\mathrm{s}_N}, \ldots, \beta_{\mathrm{h}_N})$ to the system of backward inclusions —composed of N sets of inclusions like (18)— such that the tuple $(\beta_{\mathrm{s}_1}, \ldots, \beta_{\mathrm{s}_N})$ is (component-wise) included in Ψ.

Also, as in the case of a single active graph, a verification of a backward description can be seen as a subgoal-induction verification of the corresponding system of active graphs. Now, by means of an example we will show that semantical relationships of the same nature also hold for tail-recursive "chain programs."

By *chain program* we understand a logic program having only clauses of the following general form:

$$
\begin{array}{ll}
 & p_0(X_0,X_n) \leftarrow p_1(X_0,X_1),\ p_2(X_1,X_2),\ \ldots,\ p_n(X_{n-1},X_n) \qquad (n \geq 1) \\
\text{or} & p_0(t,t') \leftarrow
\end{array}
$$

Note that all the logical descriptions so far presented are chain programs. Let us observe that the following chain program can be obtained from (22) by means of the application of "unfold" operations (i.e. resolution steps):

$$p(X,Z) \leftarrow a(X,Y),\ q(Y,Z) \qquad\qquad \begin{array}{l} q(X,Z) \leftarrow c(X,Z) \\ q(X,Z) \leftarrow p(X,U),\ d(U,V),\ q(V,Z) \end{array} \tag{23}$$

where the clauses shown on each side came from the clauses shown on the corresponding side in (22). This process can be reversed by applying the appropriate "fold" operations, so that (23) can also serve as a logical description of the original active graphs. Observe now that the last clause in (23) is tail-recursive. If we trace back the occurrence of q in the tail of this clause, we will see that it comes from the q in the last clause of (22), which in turn originates in the Q appearing in the last inclusion of (19) (where, by convenience, Q was used in place of $\textit{б}_4$). It is not difficult to show that any tail-recursive chain clause can be transformed as illustrated into a set of clauses of the form required in a backward description, with one of its clauses having as second subgoal the last subgoal of the tail-recursive clause.

Now, if we have a verification of (22), we can easily obtain a verification of (23). But we also can proceed in the other direction: if a verification of (23) is given, we can then construct a verification of (22) by assigning meanings to the $\textit{б}$'s in the following way. Assign to $\textit{б}_3$ and to $\textit{б}_6$ the identity relation, to $\textit{б}_2$ the same relation q has as meaning, and to $\textit{б}_5$ the composition of the relations d and q have respectively as meaning. Thus, we have illustrated that the verification of a tail-recursive chain program is in essence a subgoal-induction verification of certain system of active graphs.

4 Tail-recursive logic programs and backward descriptions

We have shown that the backward description of a system of active graphs is the set of subgoal-induction verification conditions for that system. This kind of description, however, consists of chain clauses. Now we will connect the meanings for the predicates in a *tail-recursive* logic program (in which the clauses may have not have chain form) with the meanings for the backward description of a system of active graphs.

4.1 "Folding" as a means of obtaining chain clauses

One way of obtaining a chain clause from a definite clause could be to fold certain binary predicates. The reason is that the fold operation replaces some subgoals by a single atom. So, if the new atom has variables as arguments, we will produce a clause having such an atom in its body. We will see, however, that such a folding operation may not always be sound.

Consider the recursive clause of the difference-list quicksort program of Section 1.

$$\begin{aligned} qs([X|Xs], Ys, Zs) \leftarrow\ & part(X, Xs, Small, Big), \\ & qs(Small, Ys, [X|Ws]), \\ & qs(Big, Ws, Zs) \end{aligned} \tag{24}$$

A first step towards obtaining chain clauses could be to partition the set of arguments of each predicate depending on whether they play the role of input (instantiated) or output (uninstantiated). By grouping all input arguments in one term and all the output arguments in another term, we can rewrite the clause using only binary predicates.

$$\begin{aligned} qs'([X|Xs], \langle Ys, Zs\rangle) \leftarrow\ & part'(\langle X, Xs\rangle, \langle Small, Big\rangle), \\ & qs'(Small, \langle Ys, [X|Ws]\rangle), \\ & qs'(Big, \langle Ws, Zs\rangle) \end{aligned} \tag{25}$$

Here, and in the rest of the paper we use angled brackets $\langle\ \rangle$ for terms grouping either input or output arguments.

As preparation for folding, we will "transfer" the input and output arguments of subgoals into equations (subgoals with the "=" predicate symbol). We can do so through the predicate substitutivity axioms of the standard-equality theory. For example, the predicate substitutivity axiom for $part'$ is:

$$part'(A, B) \leftarrow A = U,\ B = V,\ part'(U, V) \tag{26}$$

Resolving (25) with (26), we obtain:

$$\begin{aligned} qs'([X|Xs], \langle Ys, Zs\rangle) \leftarrow\ & \boxed{\langle X, Xs\rangle = U_0,\ \langle Small, Big\rangle = V_0,\ part'(U_0, V_0),} \\ & qs'(Small, \langle Ys, [X|Ws]\rangle), \\ & qs'(Big, \langle Ws, Zs\rangle) \end{aligned} \tag{27}$$

Similarly, resolving (27) with the predicate substitutivity axiom for qs', we get:

$$\begin{aligned} qs'([X|Xs], \langle Ys, Zs\rangle) \leftarrow\ & \langle X, Xs\rangle = U_0,\ \langle Small, Big\rangle = V_0,\ part'(U_0, V_0), \\ & \boxed{Small = U_1,\ \langle Ys, [X|Ws]\rangle = V_1,\ qs'(U_1, V_1),} \\ & qs'(Big, \langle Ws, Zs\rangle) \end{aligned}$$

Next, we use the symmetry axiom $X = Y \leftarrow Y = X$ and rewrite the subgoals in a more convenient order:

$$\begin{aligned} qs'([X|Xs], \langle Ys, Zs\rangle) \leftarrow\ & U_0 = \langle X, Xs\rangle,\ part'(U_0, V_0), \\ & \boxed{V_0 = \langle Small, Big\rangle,\ U_1 = Small,} \\ & qs'(U_1, V_1),\ V_1 = \langle Ys, [X|Ws]\rangle, \\ & qs'(Big, \langle Ws, Zs\rangle) \end{aligned} \tag{28}$$

Apparently, to eliminate the equations we could now introduce a predicate such as:

$$k(\langle S, B\rangle, S) \leftarrow \tag{29}$$

which has as completion [13, page 78]:

$$k(V_0, U_1) \leftrightarrow \exists S\, \exists B\, [V_0 = \langle S, B\rangle \ \&\ U_1 = S]$$

Folding this completion on (28) we arrive at:

$$\begin{aligned} qs'([X\,|\,Xs], \langle Ys, Zs\rangle) \leftarrow\ & U_0 = \langle X, Xs\rangle,\ \ part'(U_0, V_0), \\ & \boxed{k(V_0, U_1),} \\ & qs'(U_1, V_1),\ \ V_1 = \langle Ys, [X\,|\,Ws]\rangle, \\ & qs'(Big, \langle Ws, Zs\rangle) \end{aligned} \tag{30}$$

However, this is an incorrect inference. To see this we can resolve (29) with (30), and apply the reflexivity axiom $X = X \leftarrow$, to derive:

$$\begin{aligned} qs'([X\,|\,Xs], \langle Ys, Zs\rangle) \leftarrow\ & part'(\langle X, Xs\rangle, \langle S, \boxed{B}\rangle), \\ & qs'(S, \langle Ys, [X\,|\,Ws]\rangle), \\ & qs'(\boxed{Big}, \langle Ws, Zs\rangle) \end{aligned} \tag{31}$$

which is a generalization of (25): clause (31) has two different variables B and Big where clause (25) has two occurrences of the same variable. Note that the same difficulty does not arise with the variable $Small$, because in (28) such a variable only occurs in the equations which are eliminated upon folding the completion of $k(V_0, U_1)$. In fact, this is a sufficient condition for the soundness of folding, which can be proved using the result in [20, pages 57, 58].

We conclude that applying predicate substitutivity and then folding does achieve our purpose, but we should do so once we have obtained a clause of the form:

$$\hat{p}_0(s_0, s'_n) \leftarrow \hat{p}_1(s'_0, s_1),\ \hat{p}_2(s'_1, s_2),\ \ldots,\ \hat{p}_n(s'_{n-1}, s_n) \tag{32}$$

where:

$$(\mathrm{var}(s_i) \cup \mathrm{var}(s'_i)) \cap (\mathrm{var}(s_j) \cup \mathrm{var}(s'_j)) = \emptyset$$

for $i \neq j$. We will say that such a clause has *relay form*.

4.2 Converting to relay form before folding

A transformation following the idea outlined above was used in [18, 19] to obtain inference systems based on parsers. Such a transformation adds a term to the input and to the output of each subgoal, which behaves like a stack, and stores terms needed by subsequent subgoals.

This transformation, however, has two limitations for the objectives of the present work. First, predicates have fixed modes, where each argument place of a predicate is used either as input or as output. We would like, instead, to be able to convert into chain form predicates having multiple modes. (For example, we should be able to handle the $append(X, Y, Z)$ predicate regardless of whether it is used for appending or for splitting lists.)

Second, the chain clauses produced by the transformation of [18, 19] are not tail-recursive, even if the original clause is so. (A subgoal similar to $k(V_0, U_1)$ above is introduced after the last subgoal of the original clause.) Hence, the resulting chain clause corresponds to verification conditions of a flowchart which has, instead of a *goto*, a recursive call, followed by a call to the last $k(U, V)$.

To overcome the first deficiency (i.e. that predicates be well moded), we will assume an execution mechanism able to handle terms with variables. This assumption is reasonable since Prolog's execution mechanism is able to handle such terms. Now we will group all arguments of each predicate $p(X_1, \ldots, X_n)$ of the original clause in the first argument of a new predicate $\hat{p}(U, V)$. The second argument of each new predicate will hold information "needed" by the subsequent subgoals.

To overcome the second deficiency (i.e. that the resulting chain clause is not tail-recursive), we must avoid the introduction of the last $k(V, U)$ subgoal. Such a subgoal is avoided when the terms s_n and s'_n in clause (32) are the same variable.

As an example, consider again clause (24) of the quicksort example. Let us introduce the following two predicates:

$$\widehat{qs}(\langle St, Xs, Ys, Zs\rangle, St') \leftrightarrow St = St' \;\&\; qs(Xs, Ys, Zs)$$
$$\widehat{part}(\langle St, X, Xs, Small, Big\rangle, St') \leftrightarrow St = St' \;\&\; part(X, Xs, Small, Big)$$

We will show how to convert clause (24) into relay form. For the sake of clarity, we will eliminate the equations of the derived clauses by applying reflexivity. However, we refer the reader to the appendix, where we first outline a soundness proof of the whole transformation and then exemplify this proof with clause (24).

The predicates with symbols $\widehat{qs}$ and $\widehat{part}$ group all arguments of the corresponding predicates in the original clause into one term and add a stack. It remains to determine, however, which variables to store in this stack. Consider the head and the first subgoal of the clause to be transformed:

$$\begin{aligned}\underline{qs([\boldsymbol{X}_0|Xs_0], \boldsymbol{Ys}_0, \boldsymbol{Zs}_0) \leftarrow part(\boldsymbol{X}_0, Xs_0, \boldsymbol{Small}_0, \boldsymbol{Big}_0)},&\\ qs(\boldsymbol{Small}_0, \boldsymbol{Ys}_0, [\boldsymbol{X}_0|\, Ws_0]),&\\ qs(\boldsymbol{Big}_0, Ws_0, \boldsymbol{Zs}_0)&\end{aligned}$$

We will assume a left-to-right computation rule (i.e. the leftmost subgoal is selected at each goal). Then the variables which occur in such two atoms may possibly receive a substitution either when this clause is resolved with the current goal (i.e. when the head $qs([X_0|Xs_0], Ys_0, Zs_0)$ unifies with the selected subgoal of the goal) or when the first subgoal $part(X_0, Xs_0, Small_0, Big_0)$ of this clause succeeds. We next observe that the variables X_0, Ys_0, Zs_0, $Small_0$, and Big_0 occur both in the first two atoms and in the rest of the clause. Therefore, the substitutions received by such variables have an effect on the rest of the clause. Thus, we will push onto the stack the values received by these variables:

$$
\begin{aligned}
&\widehat{qs}(\langle St_0, [\boldsymbol{X}_0|Xs_0], \boldsymbol{Ys}_0, \boldsymbol{Zs}_0\rangle, St_0) \leftarrow \\
&\qquad \widehat{part}(\langle[\boldsymbol{X}_0, \boldsymbol{Ys}_0, \boldsymbol{Zs}_0, \boldsymbol{Small}_0, \boldsymbol{Big}_0|St_0], \boldsymbol{X}_0, Xs_0, \boldsymbol{Small}_0, \boldsymbol{Big}_0\rangle, \\
&\qquad\qquad [X_1, Ys_1, Zs_1, Small_1, Big_1|St_1]), \\
&\qquad qs(Small_1, Ys_1, [X_1|Ws_1]), \\
&\qquad qs(Big_1, Ws_1, Zs_1)
\end{aligned}
$$

Note that the terms which were pushed onto the stack (through variables subscripted with 0) are subsequently popped off the stack using variables with different names (variables subscripted with 1). The fact that the boldface variables do not occur below the dotted line will allow us to fold a predicate of the form $k(V, U)$ and transform into variables the terms in which such variables occur. We refer the reader to the appendix for the details on how to perform the above step through fold and unfold operations.

Now we remark that once the second subgoal $qs(Small_1, Ys_1, [X_1|Ws_1])$ has succeeded, the variables Big_1, Ws_1, and Zs_1, which occur in the last subgoal, may have received substitutions.

$$
\begin{aligned}
&\widehat{qs}(\langle St_0, [X_0|Xs_0], Ys_0, Zs_0\rangle, St_0) \leftarrow \\
&\qquad \widehat{part}(\langle[X_0, Ys_0, Zs_0, Small_0, Big_0|St_0], X_0, Xs_0, Small_0, Big_0\rangle, \\
&\qquad\qquad [X_1, Ys_1, \boldsymbol{Zs}_1, Small_1, \boldsymbol{Big}_1|St_1]), \\
&\qquad qs(Small_1, Ys_1, [X_1|\boldsymbol{Ws}_1]), \\
&\qquad qs(\boldsymbol{Big}_1, \boldsymbol{Ws}_1, \boldsymbol{Zs}_1)
\end{aligned}
$$

Hence, we push onto the stack the terms which unify with such variables and pop them off the stack through variables with a different name:

$$
\begin{aligned}
&\widehat{qs}(\langle St_0, [X_0|Xs_0], Ys_0, Zs_0\rangle, St_0) \leftarrow \\
&\qquad \widehat{part}(\langle[X_0, Ys_0, Zs_0, Small_0, Big_0|St_0], X_0, Xs_0, Small_0, Big_0\rangle, \\
&\qquad\qquad [X_1, Ys_1, \boldsymbol{Zs}_1, Small_1, \boldsymbol{Big}_1|St_1]), \\
&\qquad \widehat{qs}(\langle[\boldsymbol{Big}_1, \boldsymbol{Ws}_1, \boldsymbol{Zs}_1|St_1], Small_1, Ys_1, [X_1|\boldsymbol{Ws}_1]\rangle, \\
&\qquad\qquad [Big_2, Ws_2, Zs_2|St_2]), \\
&\qquad qs(Big_2, Ws_2, Zs_2)
\end{aligned}
$$

We can thus derive the following clause in relay form:

$$\begin{aligned}\widehat{qs}(\langle St_0, [X_0|Xs_0], Ys_0, Zs_0\rangle, St_3) \leftarrow \\ &\widehat{part}(\langle [X_0, Ys_0, Zs_0, Small_0, Big_0|St_0], X_0, Xs_0, Small_0, Big_0\rangle, \\ &\qquad [X_1, Ys_1, Zs_1, Small_1, Big_1|St_1]), \\ &\widehat{qs}(\langle [Big_1, Ws_1, Zs_1|St_1], Small_1, Ys_1, [X_1|\,Ws_1]\rangle, \\ &\quad [Big_2, Ws_2, Zs_2|St_2]), \\ &\widehat{qs}(\langle St_2, Big_2, Ws_2, Zs_2\rangle, \\ &\quad St_3)\end{aligned}$$

Finally, we can fold the completions of the following predicates, after using predicate substitutivity and symmetry as we did before:

$$\begin{aligned}&k_0(\langle St_0, [X_0|Xs_0], Ys_0, Zs_0\rangle,\ \langle [X_0, Ys_0, Zs_0, Small_0, Big_0|St_0], X_0, Xs_0, Small_0, Big_0\rangle) \leftarrow \\ &k_1([X_1, Ys_1, Zs_1, Small_1, Big_1|St_0],\ \langle [Big_1, Ws_1, Zs_1|St_1], Small_1, Ys_1, [X_1|\,Ws_1]\rangle) \leftarrow \\ &k_2([Big_2, Ws_2, Zs_2|St_2],\ \langle St_2, Big_2, Ws_2, Zs_2\rangle) \leftarrow\end{aligned}$$

The resulting clause in chain form is:

$$\begin{aligned}\widehat{qs}(U_0, St_3) \leftarrow\ & k_0(U_0, U_0'),\ \widehat{part}(U_0', U_1), \\ & k_1(U_1, U_1'),\ \widehat{qs}(U_1', U_2), \\ & k_2(U_2, U_2'),\ \widehat{qs}(U_2', St_3)\end{aligned}$$

Note that we can choose to push onto the stack other variables, and still derive correct clauses. However, if we do not choose the "right" variables, we may arrive at a clause which does not have relay form, and we may not be able to convert such a clause into chain form.

Observe also that there is no need to add a subgoal $k(V, U)$ relating the output of the last subgoal with the output of the head of the clause, since both outputs are the same variable St_3.

We can now state the following theorem.

Theorem 4.1 *Let P be a logic program, A the set of axioms of Lemma A.1 for all clauses in P, and $\hat{P}$ the program resulting from transforming each clause in P with Lemma A.1. Then $P \cup A \models \hat{P}$ and $\hat{P} \cup A \models P$.*

Proof By Lemma A.1, $P \cup A \models \hat{P}$. The logical implication $\hat{P} \cup A \models P$ can be seen as follows. Each clause in P is obtained by unfolding the predicates with symbols k_i and $\hat{p}_i$, and applying reflexivity to the subgoals of the form $St = St'$. □

5 Inductive assertions, subgoal induction, and logic programs

Our connection between tail-recursive logic programs and subgoal-induction verification conditions shows that verifying a logic program (in the sense of Kowalski) amounts to verifying a system of active graphs with subgoal induction. *This connection may be useful in cases where we can find a subgoal-induction verification more easily than we can directly verify the logic program.* We devote this section to such cases.

Forward and backward verification. The inductive-assertions method is often considered more natural than subgoal induction. A reason is perhaps that forward verification conditions can take the initial state of computations into account, whereas backward verification conditions cannot [14, page 191]. Another reason could be that the implication in the forward verification conditions goes in the direction of the computation, while the implication in the backward verification conditions goes *against* the direction of the computation [23].

Our theorem relating tail-recursive logic programs and subgoal induction can be exploited by observing that it is possible to obtain a backward verification from a forward verification [4, 8, 16]. We remark that the resulting backward verification may not correspond to the least model, even if the original forward verification does.

Recall (12):

$$\mathscr{f}_{\mathrm{s}}(X, X) \leftarrow \tag{33}$$
$$\mathscr{f}_{\mathrm{v}}(W, Y) \leftarrow \mathscr{f}_{\mathrm{s}}(W, X),\ a(X, Y) \tag{34}$$
$$\mathscr{f}_{\mathrm{v}}(W, Y) \leftarrow \mathscr{f}_{\mathrm{v}}(W, X),\ b(X, Y) \tag{35}$$
$$\mathscr{f}_{\mathrm{h}}(W, Y) \leftarrow \mathscr{f}_{\mathrm{v}}(W, X),\ c(X, Y) \tag{36}$$

Assume that we have a correct meaning of these predicates and that we wish to verify:

$$\mathscr{b}_{\mathrm{h}}(Z, Z) \leftarrow \tag{37}$$
$$\mathscr{b}_{\mathrm{v}}(X, Z) \leftarrow c(X, Y),\ \mathscr{b}_{\mathrm{h}}(Y, Z) \tag{38}$$
$$\mathscr{b}_{\mathrm{v}}(X, Z) \leftarrow b(X, Y),\ \mathscr{b}_{\mathrm{v}}(Y, Z) \tag{39}$$
$$\mathscr{b}_{\mathrm{s}}(X, Z) \leftarrow a(X, Y),\ \mathscr{b}_{\mathrm{v}}(Y, Z) \tag{40}$$

Theorem 5.1 (Morris and Wegbreit [16], King [8], Cousot [4]) *Given (33)–(36), the following predicates:*

$$\begin{aligned}
&\mathscr{b}_{\mathrm{h}}(X, Z) \leftrightarrow \mathscr{f}_{\mathrm{s}}(X, Z) \\
&\mathscr{b}_{\mathrm{s}}(X, Z) \leftrightarrow \mathscr{f}_{\mathrm{h}}(X, Z) \\
&\mathscr{b}_{\mathrm{v}}(X, Z) \leftrightarrow \forall W\,.\,[\mathscr{f}_{\mathrm{v}}(W, X) \rightarrow \mathscr{f}_{\mathrm{h}}(W, Z)]
\end{aligned} \tag{41}$$

verify (37)–(40).

Proof We will only verify (39), as the other verifications are similar. We want to prove that $\mathit{b}_{\mathrm{v}}(X, Z)$ holds, or equivalently, that:

$$\forall W\, \forall X\, \forall Z\, .\, [f_{\mathrm{v}}(W, X) \rightarrow f_{\mathrm{h}}(W, Z)] \tag{42}$$

holds. Let us take arbitrary elements $\mathbf{w}$, $\mathbf{x}$, and $\mathbf{z}$ in the domain of the intended interpretation and assume that the premise of (42) holds: $f_{\mathrm{v}}(\mathbf{w}, \mathbf{x})$. By $f_{\mathrm{v}}(\mathbf{w}, \mathbf{x})$, and the first subgoal of (39), given (35), we obtain:

$$f_{\mathrm{v}}(\mathbf{w}, \mathbf{y}) \tag{43}$$

By the second subgoal of (39), and (41):

$$f_{\mathrm{v}}(\mathbf{w}, \mathbf{y}) \rightarrow f_{\mathrm{h}}(\mathbf{w}, \mathbf{z}) \tag{44}$$

holds. By (43) and (44), we conclude that $f_{\mathrm{h}}(\mathbf{w}, \mathbf{z})$ holds. Hence (39) is verified. □

A tail-recursive Fibonacci program. Consider the following definition of Fibonacci numbers:

$$f(N) = \begin{cases} 1 & \text{if } N = 0, 1 \\ f(N-1) + f(N-2) & \text{if } N > 1 \end{cases}$$

The next tail-recursive program can be used to compute $f(N)$.

$$\begin{aligned} &\mathit{fib}(N, F) \leftarrow \mathit{fib}'(N, 0, 1, F) \\ &\mathit{fib}'(0, F_1, F_2, F_2) \leftarrow \\ &\mathit{fib}'(N, F_1, F_2, F) \leftarrow N > 0,\ N_1 \ \mathit{is}\ N - 1,\ F_{12} \ \mathit{is}\ F_1 + F_2, \\ &\qquad\qquad\qquad\qquad\quad \mathit{fib}'(N_1, F_2, F_{12}, F) \end{aligned}$$

Verifying this program directly is arguably more intricate than using Floyd's method on the corresponding active graph [23]. The trouble lies in the fact that the predicate with symbol fib' defines a generalization of Fibonacci numbers: Observe that the unit clause with this predicate symbol does not specify the first two numbers F_1 and F_2 of such a sequence. Hence, any meaning of such a predicate must allow for sequences having arbitrary first two elements.

For simplicity, we will rewrite the clauses using successor notation as:

$$\begin{aligned} &\mathit{fib}(N, F) \leftarrow \mathit{fib}'(N, 0, s(0), F) \\ &\mathit{fib}'(0, F_1, F_2, F_2) \leftarrow \\ &\mathit{fib}'(s(N), F_1, F_2, F) \leftarrow \mathit{fib}'(N, F_2, F_1 + F_2, F) \end{aligned}$$

(a) (b)

Figure 4: *(a) Iterative and (b) recursive active graphs for computing Fibonacci numbers.*

The transformed program is:

$$\widehat{fib'}(\langle St, 0, F_1, F_2, F_2\rangle, St) \leftarrow \tag{45}$$

$$\widehat{fib'}(X, Z) \leftarrow k(X, Y),\ \widehat{fib'}(Y, Z) \tag{46}$$

$$k(\langle St, s(N), F_1, F_2, F\rangle, \langle St, N, F_2, F_1 + F_2, F\rangle) \leftarrow \tag{47}$$

If interpreted as the *backward* description of an active graph, the clauses (45)–(47) correspond to the active graph in Figure 4(a). A forward verification of this active graph can be obtained by observing that in forward computations, the variables F_1 and F_2 successively take the values (returning to decimal notation): $(0,1)$, $(1,1)$, $(1,2)$, $(2,3)$, $(3,5)$, $\ldots$ for $N = N_0,\ N_0 - 1,\ N_0 - 2,\ N_0 - 3,\ N_0 - 4, \ldots$, respectively, where N_0 is the initial value of N. From this, we can inductively infer an invariant: $F_1 = f(N_0 - N - 1)\ \&\ F_2 = f(N_0 - N)$, if we extend f with $f(-1) = 0$. Note that even though we have grouped the output F of fib' along with the input argument places N, F_1, and F_2, there is no need to relate F with N, F_1, and F_2 (as happens in the backward verification, as we will see).

For a backward verification, we could infer the least model as follows [23]: Consider two terms α and β representing integers. The SLD-derivation that starts with the goal $\leftarrow fib'(0, \alpha, \beta, F)$ and uses the first clause defining fib' succeeds with $F = \beta$. Similarly, using first the second clause and then the first, a derivation starting with $\leftarrow fib'(1, \alpha, \beta, F)$ succeeds with $F = \alpha + \beta$. Likewise, $\leftarrow fib'(2, \alpha, \beta, F)$ is the initial goal of a derivation that succeeds with $F = \alpha + 2\beta$. In general, there is a derivation starting with $\leftarrow fib'(N, \alpha, \beta, F)$ that succeeds with $F = f(N-1)\alpha + f(N)\beta$. This suggests the following meaning of fib': $fib'(N, F_1, F_2, F)$ holds iff $F = f(N-1)F_1 + f(N)F_2$, which is the relation assigned to fib' in the least model.

Observe that, in general, it may be preferable obtaining a forward verification over a backward verification of a program: To infer a forward assertion we can symbolically evaluate the active graph for the first few iterations [24, page 511], computing one tuple for each iteration (i.e. we wish to infer a relation holding between *initial* and current states, by evaluating the commands in a forward direction). By contrast, to arrive at the least model of the tail-recursive program we had to construct as many refutations as examples we needed to infer such a model (i.e. we wish to infer a relation holding between current and *final* states, also by evaluating the commands in a forward direction). Computing one iteration requires evaluating the command once, whereas computing one refutation requires evaluating the command a number of times proportional to the length of the refutation.[2]

There could be another way of verifying this logic program with Floyd's method, and without recurring to Theorem 5.1. Since we have admitted subroutines, we could regard the clauses (45)–(47) as related to the *forward* description of a different active graph, with a recursive call at the end, as illustrated in Figure 4(b).

The forward description of such an active graph is:

$$
\begin{aligned}
&f_{\mathrm{s}}(X, X) \leftarrow \\
&f_{\mathrm{v}}(X, Z) \leftarrow f_{\mathrm{s}}(X, Y),\ k(Y, Z) \\
&f_{\mathrm{h}}(X, Z) \leftarrow f_{\mathrm{v}}(X, Y),\ \widehat{fib}'(Y, Z) \\
&f_{\mathrm{h}}(X, Z) \leftarrow f_{\mathrm{s}}(X, Y),\ l(Y, Z)
\end{aligned}
$$

To obtain the clauses (45)–(47), observe first that $f_{\mathrm{h}}(X, Z)$ is $\widehat{fib}'(X, Z)$. Next, we can unfold $f_{\mathrm{s}}(X, Y)$ and $f_{\mathrm{v}}(X, Y)$, to get (45)–(47).

Verifying this active graph, however, is essentially the same as obtaining the least fixpoint as we did above, since we must evaluate the recursive call to $\widehat{fib}'$ for every 4-tuple needed to infer the meaning of this predicate.

6 Concluding remarks

Transforming definite clauses into chain form. The coincidence between subgoal induction and the verification of tail-recursive logic programs is not surprising, given the results of [2]. There, Clark and van Emden observed that the subgoal-induction verification conditions of flowcharts without procedures can be written as clauses of the form:

$$p(X, Z) \leftarrow c(X, Y),\ q(Y, Z)$$

where p and q are associated with points of the flowchart and c is associated with a primitive command. Here, we have extended this result to arbitrary definite clauses through a transformation which converts a definite clause into chain form:

$$\hat{p}_0(X_0, X_n) \leftarrow \hat{p}_1(X_0, X_1),\ \hat{p}_2(X_1, X_2),\ \ldots,\ \hat{p}_n(X_{n-1}, X_n)$$

in addition to other, unit clauses. The resulting clauses are logically implied by a conservative extension of the original clause.

[2]See, however, [16] for programs in which a backward verification may be preferable to a forward verification.

A similar transformation was used in [18, 19] to obtain inference systems from parsers for context-free grammars. Although such a transformation also produces a chain program, it is restricted to fixed-mode logic programs. Also, this fixed-mode transformation may produce a chain clause which is not tail-recursive even if the original clause is so. For the present paper, we have developed a different transformation, which overcomes both these difficulties.

Connection between tail-recursive clauses and subgoal induction. After producing a chain clause from a given tail-recursive definite clause (by applying this new transformation) it is then possible to regard the chain clause as a set of subgoal-induction verification conditions of some system of active graphs.

Because the predicates of the chain clause are simply related to those of the original clause:

$$\hat{p}(\langle St, t_1, \ldots, t_m\rangle, St') \leftrightarrow St = St' \ \&\ p(t_1, \ldots, t_m)$$

a verification of the *original* definite clause in turn implies a subgoal-induction verification.

The implication in the other direction also holds, so that if we can verify the associated system of active graphs with subgoal induction, we immediately have a verification of the original definite clause.

Tail-recursive Prolog programs. We concentrated on tail-recursive *logic* programs as an abstraction of tail-recursive *Prolog* programs. In such an abstraction, (1) a program is a *set* of clauses, and (2) the subgoals which are not the last one in the recursive clause may succeed nondeterministically. As a result, our connection between subgoal induction and logic-program verification extends to Prolog programs which violate conditions (Section 2) for the application of tail-recursion optimization [15, 21]. Observe first that there is no order in the arcs incoming to a node in an active graph, and second, that the relational semantics we used to characterize active graphs can capture the meaning of *nondeterministic* commands. Hence, a verification of a system of active graphs may be used to verify a Prolog program in which (1) a tail-recursive clause does not occur last in the definition of a predicate or (2) there are nondeterministic subgoals which do not occur last in such a subgoal.

Non-tail-recursive logic programs. We also remark that subgoal induction can be applied to the verification of logic programs which are not tail-recursive, provided that we give temporal precedence to the subgoals of each clause. Note that any chain clause can be viewed as a logical consequence of the backward description of some system of active graphs (with a procedure call instead of a *goto* if the clause is not tail-recursive). This result, combined with the transformation of definite clauses into chain form, allows us to use subgoal induction in the verification of arbitrary logic programs.

Contribution. The contribution of this paper, we believe, is twofold. On the one hand, we have provided a correctness proof of a compilation process which translates tail-recursive clauses into flowcharts. The formalization of such a process with unfold/fold operations can be seen a contribution to logic program transformation. Pettorossi and Proietti observe in [17, page 261] that "... program manipulations, such as compilation and code optimization, can ... be viewed as an application of some ... transformation techniques." Typically, however, systems for transforming logic programs, which attempt to produce a logic program executable as a flowchart only obtain a program with tail-recursive clauses, and do not transform the program further. We have filled in this last stage of program transformation by showing that it is also possible to reconstruct the low-level process of tail-recursion optimization with unfold/fold applications.

On the other hand, our work provides two new methods for verifying tail-recursive logic programs in particular, and arbitrary logic programs in general. We showed first that verifying a logic program amounts to a backward verification of a system of active graphs. (If such a program is tail-recursive, then the system of active graphs will have a loop; otherwise it will have a procedure call instead.) We then recalled a result [4, 8, 16] for mechanically obtaining a backward verification from a forward verification. Thus, a logic programmer can use either a backward (i.e. subgoal induction) or a forward method (i.e. Floyd's inductive-assertions method) to verify the original program. This last possibility is valuable because forward verifications are often considered more natural than backward verifications [14, page 191].

We hope that our results help understanding the relationship between declarative and procedural knowledge in logic programs.

Acknowledgments

Many of these ideas resulted from discussions with Maarten van Emden. We also benefited from conversations with Felipe Bracho and comments made at the seminar organized by Sergio Rajsbaum at the Instituto de Matemáticas. Fernando Magariños helped us with LaTeX.

Appendix

A Conversion of logic programs into chain form

This appendix is devoted to our transformation of definite clauses into chain form. We will give a sequence of transformation steps each of which produces a program logically implied by a conservative extension of the original clause.

After the sketch proof of the following lemma, we give an example which should help in understanding such a sketch proof. Hence, we suggest reading both the sketch proof and the example concurrently.

We will use the *standard equality theory*. Given a program P, this theory consists of the following axioms:

$$
\begin{aligned}
&X = X \leftarrow \\
&X = Y \leftarrow Y = X \\
&X = Z \leftarrow X = Y,\ Y = Z \\
&\{f(X_1, \ldots, X_{n_f}) = f(Y_1, \ldots, Y_{n_f}) \leftarrow X_1 = Y_1,\ \ldots,\ X_{n_f} = Y_{n_f} | \\
&\qquad\qquad\qquad f \text{ is a function symbol occurring in } P\} \\
&\{p(X_1, \ldots, X_{n_p}) \leftarrow X_1 = Y_1,\ \ldots,\ X_{n_p} = Y_{n_p},\ p(Y_1, \ldots, Y_{n_p}) | \\
&\qquad\qquad\qquad p \text{ is a predicate symbol occurring in } P\}
\end{aligned}
$$

which are called, reflexivity, symmetry, transitivity, function substitutivity, and predicate substitutivity.

Lemma A.1 *Let C be a definite clause:*

$$A_0 \leftarrow A_1,\ A_2,\ \ldots,\ A_n$$

where each A_i is of the form $p_i(t_{1,i}, \ldots, t_{r_i,i})$ and let:

$$\Pi_i = (var(A_0) \cup \cdots \cup var(A_i)) \cap (var(A_{i+1}) \cup \cdots \cup var(A_n))$$

for $i = 1, \ldots, n$, and $\Pi_0 = \emptyset$. Then the clause C':

$$
\begin{aligned}
\hat{p}_0(U_0, U_n) \leftarrow k_0(U_0, U_0'),\ \hat{p}_1(U_0', U_1),\ k_1(U_1, U_1'),\ \hat{p}_2(U_1', U_2),\ \ldots,& \\
k_{n-1}(U_{n-1}, U_{n-1}'),\ \hat{p}_n(U_{n-1}', U_n)&
\end{aligned}
$$

is logically implied by C, the standard equality theory, the "iff" version of the function substitutivity axiom for the list-constructor function symbol:

$$[U|V] = [U'|V'] \leftrightarrow U = U' \ \&\ V = V'$$

the definitions of the $\hat{p}$'s:

$$\hat{p}_i(\langle St, X_{1,i}, \ldots, X_{r_i,i}\rangle, St') \leftrightarrow St = St' \mathbin{\&} p_i(X_{1,i}, \ldots, X_{r_i,i}) \qquad i = 0, \ldots, n$$

and the completed definitions of:

$$k_0(\langle \Sigma_0, t_{1,0}, \ldots, t_{r_0,0}\rangle, \langle \Sigma_1, t_{1,1}, \ldots, t_{r_1,1}\rangle) \leftarrow$$
$$k_i(\Sigma_i, \langle \Sigma_{i+1}, t_{1,i+1}, \ldots, t_{r_{i+1},i+1}\rangle) \leftarrow \qquad i = 1, \ldots, n-1$$

where Σ_i is any list of the form $[X_{1,i}, \ldots, X_{d_i,i}|St]$, such that $\{X_{1,i}, \ldots, X_{d_i,i}\} = \Pi_i$, if $\Pi_i \neq \emptyset$, and Σ_i is St if $\Pi_i = \emptyset$, for $i = 0, \ldots, n$.

Proof sketch For simplicity, we will assume first that each variable in C occurs at most twice in its body. Later we will explain how to deal with clauses having variables with more occurrences in their body.
Stage 1.

1. Apply predicate substitutivity to $A_2, \ldots, A_n$, respectively.

   The resulting clause C_1 has the same head and leftmost subgoal as C, but instead of each A_i, for $i = 2, \ldots, n$, it has the atom $p_i(W_{1,i}, \ldots, W_{r_i,i})$ together with the equations $t_{1,i} = W_{1,i}, \ldots, t_{r_i,i} = W_{r_i,i}$, where each $W_{j,i}$ does not occur anywhere else.

2. Repetitively apply function substitutivity to all equations with a nonvariable term on the left-hand side until all equations have a variable on the left-hand side.

   The resulting clause C_2 has the same head and leftmost subgoal as C, but instead of each A_i, for $i = 2, \ldots, n$, it has a variant B_i together with the equations $X_{1,i} = Z_{1,i}, \ldots, X_{d_i,i} = Z_{d_i,i}$, where $\mathrm{var}(A_i) = \{X_{1,i}, \ldots, X_{d_i,i}\}$ and $\mathrm{var}(B_i) = \{Z_{1,i}, \ldots, Z_{d_i,i}\}$.

3. Define $\phi(X) = \min\{i | X \in \mathrm{var}(A_i)\}$, and $\gamma(Z) = i$ for each $Z \in \mathrm{var}(B_i)$. Apply reflexivity to each equation $X = Z$ such that $\phi(X) = \gamma(Z) \geq 2$, obtaining clause C_3.

Stage 2.

1. Unfold C_3 on the "if" part of the definition of $\hat{p}_0$ and apply the substitution $\{St/St_0, St'/St_n\}$, obtaining C_4. Define a function σ (for "subscript") from $\mathrm{var}(C_4)$ to the set of nonnegative integers as follows:

   (a) If X occurs in the first argument place of the head of C_4, then $\sigma(X) = 0$.

   (b) If X occurs in the second argument place of the head of C_4, then $\sigma(X) = n$.

(c) Otherwise, $\sigma(X) = i-1$, where X occurs in a subgoal with predicate symbol p_i, for $i = 1, \ldots, n$.

Note that σ is well defined, because the subgoals with predicate symbol p_i, $i = 1, \ldots, n$ have no variables in common.

2. Repetitively apply transitivity to C_4, replacing the equation $St_0 = St_n$ by the equations $St_0 = St_1,\ St_1 = St_2,\ \ldots,\ St_{n-1} = St_n$.

3. Repetitively apply transitivity replacing each equation $X = Z$ such that:

$$\sigma(X) + 1 < \sigma(Z)$$

by the equations:

$$X = Y_{\sigma(X)+1},\ Y_{\sigma(X)+1} = Y_{\sigma(X)+2},\ \ldots,\ Y_{\sigma(Z)-1} = Z$$

obtaining clause C_5.

4. Define a function σ' from $\mathrm{var}(C_5)$ to the set of nonnegative integers as follows:

(a) If $Y \in \mathrm{var}(C_4)$, then $\sigma'(Y) = \sigma(Y)$.

(b) Otherwise, Y occurs on the right-hand side of an equation $X = Y$. Then $\sigma'(Y) = \sigma'(X) + 1$.

Note that σ' is well defined, because a variable cannot occur in more than one right-hand side of an equation, and there are no cycles.

Let $E_j = \{X_{1,j-1} = Y_{1,j},\ X_{2,j-1} = Y_{2,j},\ \ldots,\ X_{d_{j-1},j-1} = Y_{d_{j-1},j}\} \cup \{St_{j-1} = St_j\}$ be the largest subset of equations in C_5 such that $\sigma'(X_{k,j-1}) = j-1$, for $j = 1, \ldots, n$. Let u_{j-1} be $[X_{1,j-1}, \ldots, X_{d_{j-1},j-1}|St_{j-1}]$ if $d_{j-1} > 0$, or St_{j-1} if $d_{j-1} = 0$, and v_j be $[Y_{1,j}, \ldots, Y_{d_{j-1},j}|St_j]$ if $d_{j-1} > 0$, or St_j if $d_{j-1} = 0$. Replace the equations in E_j by the equation $u_{j-1} = v_j$, by repetitively folding function substitutivity for the list-constructor function symbol, for $j = 1, \ldots, n$. Let this clause be C_6.

Consider any variable W in C which occurs both in A_i and in A_k with $i < k$, where $i \geq 1$. Then in C_4 there is an equation $X_{i-1} = Z_{k-1}$, with $\sigma(X_{i-1}) = i-1$ and $\sigma(Z_{k-1}) = k-1$. So in C_5 for each $j = i, i+1, \ldots, k-1$ there is an equation $X_{j-1} = Y_j$ such that $\sigma'(X_{j-1}) = j-1$. Because W was arbitrary, then for a fixed j, E_j has one equation for each variable which occurs both in A_i with $i \leq j$ and in A_k with $j < k$, where $i \geq 1$. Therefore u_{j-1} has one X_{j-1} for each variable in $(\mathrm{var}(A_1) \cup \cdots \cup \mathrm{var}(A_j)) \cap (\mathrm{var}(A_{j+1}) \cup \cdots \cup \mathrm{var}(A_n))$. In addition, E_j has an equation for each variable which occurs both in A_0 and in A_k, where $1 \leq j < k$. Since these equations, together with $St_{j-1} = St_j$, are all the equations in E_j, then u_{j-1} only has one X_{j-1} for each variable in $(\text{var}(A_0) \cup \text{var}(A_1) \cup \cdots \cup \text{var}(A_j)) \cap (\text{var}(A_{j+1}) \cup \cdots \cup \text{var}(A_n))$, in addition to St_{j-1}.

5. Fold in C_6 the definitions of $\hat{p}_i$ by replacing $u_{i-1} = v_i$ and $p_i(t_{1,i}, t_{2,i}, \ldots, t_{r_i,i})$ by $\hat{p}_i(\langle u_{i-1}, t_{1,i}, t_{2,i}, \ldots, t_{r_i,i}\rangle, v_i)$. Let this clause be C_7.

   C_7 is of the form:

$$\hat{p}_0(s_0, s'_n) \leftarrow \hat{p}_1(s'_0, s_1),\ \hat{p}_2(s'_1, s_2),\ \ldots,\ \hat{p}_n(s'_{n-1}, s_n)$$

   where:

$$(\text{var}(s_i) \cup \text{var}(s'_i)) \cap (\text{var}(s_j) \cup \text{var}(s'_j)) = \emptyset$$

   for $i, j = 0, \ldots, n$ and $i \neq j$. This will allow us to fold the k's.

Stage 3.

1. Resolve C_7 with predicate substitutivity for $\hat{p}_0$.

2. Apply predicate substitutivity to the subgoals with predicate symbol $\hat{p}_i$, for $i = 1, \ldots, n$, and then apply symmetry to the equations generated in this step.

3. Fold the definitions of k_i for $i = 0, \ldots, n-1$, and apply reflexivity to the equations having St_n on the left-hand side.

The final clause has the claimed form. The soundness of the fold steps can be proved using, for instance, the result in [20, pages 57, 58].

If there is a variable with more than two occurrences in the body of C, say three, then the desired clause can be obtained by changing the above procedure as follows. Assume that in C_4 we have equations $X = Z'$ and $X = Z$ with $\sigma(X) < \sigma(Z') < \sigma(Z)$. Then in Step 3 of Stage 2, using transitivity, but this time also factoring, we replace such equations by:

$$X = Y_{\sigma(X)+1},\ Y_{\sigma(X)+1} = Y_{\sigma(X)+2},\ \ldots,\ Y_{\sigma(Z')-1} = Z',\ Z' = Y_{\sigma(Z')+1},\ \ldots,\ Y_{\sigma(Z)-1} = Z$$

The rest of the transformation procedure would continue in essentially the same way. We conclude that the lemma holds. $\square$

Example. To illustrate this sketch proof, consider the nonunit clause of the difference-list quicksort program. In this case, C is:

$$\begin{aligned} &qs([X_0|Xs_0], Ys_0, Zs_0) \leftarrow \\ &\qquad part(X_0, Xs_0, Small_0, Big_0), \\ &\qquad qs(Small_0, Ys_0, [X_0|Ws_0]), \\ &\qquad qs(Big_0, Ws_0, Zs_0) \end{aligned}$$

C_1 is:

$$
\begin{aligned}
&qs([X_0|Xs_0], Ys_0, Zs_0) \leftarrow \\
&\qquad part(X_0, Xs_0, Small_0, Big_0), \\
&\qquad qs(Small_1, Ys_1, W_{3,2}), \\
&\qquad\qquad Small_0 = Small_1,\ \ Ys_0 = Ys_1,\ \ [X_0|\,Ws_0] = W_{3,2}, \\
&\qquad qs(Big_2, Ws_2, Zs_2), \\
&\qquad\qquad Big_0 = Big_2,\ \ Ws_0 = Ws_2,\ \ Zs_0 = Zs_2
\end{aligned}
$$

C_2 is:

$$
\begin{aligned}
&qs([X_0|Xs_0], Ys_0, Zs_0) \leftarrow \\
&\qquad part(X_0, Xs_0, Small_0, Big_0), \\
&\qquad qs(Small_1, Ys_1, [X_1|\,Ws_1]), \\
&\qquad\qquad Small_0 = Small_1,\ \ Ys_0 = Ys_1,\ \ X_0 = X_1,\ \ Ws_0 = Ws_1, \\
&\qquad qs(Big_2, Ws_2, Zs_2), \\
&\qquad\qquad Big_0 = Big_2,\ \ Ws_0 = Ws_2,\ \ Zs_0 = Zs_2
\end{aligned}
$$

C_3 is:

$$
\begin{aligned}
&qs([X_0|Xs_0], Ys_0, Zs_0) \leftarrow \\
&\qquad part(X_0, Xs_0, Small_0, Big_0), \\
&\qquad qs(Small_1, Ys_1, [X_1|\,Ws_1]),\ \ Small_0 = Small_1,\ \ Ys_0 = Ys_1,\ \ X_0 = X_1, \\
&\qquad qs(Big_2, Ws_2, Zs_2),\ \ Big_0 = Big_2,\ \ Ws_1 = Ws_2,\ \ Zs_0 = Zs_2
\end{aligned}
$$

C_4 is:

$$
\begin{aligned}
&\widehat{qs}(\langle St_0, [X_0|Xs_0], Ys_0, Zs_0\rangle, St_3) \leftarrow St_0 = St_3, \\
&\qquad part(X_0, Xs_0, Small_0, Big_0), \\
&\qquad qs(Small_1, Ys_1, [X_1|\,Ws_1]),\ \ Small_0 = Small_1,\ \ Ys_0 = Ys_1,\ \ X_0 = X_1, \\
&\qquad qs(Big_2, Ws_2, Zs_2),\ \ Big_0 = Big_2,\ \ Ws_1 = Ws_2,\ \ Zs_0 = Zs_2
\end{aligned}
$$

C_5 is:

$$
\begin{aligned}
&\widehat{qs}(\langle St_0, [X_0|Xs_0], Ys_0, Zs_0\rangle, St_3) \leftarrow St_0 = St_1,\ \ St_1 = St_2,\ \ St_2 = St_3, \\
&\qquad part(X_0, Xs_0, Small_0, Big_0), \\
&\qquad qs(Small_1, Ys_1, [X_1|\,Ws_1]),\ \ Small_0 = Small_1,\ \ Ys_0 = Ys_1,\ \ X_0 = X_1, \\
&\qquad qs(Big_2, Ws_2, Zs_2),\ \ Big_0 = Big_1,\ \ Big_1 = Big_2, \\
&\qquad\qquad Ws_1 = Ws_2, \\
&\qquad\qquad Zs_0 = Zs_1,\ \ Zs_1 = Zs_2
\end{aligned}
$$

C_6 is:

$$\begin{aligned}\widehat{qs}(\langle St_0, [X_0|Xs_0], Ys_0, Zs_0\rangle, St_3) \leftarrow\ & \\ & [X_0, Ys_0, Zs_0, Small_0, Big_0|St_0] = [X_1, Ys_1, Zs_1, Small_1, Big_1|St_1], \\ & part(X_0, Xs_0, Small_0, Big_0), \\ & [Big_1, Ws_1, Zs_1|St_1] = [Big_2, Ws_2, Zs_2|St_2], \\ & qs(Small_1, Ys_1, [X_1|Ws_1]), \\ & St_2 = St_3, \\ & qs(Big_2, Ws_2, Zs_2)\end{aligned}$$

C_7 is:

$$\begin{aligned}\widehat{qs}(\langle St_0, [X_0|Xs_0], Ys_0, Zs_0\rangle, St_3) \leftarrow\ & \\ & \widehat{part}(\langle [X_0, Ys_0, Zs_0, Small_0, Big_0|St_0], X_0, Xs_0, Small_0, Big_0\rangle, \\ & \qquad [X_1, Ys_1, Zs_1, Small_1, Big_1|St_1]), \\ & \widehat{qs}(\langle [Big_1, Ws_1, Zs_1|St_1], Small_1, Ys_1, [X_1|Ws_1]\rangle, \\ & \quad [Big_2, Ws_2, Zs_2|St_2]), \\ & \widehat{qs}(\langle St_2, Big_2, Ws_2, Zs_2\rangle, \\ & \quad St_3)\end{aligned}$$

The k's are:

$$\begin{aligned}& k_0(\langle St, [X|Xs], Ys, Zs\rangle,\ \langle [X, Ys, Zs, Small, Big|St], X, Xs, Small, Big\rangle) \leftarrow \\ & k_1([X, Ys, Zs, Small, Big|St],\ \langle [Big, Ws, Zs|St], Small, Ys, [X|Ws]\rangle) \leftarrow \\ & k_2([Big, Ws, Zs|St],\ \langle St, Big, Ws, Zs\rangle) \leftarrow\end{aligned}$$

The clause after Stage 3 is:

$$\begin{aligned}\widehat{qs}(U_0, St_3) \leftarrow\ & k_0(U_0, U_0'),\ \widehat{part}(U_0', U_1), \\ & k_1(U_1, U_1'),\ \widehat{qs}(U_1', U_2), \\ & k_2(U_2, U_2'),\ \widehat{qs}(U_2', St_3)\end{aligned}$$

References

[1] Krzysztof R. Apt. Logic programming. In Jan Van Leeuwen, editor, *Handbook of Theoretical Computer Science, vol. B: Formal Models and Semantics*, pages 493–574. Elsevier Science Publishers B. V., 1990.

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

[3] Keith Leonard Clark. Predicate logic as a computational formalism. Technical Report 79/59, Dept. of Computing and Control, Imperial College, 1979.

[4] Patrick Cousot. Methods and logics for proving programs. In Jan Van Leeuwen, editor, *Handbook of Theoretical Computer Science, vol. B: Formal Models and Semantics*, pages 841–993. Elsevier Science Publishers B. V., 1990.

[5] Joost Engelfriet. *Simple Program Schemes and Formal Languages.* Springer-Verlag, 1974. Lecture Notes in Computer Science No. 20.

[6] Robert W. Floyd. Assigning meanings to programs. In J.T. Schwartz, editor, *Proc. Symp. Appl. Math.*, volume XIX, pages 19–32. Amer. Math. Soc., 1967.

[7] Stanley Gill. Automatic computing: its problems and prizes. *Computer Journal*, 8(3):177–189, 1965.

[8] James C. King. Program correctness: On inductive assertion methods. *IEEE Transactions on Software Engineering*, 6(5):465–479, 1980.

[9] D.E. Knuth. Structured programming with *goto* statements. *Computing Surveys*, 6(4):261–301, 1974.

[10] D.E. Knuth and R.W. Floyd. Notes on avoiding *goto* statements. *Information Processing Letters*, 1:23–31, 1971.

[11] R. Kowalski. The relation between logic programming and logic specification. In C.A.R. Hoare and J.C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 11–27. Prentice Hall, 1985.

[12] Robert Kowalski. Logic programming principles. In Anthony Ralston and Edwin D. Reilly, editors, *Encyclopedia of Computer Science*, pages 778–783. Van Nostrand Reinhold, 3rd edition, 1993.

[13] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 2nd edition, 1987.

[14] Jacques Loeckx and Kurt Sieber. *The Foundations of Program Verification.* John Wiley & Sons, 2nd edition, 1987.

[15] David Maier and David S. Warren. *Computing with Logic. Logic Programming with Prolog.* Benjamin/Cummings, 1988.

[16] J.H. Morris and B. Wegbreit. Subgoal induction. *Communications of the ACM*, 20:209–222, 1977.

[17] Alberto Pettorossi and Maurizio Proietti. Transformation of logic programs: Foundations and techniques. *The Journal of Logic Programming*, 19, 20:261–320, 1994.

[18] David A. Rosenblueth. Chart parsers as proof procedures for fixed-mode logic programs. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 1125–1132, Tokyo, Japan, 1992. (Extended version: Chart parsers as inference systems for fixed-mode logic programs. To appear in *New Generation Computing*).

[19] David A. Rosenblueth and Julio C. Peralta. LR inference: Inference systems for fixed-mode logic programs, based on LR parsing. In Maurice Bruynooghe, editor, *Proceedings of the International Logic Programming Symposium*, pages 439–453, Ithaca, N.Y., U.S.A., 1994. (Extended version: SLR inference: an inference system for fixed-mode logic programs, based on SLR parsing. To appear in *The Journal of Logic Programming*).

[20] Joseph R. Shoenfield. *Mathematical Logic*. Addison-Wesley, 1967.

[21] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. MIT Press, 2nd edition, 1994.

[22] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5:285–309, 1955.

[23] M.H. van Emden and David A. Rosenblueth. A pure Prolog program is not necessarily a logic program. Technical Report DCS-129-IR, Department of Computer Science, University of Victoria, Victoria, Canada, 1990.

[24] Ben Wegbreit. Complexity of synthesizing inductive assertions. *Journal of the Association for Computing Machinery*, 24(3):504–512, 1977.