Program transformation for development, verification, and synthesis of programs

Fabio Fioravanti, Alberto Pettorossi, Maurizio Proietti, and Valerio Senni

Abstract: This paper briefly describes the use of the program transformation methodology for the development of correct and efficient programs. In particular, we will refer to the case of constraint logic programs and, through some examples, we will show how by program transformation, one can improve, synthesize, and verify programs.

Keywords: Constraint logic programming, model checking, program development, program synthesis, unfold/fold program transformation, software verification

1. Introduction

The program transformation methodology has been introduced in the case of functional programs by Burstall and Darlington [4] and then it has been adapted to logic programs by Hogger [13] and Tamaki and Sato [30]. The main idea of this methodology is to transform, maybe in several steps and by applying various transformation rules, the given initial program into a final program with the aim of improving its efficiency and preserving its correctness. If the initial program encodes a declarative specification of a problem and the final program encodes an efficient algorithm to solve that problem, then program transformation is equivalent to program synthesis. Thus, program transformation can be viewed as a technique both: (i) for program improvement and advanced compilation, and (ii) for program synthesis and program derivation.

In recent years program transformation has also been used as a technique for program verification. It has been shown, in fact, that via program transformation one can perform model checking and, in general, one can prove properties of infinite state systems that cannot be analyzed by using standard model checking techniques. In this paper we will illustrate the three uses of the program transformation methodology we mentioned above, namely, program improvement, program synthesis, and program verification. In particular, we will consider the case of specifications and algorithms written as constraint logic programs [14] and we will consider the following transformation rules: definition, unfolding, folding, goal replacement, and clause splitting [2, 7, 10, 11, 27, 30]. These rules are correct in the sense that they preserve the perfect model semantics [27] and they are applied according to some specific strategies, which ensure that the objectives of the transformations are actually achieved. This approach to program transformation is, thus, called the \textit{‘rules + strategies approach’}.

Transformation rules similar to those we consider, have been proposed also for: (i) concurrent constraint logic programs [8], (ii) constraint handling rules [29],
and (iii) functional logic programs [1], and thus, the transformation methodology we will present, can also be used for those classes of programs. In the literature one can find also other sets of transformation rules which are shown to be correct with respect to other semantics (see, for instance [28] for rules which preserve the well-founded semantics).

2. Program improvement

Programs are often written in a parametric form so that they can be reused in different contexts, and when a parametric program is reused, one may want to transform it for taking advantage of the new context of use. This transformation, called program specialisation [12, 15, 18], often allows great efficiency improvements. Let us present an example of this transformation by deriving a deterministic, specialized pattern matcher starting from a given nondeterministic, parametric pattern matcher and a given specific pattern.

In this example we consider the matching relation \( le_m(P, S) \) on strings of numbers which holds between a pattern \( P = [p_1, \ldots, p_n] \) and a string \( S \) if in \( S \) there is a substring \( Q = [q_1, \ldots, q_n] \) such that for \( i = 1, \ldots, n \), \( p_i \leq q_i \). (This example can be generalized by considering, instead of \( p_i \leq q_i \), any relation that can be expressed via a constraint logic program.)

The following constraint logic program can be taken as the specification of our general pattern matching problem:

\[
\begin{align*}
1. le_m(P, S) & \leftarrow ap(B, C, S) \land le(C, P) \\
2. ap([], Ys, Ys) & \leftarrow \\
3. ap([X], Ys, [X|Ys]) & \leftarrow ap(Xs, Ys, Ys) \\
4. le([], []) & \leftarrow \\
5. le([X], [Y|Ys]) & \leftarrow X \leq Y \land le(Xs, Ys)
\end{align*}
\]

where \( ap \) denotes the familiar list concatenation predicate. Now let us specialize this general program for a specific pattern \( P \), say \( [1, 0, 2] \). The transformation starts off by applying the so-called definition rule, thereby introducing the following clause:

\[
\begin{align*}
6. le_m([1,0,2], S) & \leftarrow le_m([1,0,2], S)
\end{align*}
\]

The transformation rules will be applied according to the so-called Determinization Strategy [10]. This strategy, which will not be presented here, allows a fully automatic derivation of a deterministic, efficient pattern matcher. First, clause 6 is unfolded w.r.t. the atom \( le_m([1,0,2], S) \), that is, the atom \( le_m([1,0,2], S) \), which is an instance of the head of clause 1, is replaced by the corresponding instance of the body of clause 1.

The result is:

\[
\begin{align*}
7. le_m(S) & \leftarrow ap(B, C, S) \land ap(A, Q, B) \\
& \quad \land le([1,0,2], Q)
\end{align*}
\]

Then, in order to fold clause 7, the following definition is introduced:

\[
\begin{align*}
8. new1(S) & \leftarrow ap(B, C, S) \land ap(A, Q, B) \\
& \quad \land le([1,0,2], Q)
\end{align*}
\]

and then clause 7 is folded, that is (the instance of) the body of clause 8 which occurs in the body of clause 7 is replaced by (the corresponding instance of) the head of clause 8. The result is:

\[
\begin{align*}
9. le_m(S) & \leftarrow new1(S)
\end{align*}
\]

Now, the clause splitting rule is applied to clause 11. We have two cases: (i) \( 1 \leq X \), and (ii) \( 1 > X \), and we get the following two clauses:

\[
\begin{align*}
10. new1([X|Xs]) & \leftarrow 1 \leq X \land ap(B, C, Xs) \\
& \quad \land le([0,2], Q) \\
11. new1([X|Xs]) & \leftarrow ap(B, C, Xs) \land ap(A, Q, B) \\
& \quad \land le([1,0,2], Q)
\end{align*}
\]

Now, in order to fold clauses 10 and 12, we introduce these two clauses defining the predicate \( new2: \)

\[
\begin{align*}
12. new2(Xs) & \leftarrow ap(Q, C, Xs) \land le([0,2], Q) \\
13. new2(Xs) & \leftarrow ap(B, C, Xs) \land ap(A, Q, B) \\
& \quad \land le([1,0,2], Q)
\end{align*}
\]

Next, in order to fold clauses 10 and 12, we introduce these two clauses defining the predicate \( new2: \)

Then clauses 10 and 12 are folded by using clauses 14 and 15. Moreover, clause 13 is folded by using clause 8.

The resulting two clauses are the following:

\[
\begin{align*}
14. new1([X|Xs]) & \leftarrow 1 \leq X \land new2(Xs) \\
15. new1([X|Xs]) & \leftarrow 1 > X \land new1(Xs)
\end{align*}
\]
They are mutually exclusive because of the constraints $1 \leq X$ and $1 > X$. Now the program transformation process continues in a way similar to the one we have followed above, when deriving clauses 16 and 17 starting from clause 8. By starting from clauses 14 and 15, the new predicates new3 through new6 are introduced, and by application of the unfold and fold rules, their defining clauses are derived. Eventually, the following deterministic, specialized program is obtained:

3. Program synthesis

Program synthesis is a technique for deriving programs from formal specifications (see, for instance [13, 26] for the derivation of logic programs from first-order logic specifications). In this section we present an example of use of program transformation for program synthesis and we derive a constraint logic program from its specification provided as a first-order formula.

We consider the $N$-queens problem, which is often used in the literature for introducing techniques such as recursion and backtracking. That problem can be described as follows: $N$ ($\geq 0$) queens are to be placed on an $N \times N$ chess board, so that no two queens attack each other, that is, they do not lie on the same row, column, or diagonal. Since no two queens should lie on the same column, the positions of the $N$ queens on the chess board can be denoted by the list $L = [1, \ldots, 1_N]$ such that, for $1 \leq k \leq N$, $i_k$ is the row where the queen on column $k$ is placed. A solution of the specification $L$ for the $N$-queens problem is given by the following first-order formula $\phi(N, L)$:

$$
\phi(N, L) = \text{nat}(N) \land \text{nat_list}(L) \land \text{length}(L, N) \land \\
\forall X (\text{member}(X, L) \rightarrow \text{in}(X, 1, N)) \land \\
\forall A, B, K, M ((1 \leq K < K < M \\
\land \text{occurs}(A, K, L) \land \text{occurs}(B, M, L))) \\
\rightarrow \text{(A} \land B \land A \land B \land M \land K \land B \land A \land M \land K))
$$

where: $\text{nat}(N)$ holds iff $N$ is a natural number, $\text{nat_list}(L)$ holds iff $L$ is a list of natural numbers, $\text{length}(L, N)$ holds iff $L$ is a list of length $N$, $\text{member}(X, L)$ holds iff $X$ is an element of the list $L$, $\text{in}(X, M, N)$ holds iff $M \leq X \leq N$, and $\text{occurs}(X, I, L)$ holds iff $X$ occurs in the list $L$ at position $I$, with $1 \leq I \leq N$, where $N$ is the length of $L$. Let $P$ be the program made out of the clauses defining the predicates $\text{nat}$, $\text{nat_list}$, $\text{member}$, $\text{in}$, and $\text{occurs}$.

Now, our goal is to synthesize a constraint logic program $R$ which computes a predicate $\text{queens}(N, L)$ such that the following Property $(\tau)$ holds:

$$
(\tau) \quad M(R) \models \text{queens}(N, L) \iff M(P) \models \phi(N, L)
$$

where $M(R)$ and $M(P)$ denote the perfect model of the program $R$ and $P$, respectively. Now we apply the technique presented in [11] starting from the formula $\text{queens}(N, L) \models \phi(N, L)$. The first step of that technique consists in applying a variant of the Lloyd-Topor transformation [19] which, starting from $\text{queens}(N, L) \models \phi(N, L)$, derives the following constraint logic program $F$:

$$
\text{queens}(N, L) \leftarrow \text{nat}(N) \land \text{nat_list}(L) \land \\
\text{length}(L, N) \land \neg \text{aux1}(L, N) \land \neg \text{aux2}(L) \land \\
\text{aux1}(L, N) \leftarrow \text{member}(X, L) \land \text{in}(X, 1, N) \land \\
\text{aux2}(L) \leftarrow 1 \leq K < K < M \land \neg (A \land B \land A \land B \land M \land K) \land \\
\text{occurs}(A, K, L) \land \text{occurs}(B, M, L)
$$

It can be shown that this variant of the Lloyd-Topor transformation preserves the perfect model semantics.

As a consequence:

$$
M(P \cup F) \models \text{queens}(N, L) \iff M(P) \models \phi(N, L)
$$

Unfortunately, the performance of the derived program $P \cup F$ is not satisfactory, because it exhibits an
inefficient generate-and-test behavior, whenever we evaluate queries by applying the usual depth-first search strategy with the left-to-right computation rule. In particular, for any query of the form \( \text{queens}(n, L) \), where \( n \) is a natural number and \( L \) is a variable list, program \( P \cup F \) first generates a value \( l \) for the list \( L \) and then tests whether or not \( l \text{not in } L \) \( \land \neg \text{aux } 1(l, n) \land \neg \text{aux } 2(l) \) holds. Note that this generate-and-test behavior may even lead to nontermination, because an infinite number of lists may be generated.

Thus, the process of program synthesis proceeds by applying the definition, unfolding, folding, and goal replacement rules (see [11] for details), with the objective of deriving a more efficient, terminating program. The result of this transformation is the following program \( R \):

\[
\text{queens}(N, L) \leftarrow \text{new}(N, L, 0)
\]
\[
\text{new}(N, [\ ], K) \leftarrow N = K
\]
\[
\text{new}(N, [H(T), K) \leftarrow N \geq K + 1
\]
\[
\land \text{new}(N, T, K + 1)
\]
\[
\land \text{new}(H, T, N, 0)
\]
\[
\text{new}(A, [\ ], N, M) \leftarrow \text{init}(A, 1, N) \land \text{nat}(A)
\]
\[
\text{new}(A, [B(T), N, M) \leftarrow A \neq B \land A \neq M + 1
\]
\[
\land \text{new}(A, T, N, M + 1)
\]

\[
\text{together with the clauses defining the predicates } in \text{ and } nat. \text{ Since the transformation rules preserve the perfect model semantics, } M(R) \models \text{queens}(N, L) \text{ iff } M(P \cup F) \models \text{queens}(N, L) \text{ and, thus, Property (R) holds. It can be shown that } R \text{ terminates for all queries of the form } \text{queens}(n, L). \text{ Note that program } R \text{ solves the } N \text{-queens problem in a clever way: each time a new queen is placed on the board, } R \text{ tests whether or not it attacks another queen already on the board.}

4. Program verification

Proof of program properties are often needed for checking the correctness of software components with respect to their specifications. This section illustrates the use of program transformation for proving program properties specified by either first-order formulas or temporal logic formulas.

Proofs performed by using program transformation are strongly related to proofs by mathematical induction (see [3] for a survey on inductive proofs). In particular, the unfolding rule can be used for decomposing a formula of the form \( \psi(t(X)) \), where \( t(X) \) is a complex term, into a combination of \( n \) formulas of the form \( \psi_1(X), \ldots, \psi_n(X) \), and the folding rule can be used for applying inductive hypotheses.

The unfold/fold transformations introduced in [4, 30] can be used for proving several kinds of program properties, such as equivalences of functions defined by recursive equation programs [16], equivalences of predicates defined by logic programs [22], first-order properties of predicates defined by constraint logic programs [23], and temporal properties of concurrent systems [9, 25].

4.1. The unfold/fold proof method

Now we present, by means of a simple example taken from [23], a technique called unfold/fold proof method which is based on program transformation and can be used for proving first-order properties of constraint logic programs.

Let us consider the following program \( \text{Member} \) which defines the membership relation for lists:

\[
\text{member}(X, [Y | L]) \leftarrow X = Y
\]
\[
\text{member}(X, [Y | L]) \leftarrow \text{member}(X, L)
\]

The following formula \( \psi \) states that every finite list of numbers has an upper bound:

\[
\psi : \forall L \exists U ( \text{member}(X, L) \rightarrow X \leq U)
\]

To show that \( \psi \) holds, we apply the unfold/fold proof method which consists of the following two steps. In the first step, \( \psi \) is transformed into a set of clauses by applying a variant of the Lloyd-Topor transformation [19], thereby deriving the following program \( \text{Prop}_1 \) which defines the predicate \( prop \):

\[
\text{prop} \leftarrow \neg p
\]
\[
p \leftarrow \text{list}(L) \land \neg q(L)
\]
\[
q(L) \leftarrow \text{list}(L) \land \neg r(L, U)
\]
\[
r(L, U) \leftarrow X > U \land \text{list}(L) \land \text{member}(X, L)
\]

The predicate \( prop \) is equivalent to \( \psi \) in the sense that

\[
M(\text{Member}) \models \psi \text{ iff } M(\text{Member } \cup \text{Prop}_1) \models prop.
\]

In the second step, the existential variables occurring in \( \text{Prop}_1 \) (that is, the variables occurring in the body of a clause and not in its head) are eliminated by applying the transformation strategy presented in [23]. As a result we get the following program \( \text{Prop}_2 \):

\[
\text{prop} \leftarrow \neg p \quad p \leftarrow p_1 \quad p_1 \leftarrow p_1
\]
Now, Prop₂ is a propositional program and its finite perfect model M(Prop₂) is \( \{ \text{prop} \} \). Since it can be shown that every transformation we have applied preserves the perfect model, we get that \( M(\text{Member}) \models \text{prop} \) iff \( M(\text{Prop}_2) \models \text{prop} \). Therefore, we have that \( \text{prop} \) holds for the program Member because prop belongs to \( M(\text{Prop}_2) \).

Note that the unfold/flatten proof method can be viewed as an instance of the quantifier elimination method which has well-known applications in the field of automated theorem proving (see [24] for a brief survey).

### 4.2. Infinite-state model checking

In this section we present a method [9] for verifying temporal properties of infinite state systems by transforming constraint logic programs.

As indicated in [5], the behavior of a concurrent system that evolves over time according to a given protocol can be modeled by means of a state transition system, that is, (i) a set \( S \) of states, (ii) an initial state \( s_0 \in S \), and (iii) a transition relation \( t \subseteq S \times S \). Let us assume that \( t \) is a total relation, that is, for every state \( s \in S \) there exists a state \( s' \in S \) called the successor state of \( s \), such that \( t(s, s') \) holds. A computation path starting from a state \( s_1 \) is an infinite sequence of states \( s_1, s_2, \ldots \) such that, for every \( i \geq 1 \), there is a transition from \( s_i \) to \( s_{i+1} \), that is, \( t(s_i, s_{i+1}) \) holds.

The properties of the evolution over time of a concurrent system are specified by using formulas of the temporal logic called Computation Tree Logic (or CTL), for short [5] which specify the properties of the computation paths. The formulas of CTL are built from a given set of elementary properties of the states by using: (i) the usual connectives: \( \neg \) (‘not’), and \( \land \) (‘and’), (ii) the following quantifiers along single computation paths: \( \exists \) (‘for all states on the path’ or ‘globally’), \( \forall \) (‘there exists a state on the path’ or ‘in the future’), \( x \) (‘next time’), and \( u \) (‘until’), and (iii) the two quantifiers over computation paths: \( a \) (‘for all paths’) and \( e \) (‘there exists a path’).

There exist very efficient algorithms and tools which use model checking techniques for verifying temporal properties of finite state systems, that is, systems where the set \( S \) of states is finite [5]. Unfortunately, many concurrent systems cannot be modeled by finite state systems and one should look for verification methods for infinite state systems. Various methods based on automated theorem proving have been proposed in the literature and, in particular, a method based on constraint logic programming is described in [6]. Note that all of the verification methods for infinite state systems are necessarily incomplete, because the problem of verifying CTL properties of those systems is in general undecidable.

As an example of use of program transformation for verifying CTL properties of infinite state systems, now we will consider the Bakery protocol [17] and we will verify that it satisfies the mutual exclusion property. Let \( A \) and \( B \) be two agents which want to access a shared resource in a mutual exclusive way by using the Bakery protocol. A state of agent \( A \) is represented by the pair \( (A_1, A_2) \), where \( A_1 \) is a control state that takes values in the set \( \{ t, w, a \} \) (where \( t, w, \) and \( a \) stand for think, wait, and use, respectively), and \( A_2 \) is a counter that takes values in the set of natural numbers. Analogously, the state of agent \( B \) is represented by a pair \( (B_1, B_2) \) of the control state \( B_1 \) and the counter \( B_2 \). Then the state of the system consisting of the two agents \( A \) and \( B \) is represented by the 4-tuple \( (A_1, A_2, B_1, B_2) \), and the transition relation \( t \) from an old state \( \text{OldS} \) to a new state \( \text{NewS} \), is defined by:

\[
\begin{align*}
\text{t(OldS, NewS)} & \leftarrow t_A(\text{OldS}, \text{NewS}) \\
\text{t(OldS, NewS)} & \leftarrow t_B(\text{OldS}, \text{NewS})
\end{align*}
\]

where the transition relation \( t_A \) for the agent \( A \) is given by the following clauses whose bodies are atomic constraints (in general, they may be conjunctions of atomic constraints) (see also Fig. 1):

\[
\begin{align*}
t_A((t, A_1, A_2, B_1, B_2), (w, A_21, B_1, B_2)) & \leftarrow A_21 = B_2 + 1 \\
t_A((w, A_1, A_2, B_1, B_2), (u, A_2, B_1, B_2)) & \leftarrow A_2 < B_2 \\
t_A((u, A_1, A_2, B_1, B_2), (u, A_2, B_1, B_2)) & \leftarrow B_2 = 0 \\
t_A((w, A_1, A_2, B_1, B_2), (t, A_21, B_1, B_2)) & \leftarrow A_21 = 0
\end{align*}
\]

Fig. 1. The Bakery protocol: a graphical representation of the transition relation \( t_A \) for agent \( A \).
In a similar way the following clauses define the transition relation \( t_B \) for the agent \( B \):

\[
\begin{align*}
& t_B((A_1, A_2, t, B_2), (A_1, A_2, w, B_21)) \leftarrow B_2 = A_2 + 1 \\
& t_B((A_1, A_2, w, B_2), (A_1, A_2, u, B_2)) \leftarrow B_2 < A_2 \\
& t_B((A_1, A_2, w, B_2), (A_1, A_2, u, B_2)) \leftarrow A_2 = 0 \\
& t_B((A_1, A_2, u, B_2), (A_1, A_2, t, B_21)) \leftarrow B_21 = 0
\end{align*}
\]

Note that the system has an infinite number of states, because counters may increase in an unbounded way.

The temporal properties of a transition system are specified by defining a predicate \( \text{sat}(S, F) \) which holds if and only if the temporal formula \( F \) holds at state \( S \). The clauses defining \( \text{sat}(S, F) \) for the cases where \( F \) is: (i) an elementary formula \( F \), (ii) a formula of the form \( \neg F \), (iii) a formula of the form \( F_1 \land F_2 \), and (iv) a formula of the form \( \sigma(F) \), are the following:

\[
\begin{align*}
& \text{sat}(S, F) \leftarrow \text{elem}(S, F) \\
& \sigma(S, \sigma(F)) \leftarrow \neg \text{sat}(S, F) \\
& \text{sat}(X, F_1 \land F_2) \leftarrow \text{sat}(X, F_1) \land \text{sat}(X, F_2) \\
& \text{sat}(S, \sigma(F)) \leftarrow \text{sat}(S, F) \\
& \text{sat}(S, \sigma(F)) \leftarrow \text{sat}(S, \sigma(F)) \land \text{sat}(T, \sigma(F))
\end{align*}
\]

where \( \text{elem}(S, F) \) holds if \( F \) is an elementary property which holds at state \( S \). In particular, for the Bakery protocol, the following clause:

\[
\text{elem}(u, A_2, u, B_2), \text{unsafe} \leftarrow
\]

encodes that \text{unsafe} holds at a state where both agents \( A \) and \( B \) are in the control state \( s \) (that is, both agents are accessing the shared resource at the same time).

In order to make program transformation even more effective in the future, it is necessary to further develop and automate the transformation strategies for program improvement, program synthesis, and program verification, and incorporate them into powerful program development tools.

5. Conclusions and future directions

This paper presents a brief overview of the program transformation methodology and illustrates its use for: (i) the derivation of correct software modules from their formal specifications, and (ii) the proof of properties of programs. Since program transformation preserves correctness and improves efficiency, it is very effective for constructing programs which are provably correct and whose performance is very high.


\[
(\mu) \quad \text{mex} \leftarrow \text{sat}((1,0,0,1), \neg \rho(\text{unsafe}))
\]

and the program \( P_{\text{max}} \cup \{ \text{mex} \} \) is transformed into a new program \( \bar{Q} \) which contains a clause with empty body of the form: \( \rho \) (see [9] for details). This transformation is performed by using the definition, unfolding, and folding rules according to the specialization strategy [12, 15, 18] that in our case derives the clauses specialized to the evaluation of the predicate \( \text{mex} \). By the correctness of the rules, \( \text{mex} \in M(\bar{Q}) \) iff \( \text{mex} \in M(P_{\text{max}} \cup \{ \text{mex} \}) \) and, hence, \( \text{sat}((1,0,0,1), \neg \rho(\text{unsafe})) \in M(P_{\text{max}}) \), that is, the mutual exclusion property holds, because \( \text{mex} \) is a clause of \( Q \).

The derivation needed for verifying the mutual exclusion property was performed in a fully automatic way by using our experimental constraint logic program transformation system MAP [20].

5.5 Conclusions and future directions

This paper presents a brief overview of the program transformation methodology and illustrates its use for: (i) the derivation of correct software modules from their formal specifications, and (ii) the proof of properties of programs. Since program transformation preserves correctness and improves efficiency, it is very effective for constructing programs which are provably correct and whose performance is very high.

In order to make program transformation even more effective in the future, it is necessary to further develop and automate the transformation strategies for program improvement, program synthesis, and program verification, and incorporate them into powerful program development tools.

Acknowledgements

Many thanks to the guest editors of the Intelligence Artificialle Journal for their kind invitation to write a paper in honor of Alberto Martelli. His paper [21] taught us how to write algorithms in a structural way.
References


