Development of Correct Transformation Schemata for Prolog Programs

Julian Richardson[1]* and Norbert Fuchs[2]

[1] Department of Artificial Intelligence, Edinburgh University, 80 South Bridge, Edinburgh EH1 1HN, Scotland
julianr@dai.ed.ac.uk

[2] Department of Computer Science, University of Zurich, CH-8057 Zurich, Switzerland
fuchs@ifi.unizh.ch

Abstract. Schema-based program transformation [8] has been proposed as an effective technique for the optimisation of logic programs. Schemata are applied to a logic program, mapping inefficient constructs to more efficient ones. One challenging aspect of the technique is that of proving that the schemata are correct.

This paper addresses the issue of correctness. We define operations for developing correct schemata by construction. The schema development operations are higher order equivalents of the classic program transformations of fold/unfold [6]. We consider a transformation schema to be correct if its application yields a target program which is equivalent to the source program under the *pure Prolog semantics*.

The work described in this paper makes three contributions: a methodology for the development of provably correct program transformation schemata, abstraction of program transformation operations to transformation operations on schemata, and a higher-order unification algorithm which forms the basis of the schema transformation operations.

1 Schema-Based Transformations

A program transformation technique based on transformation *schemata* is described in [8]. Transformation schemata are defined using patterns — higher-order terms which can be instantiated to program fragments. A transformation schema is applied to a program by scanning the program for a piece of code which matches the source pattern, and replacing it with the instantiated target pattern.

* The first author is supported by EPSRC Grant GR/L11724. Most of the work described was performed while the first author was a visiting research fellow at the University of Zurich, supported by HCM-Network "Logic Program Synthesis and Transformation", contract no. CHRX-CT93-0414, Project BBW 93.0268.

A program transformation schema is defined in [8] as a 4-tuple, which specifies that a conjunction of goals $G_1, ..., G_n$ with corresponding predicate definitions $S_1, ..., S_n$ can be transformed into a conjunction of goals $H_1, ..., H_n$ with corresponding predicate definitions $T_1, ..., T_n$:

$$\langle\langle G_1, ..., G_n\rangle, \langle S_1, ..., S_n\rangle, \langle H_1, ..., H_n\rangle, \langle T_1, ..., T_n\rangle\rangle \tag{1}$$

Such schemata can encode a wide range of useful transformations, e.g. loop fusion, accumulator introduction and goal reordering.

This paper addresses the issue of proving the correctness of transformation schemata. There are two basic design decisions we must make when considering the correctness of program transformation schemata:

1. Do we seek to prove the correctness of existing schemata, or do we instead only provide tools for constructing new schemata which are guaranteed to be correct? We have decided to take the latter approach.
2. How do we define the correctness of a program transformation? This question is discussed in the next section.

The paper is organised as follows. First we discuss related work in §2. We define equivalence of programs in §3 relative to the pure Prolog semantics, then outline in §4 how correct transformation schemata can be constructed incrementally by the application of abstract transformation operations, which are the equivalents on program patterns of the classic fold/unfold transformations on programs. We define a language for expressing program patterns and transformation schemata in §5, and outline a unification algorithm for program patterns in §6. The abstract transformation operations are described in §7, concentrating on the development of a correct unfold operation. We discuss the representation of transformation correctness conditions (§8) and the progress of our implementation (§9), before outlining how predicate termination information could be used (§10). Finally, we discuss further work (§12) and draw our conclusions (§13). Appendix A presents the unification algorithm, and appendix B goes through an example schema development using schema folding and unfolding operations.

2 Related Work

Research on schemata is a very active field. Of the many papers that have been published in the last years we will focus on three that describe recent developments.

After discussing the advantages and disadvantages of various schema languages, [4] introduce a new language to represent program schemata based on a subset of second-order logic, enhanced with specific schema features, and with global and local constraints. Their language is a variant of the language proposed by [8], though with the essential difference that constraints are not part of the schemata but are made explicit in a first-order language. The introduction of explicit constraints not only increases the expressiveness of the language but also guides the matching of the schema with a program through the successive application of rewriting and reduction rules. Starting with a schema S, a set of initial constraints C, and a program P, the pair $\langle S = P, C\rangle$ is successively rewritten and reduced to $\langle \emptyset, C'\rangle$. All occurring constraint sets are consistent. Then there is a substitution $\theta \in C'$ so that $S\theta = P$ and θ satisfies the initial constraint set C.

While most researchers — including [4] and ourselves — represent schemata purely syntactically as first- or second-order expressions, [5] express schemata as full first-order theories called specification frameworks. A specification framework axiomatises an application domain. It contains an open — i.e. only partially defined — program that represents the schema itself. The authors state that the main advantage of their approach is that it simplifies the semantics of schemata since the specification framework can be given a model-theoretic semantics in the form of reachable isoinitial models. Based on this semantics the authors define a notion of correctness for schemata. Correct schemata are expressed as parametric specification frameworks that contain steadfast open programs, where steadfast means that the programs are always correct provided their open relations are computed correctly. Based on the notion of correct program schemata one can synthesise steadfast open programs which are not only correct but also reusable.

The authors of [3] base their work on the concepts of specification-frameworks and steadfastness suggested by [5]. To extend these concepts to be used for program transformations they introduce the additional concept of the equivalence of the open — i.e. partially defined — programs computing the same relation. Then they define a transformation schema as a 5-tuple $\langle S_1, S_2, A, O_{12}, O_{21}\rangle$ where S_1 and S_2 are program schemata, i.e. specification frameworks, A an applicability condition ensuring the equivalence of the open programs within S_1 and S_2 with respect to the top-level relation computed, and O_{12} (resp. O_{21}) a set of optimisability conditions which ensure the optimisability of S_2 (resp. S_1). The authors present some concrete transformation schemata that they have implemented: divide-and-conquer, tupling generalisation, descending generalisation, and duality laws. The authors evaluate these transformation schemata with a small number of performance tests. The results show that the transformations cannot be used blindly thus necessitating the above mentioned optimisability conditions.

3 Choice of Semantics

A transformation is defined to be totally correct if the source and target programs are equivalent under the chosen semantics. We must choose which semantics to use.

The correctness of transformations of Prolog programs under a variety of semantics is discussed in [6]. Much of the existing work on the transformation of Prolog programs only considers the simplest semantics, the Herbrand semantics. This permits many powerful and interesting transformations. Unfortunately, if these transformations were let loose on a real Prolog program, they would wreak havoc, because the Herbrand semantics does not capture the intended meaning of

Prolog programs as they are written in practice. Schema-based transformations have, from the outset, been intended as a practical tool for program transformation, so we have decided to use a realistic semantics, the so-called *pure Prolog semantics* so that the resulting transformations can be applied to real programs. The pure Prolog semantics accurately reflects the semantics of Prolog programs when executed under the standard SLD resolution strategy. Programs must be cut-free and negation-free, and it is assumed that Prolog unification includes an occurs check.

4 Correctness by Construction

Many of the program transformations which are correct under the Herbrand semantics, e.g. unfold, are only correct under the pure Prolog semantics if certain conditions hold. For example, a literal (other than the first one) can only be unfolded in a clause if it is "non-left propagating". Generally, conditions such as this can only be established when the transformation schema is applied to a concrete program. We therefore modify the presentation of schemata given in [8] (as the authors of [8] suggest) by extending schemata with correctness conditions. The goal conjunctions $G_1, ...G_n$ and $H_1, ..., H_n$ which provide the transformation context in (1) can be provided by adding a predicate with body $G_1, ..., G_n$ to the source program, so we omit them in this presentation.

The state of development of a schema transformation is represented by a tuple (2), where *Schema* is a list of labeled clauses as described in §5, *Op* is the schema transformation operation which was applied to bring us to this state, Φ is a set of conditions which must be satisfied when the schema is applied to a Prolog program to ensure correctness of the resulting transformation, and *PreviousHistory* is another history structure.

$$history(Schema, Op, \Phi, PreviousHistory). \tag{2}$$

Correct schemata are developed by successive application of the available transformation operations, which are defined in such a way that when the resulting schema is applied to a program, there is a correspondence between the abstract operations which were used to develop that schema, and transformation operations on the source program yielding the target program.

The top line of figure 1 depicts a complete schema development. The figure as a whole illustrates how matching a schema pattern to a program fragment induces a matching process from the transformation operations and correctness conditions on schema patterns to corresponding transformation operations and correctness conditions on programs. The correctness conditions, Φ_i are accumulated during the development process, so that $\Phi_{k+1} \rightarrow \Phi_1 \wedge ... \wedge \Phi_k$. Once development has been completed, the intermediate stages can be stripped away, leaving just $Schema_0$, Φ_n and $Schema_n$. In principle, such a development could perform the transformation of a number of predicates. If the predicates which are transformed have heads $P_1, ..., P_k$ in $Schema_0$, and $P'_1, ..., P'_k$ in $Schema_n$,

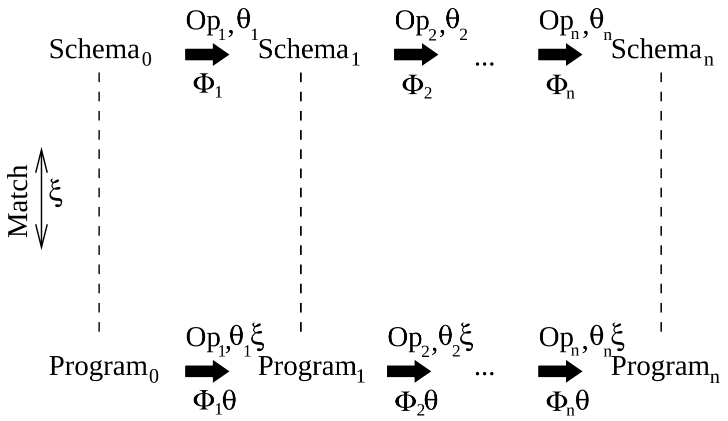

Fig. 1. A schema development. Each horizontal arrow is labeled with an operation together with its associated substitution and correctness conditions.

then we could express the resulting transformation schema in the language of Fuchs and Vasconcelos [8] as a tuple:

$$\langle\langle P_1, ..., P_k\rangle, Schema_0, \langle P'_1, ..., P'_k\rangle, Schema_n, \Phi_n\rangle$$

Note that some of the schema development operations $Schema_k \overset{Op}{\Rightarrow} Schema_{k+1}$ may modify $Schema_k$ in addition to $Schema_{k+1}$. If such an operation is applied, then we can only define a transformation schema from $Schema_k$ to $Schema_n$, and we say that the source pattern has been reset.

5 The Schema Language

The starting point for a schema development is a list of clauses. Each clause is represented internally as a term *label* : *clause*(*Head*, *Tail*), which is portrayed as: *label* : *Head* $\leftarrow$ *Tail*. We distinguish the following kinds of term:

1. Object-level variables *var*(*Name*), portrayed form "Name".
2. Vector variables *vec*(*Type*, *Length*, *Name*). *Type* is the atom '*G*' to indicate that this is a vector of goals, or '*A*' to indicate that it is a vector of arguments. *Length* is either a positive integer or zero or a Prolog variable. *Name* is either an atom or an integer. The portrayed form is TypeName:Length, so for example *vec*(*'G'*, _, *x*) is portrayed as "Gx:_".
3. Predicates *pred*(*P*, *Args*), where *P* is an atom or a variable *var*(*Name*) and *Args* is a list of arguments. The portrayed form is "P(Args)".
4. Function applications *apply*(*F*, *Args*), where *F* is an atom or a variable *var*(*Name*) and*Args* is a list of arguments. The portrayed form is "F(Args)".

5. Lists of arguments or goals. The elements of the list can either be predicates or function applications, as appropriate for the type of the list, or vectors of the appropriate type.

6. Clauses $Head \leftarrow Tail$. $Head$ must be a single goal $pred(Name, Args)$. $Tail$ is a (possibly empty) list of goals or goal vectors.
7. Sequences of clauses $\underline{\overline{\chi}}$.

The language defined above is close to that defined in [8], except that we do not require the notation $x\#n$ (used in [8] to mean that x occurs as the n^{th} argument of a predicate or function). The facility to specify the lengths of vectors can be used to perform the same function, e.g. $f(\overline{L}, X\#n, \overline{R})$ is replaced by $f(vec(\text{‘}A\text{’}, n-1, L), X, vec(\text{‘}A\text{’}, \_, R))$. In addition, we define a notation ($\underline{\overline{\chi}}$) for representing sequences of clauses, which allows us to refer during schema development to the context of the predicates to be transformed.

6 Unification of Schemata

Unification is the essential component of the Prolog execution strategy. A higher-order matching algorithm is also an essential component of the application of schemata to programs, during which patterns in the schema must be matched to parts of the Prolog program which are to be transformed. When we are constructing correct transformation schemata, unification is again essential, to allow us to apply abstract versions of the familiar program transformation steps of fold, unfold etc. This unification is an extension of the matching used when applying schemata to programs.

In the next section, we outline a higher-order unification algorithm for program patterns. The generality of this algorithm allows us to define very powerful equivalents on program patterns of the fold and unfold program transformations, but this generality also means that we have to be quite careful how we define them.

6.1 There Is No Single Most General Unifier

The principal problem is of how to unify two terms which consist of a mixture of goal (or argument) vectors and predicates (or functions). If we consider vectors of goals to be lists, and single goals to be singleton lists, then the separating comma can be considered as the list append operation. Unification of goal patterns then involves unification of lists modulo the associative append operation. Associative unification is discussed in [1, p309]. A decision procedure exists to determine whether two terms are unifiable, but there may be an infinite number of most general unifiers (mgus). For example, $p(\overline{X}), q(\overline{X}, a)$ and $p(\overline{Y}), q(a, \overline{Y})$ unify producing an infinite sequence of substitutions:[1]

$$\{a/\overline{X}, a/\overline{Y}\}, \{(a,a)/\overline{X}, (a,a)/\overline{Y}\}, ..., \{a^n/\overline{X}, a^n/\overline{Y}\}, ...$$

Usually, however, we expect only a finite set of most general unifiers, and may raise an error condition if there are too many unifiers to process. Often, we only

[1] Substitutions are oriented such that applying a substitution $\{X/Y\}$ to Y yields X.

need the first unifier which is returned by the algorithm, and need not consider other possible unifiers. When this is not the case, we may be able to rank the unifiers according to some measure and only use the best one. We will come back to the problem of multiple unifiers later.

6.2 The Unification Algorithm

Unification proceeds left-to-right. We allow terms in which function and predicate symbols may be variables. We have implemented such a unification algorithm, which is presented in appendix A. As an example, consider the problem of unifying $\overline{G}_2, Q(X), \overline{G}_3$ with $\overline{H}_2, Q(Y), \overline{H}_3$, given the substitution σ. There are four possible distinct ways to partition the goals and vectors between the two terms:

1. $\overline{H}_2 = \overline{G}_2$. Add the substitution $\{\overline{G}_2/\overline{H}_2\}$ to σ, and try to unify $Q(X), \overline{G}_3$ with $Q(Y), \overline{H}_3$.
2. $\overline{H}_2 = \overline{G}_2, \overline{D}_2$ where $\overline{D}_2$ is nonempty. Add the substitution $\{(\overline{G}_2, \overline{D}_2)/\overline{H}_2\}$ to σ, and try to unify $Q(X), \overline{G}_3$ with $\overline{D}_2, Q(Y), \overline{H}_3$.
3. $\overline{G}_2 = \overline{H}_2, \overline{D}_2$ where $\overline{D}_2$ is nonempty. $\{(\overline{H}_2, \overline{D}_2)/\overline{G}_2\}$ to σ, and try to unify $\overline{D}_2, Q(X), \overline{G}_3$ with $Q(Y), \overline{H}_3$.
4. Otherwise. The two terms do not unify. This branch of unification fails, and the algorithm backtracks.

In the example above, there are 5 unifiers: $(\overline{H}_2, Q(Y), \overline{H}_3)$, $(\overline{H}_2, Q(Y), Q(X), \overline{G}_3)$, $(\overline{H}_2, Q(Y), \overline{D}_3, Q(X), \overline{G}_3)$, $(\overline{G}_2, Q(X), Q(Y), \overline{H}_3)$ and $(\overline{G}_2, Q(X), \overline{D}_3, Q(Y), \overline{H}_3)$.

7 Transformation Operations on Program Patterns

Theorem 14 of [6] describes how the transformation rules of leftmost unfolding, deterministic non-left propagating unfolding, Tamaki-Sato folding, Tamaki-Sato definition and definition elimination can be used correctly to transform a definite-clause program under the pure Prolog semantics. In this section we show how to define versions of these operations on program schemata. We concentrate on the fold and unfold, only outlining the case for the other transformations. The example of appendix B uses these operations to develop an example transformation schema.

7.1 Schema Initialisation, Clause Addition, Specialisation

Schema development starts with a completely blank program pattern. Three operations are provided whose main purpose is to introduce sufficient structure into the program pattern to give the operations of folding and unfolding something to work with. These operations are: initialisation, clause addition, and specialisation. *Initialisation* throws away any current schema development and starts again from the blank pattern. *Clause addition* appends a new clause pattern $H \leftarrow \overline{T}$ to the program pattern. There are two types of *clause specialisation*:

pattern instantiation unifies a chosen subterm of a clause pattern with a user-specified pattern term, and division splits a chosen argument or goal vector in a clause into two.

Although each of these operations can be applied at any time during development, applying one of them to a program pattern P to give a new program pattern P' resets the source program pattern to P'. It is therefore preferable to apply them only at the beginning of a schema development.

7.2 Abstract Unfold

Care Is Needed with Unification

We would like the abstract unfold operation on program patterns to mirror the concrete unfold operation on programs, as described in [6, p.284]. At first sight, we can say that to apply an abstract unfold operation to an atom A in an abstract clause, $c : H \leftarrow \overline{L}, A, \overline{R}$ using abstract clauses $u_1 : H_1 \leftarrow \overline{T}_1, ..., u_n : H_n \leftarrow \overline{T}_n$, we replace c by n clauses $c_1, ..., c_n$ defined by $c_i : H\theta_i \leftarrow (\overline{L}, \overline{T}_i, \overline{R})\theta_i$ where for each c_i, θ_i is a most general unifier such that $A\theta_i = H_i\theta_i$, plus abstract versions of the concrete unfold correctness conditions, in particular that the unfold be *non-left propagating*.[2] This means that the atoms to the left of the $\overline{T}_i$ in the new clauses c_i must be variants of (i.e. identical up to renaming) the atoms to the left of A in c.

Unfortunately, the definition above has several problems. As noted in §6.1, there may be more than one mgu for each unification problem. Each such mgu makes certain commitments in the transformed program. We cannot use *all* the mgus, since they are not in general disjoint, and so doing would introduce repeated solutions. Therefore we must pick one mgu for each unification.

Picking one unifier (and hence one set of commitments) from several possibilities means that the resulting transformation schema may be a program *specialisation*, not an equivalence. In order to prevent specialisation of the transformed program, we must apply the unifier to both the source and the target pattern. How can we do this when there are multiple clauses with which to unfold? The choice of unifier we make when unfolding with each of the clauses will generally be incompatible. For example, when unfolding an atom $p(\overline{A}, K, \overline{B})$ using a clause with head $p(\overline{W}, cons(H, T), \overline{Z})$ we are faced with a choice of unifiers like those in §6.2.

The solution we adopt here to ensure that we do not pick incompatible unifiers for the unfolding clauses is to generalise the heads of the unfolding clauses u_i so that they are identical, and insert appropriate equality atoms explicitly into the bodies of the clauses. This allows us to pick a single unifier for all the u_i.

[2] As the example of [6][Example 14] shows, left-propagating unfolds can change the order of the returned answer substitutions, so we must disallow them.

The Abstract Unfold Operation

For an example of the application of the unfold operation defined in this section, see appendix B. To resolve the problems outlined above, we unfold a clause $c : P \leftarrow \overline{L}, A, \overline{R}$ using clauses $u_1 : H_1 \leftarrow \overline{T}_1, ..., u_n : H_n \leftarrow \overline{T}_n$, in the following steps:

1. Generalise the H_i to $\mathcal{H}$ such that the arguments of $\mathcal{H}$ are variables or vectors. Let θ_i be an assignment of values to these arguments so that $\mathcal{H}\theta_i = H_i$. In order to ensure that each θ_i is unique, the unification is made deterministic by disabling rules (4) and (5) of Appendix A, which disallows substitutions of the form $\{()/\mathit{vector}\}$ or $\{(\mathit{vector}, \mathit{vector})/\mathit{vector}\}$. Each substitution θ_i is expressed as a list of equations[3] $X^i_j = E^i_j$.
2. Find an mgu Θ such that $A\Theta = \mathcal{H}\Theta$. This is the only point in the unfold step at which we must make a choice of unifier, and making this choice necessitates the next step.
3. In the source, replace clause c with $c' : P\Theta \leftarrow \overline{L}\Theta, A\Theta, \overline{R}\Theta$. In certain special cases, this may not be necessary. See the explanatory note in the paragraph at the end of this section.
4. In the target, replace clause c with n clauses:
$$c_i : P\Theta \leftarrow (X^i_1 = E^i_1, ..., X^i_{k_i} = E^i_{k_i})\Theta, \overline{L}\Theta, \overline{T}_i\Theta, \overline{R}\Theta$$
5. For each c_i, eliminate any intermediate variables and vectors which were introduced by the unification algorithm, i.e. were not present in c or $u_1, ..., u_n$.

In accordance with [6, p.284], we require the following conditions to hold:

1. $u_1, ..., u_n$ constitute one entire predicate definition — the program to be transformed contains no clauses apart from $u_1, ..., u_n$ which define the same predicate, and these are all the clauses which define the predicate.
2. In addition, either:
   (a) $\overline{L} = ()$ — the unfold is a leftmost unfold, or
   (b) the unfold is non-left propagating. This condition can only be checked when the pattern has been instantiated by matching with a program fragment, so for each c_i, a term *non_left_propagating(c,A,c_i)* is added to the set Φ of transformation conditions.

By applying the unifier Θ to the source pattern as well as to the target, we ensure that the resulting schema does not specialise the program to which it is applied. This disturbs the development history, and means that we must essentially start a new development with the new source pattern as the initial pattern.

[3] Note that each of the unifying substitutions θ_i must be converted to a list of equations. The equations are unoriented, and this allows us to compose the θ_i with the mgu θ, since applying θ to an equation $X = Y$ yields a new equation (which is later solved by unification in step (5) above) $X\theta = Y\theta$, whereas applying θ to a substitution $\{X/Y\}$ may not yield a valid substitution, e.g. when $\theta = \{f(a)/Y\}$.

However, in the special case where Θ is only a renaming of c, no modification of the source pattern is necessary, allowing the development history to be retained. In the case that there is more than one possible unifier Θ, it is preferable to choose one which is a renaming where this is possible — this can be achieved by modifying the pattern unification algorithm to return renamings before other unifiers. In order to encourage the existence of renaming unifiers, it is important to try to develop enough structure in the pattern using the clause addition and specialisation operations before applying unfolding.

7.3 Abstract Fold

With folding, as with unfolding, we must take care not to specialise the transformed program. An example of the development of a transformation schema for loop fusion using folding and unfolding is contained in appendix B.

The abstract fold definition mirrors that of Tamaki-Sato folding (definition R3 of [6]). There are a number of conditions which need to be checked to ensure that the fold is a correctness-preserving transformation:

1. The folding atoms must be fold-allowing as described in Theorem 14 and Definition 7 of [6]. In order to keep track of which atoms are fold-allowing and which are not we must allow vectors to be marked with their fold-allowing status ($true$ or $false$), and ensure that any unification which splits a vector into a number of vectors or atoms marks them with the same fold-allowing status as their parent.
2. As stated in [6], there is also a restriction on the substitution which unifies the body of the folding predicate with the folded atoms; suppose we are folding a number of atoms $\overline{E}$ in the body of a clause $H \leftarrow \overline{L}, \overline{E}, \overline{R}$ using the body of a clause $H' \leftarrow \overline{G}$. If θ is the unifying substitution, i.e. $\overline{E}\theta = \overline{G}\theta$, then θ restricted to the set $vars(\overline{G}) - vars(H')$ is a variable renaming whose image has an empty intersection with the set $vars(H, \overline{L}, H'\theta, \overline{R})$. In general, it is not possible to decide this until the schema is matched with a program, but there is a significant special case when $vars(\overline{G}) - vars(H') = \{\}$ which arises when all the vectors and object-level variables of the folding predicate's body, $\overline{G}$, also occur in the folding predicate's head, H'. In this special case, the condition is trivially satisfied (clause $c6$ of Appendix B falls into this special case, for example).
3. Folding generally requires that some states in the schema development will simultaneously contain both old and new predicates, and the two types of clause must be carefully distinguished to ensure correctness is guaranteed.

7.4 Definition Introduction

In order to develop powerful transformations such as loop fusion, it is necessary to be able to apply the *definition introduction* rule. We use the Tamaki-Sato definition rule described in [6, R15], which allows a new clause to be introduced as long as its head is a predicate which does not already occur in the program, and the literals in its body are made from predicates which are already in the program.

8 Transformation Correctness Conditions

Application of a schema development operation entails checking certain correctness conditions. These conditions can be partially checked while the schema development is taking place, but there is always a residual condition which can only be checked when the schema is applied to a program. For example, suppose we need to ensure that a certain clause is not recursive. This means checking that there is no goal in the body of the predicate with the same predicate (and arity) as the head of the predicate. For the following clause, it is easy to verify that this condition does not hold:

$$P(X) \leftarrow \overline{G}, P(Y), \overline{H}$$

If, however, we replace $P(Y)$ in the body by $Q(Y)$, we cannot be certain. We cannot say immediately that the condition does not hold, but there is a residual which must be checked whenever the resulting transformation schema is applied to a program, namely that P is not equal to Q, and that P does not appear in $\overline{G}$ or $\overline{H}$.

The way in which the correctness conditions are expressed will be an important factor in ensuring the correctness of a schema development system. For example, for each schema development operation there could be a corresponding Prolog predicate. This simplifies the implementation of the correctness conditions (they are just pieces of Prolog code), but is unsatisfactory for several reasons:

1. ensuring the correctness of the schema development system is then made difficult, because it depends on the correctness of these checking predicates, and
2. it is difficult to perform reasoning about Prolog predicates.

The preferable alternative is to express these conditions using logical formulae containing a small number of primitives such as $subterm(X, T)$ — X is a subterm of T, $vars(T, V)$ — V is the set of variables in term T etc., connected by logical connectives (universal and existential quantifiers, conjunction and negation). This not only permits a high degree of confidence in the correctness conditions, but the finer grain and logical nature of such a language also makes it possible to perform some reasoning with the conditions.

9 Implementation

The unification algorithm, and simplified versions of the fold and unfold operations have been implemented in Sicstus Prolog. Some correctness conditions are enforced, but many more need to be added. A text-based user interface allows the user to apply the abstract schema development operations discussed in this paper, and allows schema developments to be loaded and saved. Output of pattern terms is easily achieved with suitable definitions of the `portray/1` predicate. Input is more tricky, since we would like to be able to input higher-order terms in a natural way. For example, the term $P(\overline{L}, A, \overline{R})$ is represented internally as `pred(var(p),[vec('A',_,l),var(a),vec('A',_,r)])`, and both portrayed and input as `P(Al:_,A,Ar:_)`. This is achieved by reading input into a string, which is then parsed by a DCG. Vector lengths are represented by Prolog variables, and Prolog handles their unification.

10 Exploiting Predicate Termination Information

One of the problems we encounter when constructing transformations which are correct under the pure Prolog semantics is that we must take care to preserve the termination behaviour of predicates. This means that we cannot simply prune a clause such as (3), because if G_1 or G_2 does not terminate, then nor will c, whereas the pruned version (in which the entire body is replaced by "fail") fails finitely.

$$c : \mathit{Head} \leftarrow G_1, G_2, 1 = 2, G_3. \tag{3}$$

This can be overcome either by allowing the transformation of infinite to finite failure, or by exploiting the termination properties of predicates when the transformation schema is applied. It is expected that many useful transformation schemata will require information on the termination properties of the predicates to which they are applied.

The termination properties of some predicates are already known, for example arithmetic goals, unification and test predicates. More generally, determining termination properties of predicates is a problem which has already been addressed by several researchers. For example termination properties are heavily used in the Mercury system [7]. The simplest approach is to allow the user to annotate predicate definitions to indicate termination properties. Termination may be established only for certain modes of a predicate. Modal inference and annotation would therefore be another useful extension.

11 Laws

The application of laws is essential to many transformation techniques. For example, the associativity of append is necessary for the transformation of naïve reverse into the tail-recursive version. Laws can be represented using program patterns. For example, associativity can be represented as below, and declaring a predicate to be associative corresponds to a particular instantiation of P:

$$P(A, B, T), P(T, C, E) \equiv P(B, C, V), P(A, V, E).$$

12 Further Work

There are many directions for further work. First and foremost, it is necessary to prove the correctness of the abstracted unfold operation, fully define and prove correctness conditions for the abstracted fold operation, define a flexible mechanism for applying laws, and extend the implementation accordingly. Following this, further schema transformation operations can be developed, for example a pruning operation, as described in §10.

Vector lengths can be used very effectively to reduce the number of solutions returned when two patterns are unified. Currently only equality relationships between vector lengths can be expressed. In particular, if a vector $\overline{V}{:}L_1$ is unified with a pattern $\overline{X}_1{:}M_1, \overline{X}_2{:}M_2$, we cannot express the fact that $L_1 = M_1 + M_2$. In [4], vector lengths can be constrained using $=, =<$ inequalities, but in the light of the above example, we could go further and extend the system to represent and solve such constraints, which are problems in Presburger arithmetic and therefore decidable.

It may be useful to introduce some automatic guidance into a schema development system, which may suggest strategies such as generalisation or tupling when appropriate. Proof plans [2] may be suitable for this. Indeed, we can view the schema development operations as tactics, in which case it is natural that proof plans should provide the meta-level.

It is also important to study the expressiveness of the resulting transformation schemata. The lack of induction in schema development operations is likely to be one source of these restrictions.

13 Conclusions

In this document we have proposed a technique for developing correct transformation schemata for Prolog programs. Correct transformation schemata are constructed by applying a sequence of development operations. A transformation is correct if when it is applied to a program, it yields a new program which is equivalent to the original one under the pure Prolog semantics. This means the program must be negation and cut-free.

The system is based on a higher-order unification algorithm for schema terms. The schema development operations, which are abstract equivalents of the classical fold/unfold etc. transformation operations, are defined in terms of this unification, and conditions are defined to ensure their correctness. These conditions can be partially checked during schema development, but generally leave a residual which can only be checked when the schema is applied to a program.

We have implemented the higher-order unification algorithm, and a simplified version of some of the transformations.

14 Acknowledgements

We would like to thank the respective funding bodies for their support, and the members of the IFI for their generous hospitality to the first author, in particular

Norbert Fuchs, Rolf Schwitter, Raja Dravid and Alex Riegler. We are grateful to the referees for their comments.

References

1. Alan Bundy. *The Computer Modelling of Mathematical Reasoning.* Academic Press, 1983. Second Edition.
2. Alan Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *9th Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988. Longer version available from Edinburgh as DAI Research Paper No. 349.
3. H. Büyükyıldız and P. Flener. Generalized logic program transformation schemas. In N. E. Fuchs, editor, *LOPSTR '97: Proceedings of the Seventh International Workshop on Logic Program Synthesis and Transformation, Leuven, Belgium, July 10-12 1997 (this volume).* Lecture Notes in Computer Science, Springer Verlag, forthcoming, 1998.
4. E. Chasseur and Y. Deville. Logic program schemas, constraints and semi-unification. In N. E. Fuchs, editor, *LOPSTR '97: Proceedings of the Seventh International Workshop on Logic Program Synthesis and Transformation, Leuven, Belgium, July 10-12 1997 (this volume).* Lecture Notes in Computer Science, Springer Verlag, forthcoming, 1998.
5. P. Flener, K.-K. Lau, and M. Ornaghi. On correct program schemas. In N. E. Fuchs, editor, *LOPSTR '97: Proceedings of the Seventh International Workshop on Logic Program Synthesis and Transformation, Leuven, Belgium, July 10-12 1997 (this volume).* Lecture Notes in Computer Science, Springer Verlag, forthcoming, 1998.
6. A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming*, 19/20:261–320, 1994.
7. Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1):17–64, October 1996.
8. W. W. Vasconcelos and N.E. Fuchs. An opportunistic approach for logic program analysis and optimisation using enhanced schema-based transformations. In *Proceedings of LoPSTr'95, Fifth International Workshop on Logic Program Synthesis and Transformation, Utrecht, Netherlands*, volume 1048 of *Lecture Notes in Computer Science*, pages 175–188. Springer Verlag, 1996.

A The Schema Unification Algorithm

In the following presentation, Var denotes an object-level variable, $\overline{V}{:}l$ represents a vector with length l, $Atom$ denotes an atom, $P(\overline{A})$ represents a function or predicate with head P and arguments $\overline{A}$. For brevity, it is assumed in this presentation that atoms, predicates, functions or object-level variables can be freely converted to vectors of length 1.

$$\frac{B =_\sigma A}{A =_\sigma B}$$

$$\frac{P =_{\sigma_1} Q,\ \overline{A}_1\sigma_1 =_\sigma \overline{A}_2\sigma_2}{P(\overline{A}_1) =_{\sigma_1 \circ \sigma} Q(\overline{A}_2)} \qquad \frac{Var \notin P(\overline{A}_1)}{P(\overline{A}_1) =_{\{P(\overline{A}_1)/Var\}} Var}$$

$$\frac{}{Atom =_{\{\}} Atom} \qquad \frac{}{Atom =_{\{Atom/Var\}} Var}$$

$$\frac{}{\overline{V}{:}l =_{\{Atom/\overline{V},\, l=1\}} Atom} \qquad \frac{}{\overline{V}{:}l =_{\{Var/\overline{V},\, l=1\}} Var}$$

$$\frac{\overline{V}{:}l \notin P(\overline{A})}{P(\overline{A}) =_{\{P(\overline{A})/\overline{V},\, l=1\}} \overline{V}{:}l} \qquad \frac{}{Var_1 =_{\{Var_1/Var_2\}} Var_2}$$

$$\frac{}{\overline{V}_1{:}l_1 =_{\{V_1/V_2,\, l_1=l_2\}} \overline{V}_2{:}l_2} \qquad \frac{l_j = 1,\ l_{i\,(i \neq j)} = 0,\ \overline{V}_j{:}1 =_\sigma P(\overline{A})}{(\overline{V}_1{:}l_1, ..., \overline{V}_k{:}l_k) =_\sigma P(\overline{A})}$$

$$\frac{l_j = 1,\ l_{i\,(i \neq j)} = 0,\ \overline{V}_j{:}1 =_\sigma Var}{(\overline{V}_1{:}l_1, ..., \overline{V}_k{:}l_k) =_\sigma Var}$$

$$\frac{\sum_{i=1}^{k} l_i = n}{(\overline{V}_1{:}l_1, ..., \overline{V}_k{:}l_k) =_{\{(\overline{V}_1:l_1,...,\overline{V}_k:l_k)/\overline{V}:n\}} \overline{V}{:}n}$$

$$\frac{\exists \overline{X}, n\,.\,(\overline{X}{:}n, \overline{V}_2{:}l_2, ..., \overline{V}_k{:}l_k) =_\sigma (\overline{W}_2{:}m_2, ..., \overline{W}_j{:}m_j),\ m_1 + n = l_1}{(\overline{V}_1{:}l_1, ..., \overline{V}_k{:}l_k) =_{\{(\overline{W},\overline{X})/\overline{V}_1\}\circ\sigma} (\overline{W}_1{:}m_1, ..., \overline{W}_j{:}m_j)} \tag{4}$$

$$\frac{(\overline{V}_2{:}l_2, ..., \overline{V}_k{:}l_k) =_\sigma (\overline{W}_1{:}m_1, ..., \overline{W}_j{:}m_j) \quad l_1 = 0}{(\overline{V}_1{:}l_1, ..., \overline{V}_k{:}l_k) =_{\{()/\overline{V}_1\}\circ\sigma} (\overline{W}_1{:}m_1, ..., \overline{W}_j{:}m_j)} \tag{5}$$

B Fold/Unfold Example

We now present a modified version of the tupling example from §2 of [6]. Vectors and atoms which are not fold-allowing are underlined.

B.1 Initial Program Schema

We start with a program *Program* with clauses as follows:

$$\begin{aligned}
&\underline{\overline{X}}_1\\
&c1 : P(\overline{X}_1) \leftarrow P_1(\overline{Y}_1, L, \overline{Y}_2), P_2(\overline{Z}_1, L, \overline{Z}_2), \overline{R}.\\
&c2 : P_1(\overline{A}_1, nil, \overline{A}_2) \leftarrow .\\
&c3 : P_1(\overline{A}_3, cons(H_1, T_1), \overline{A}_4) \leftarrow P_1(\overline{A}_5, T_1, \overline{A}_6).\\
&c4 : P_2(\overline{A}_7, nil, \overline{A}_8) \leftarrow .\\
&c5 : P_2(\overline{A}_9, cons(H_2, T_2), \overline{A}_{10}) \leftarrow P_2(\overline{A}_{11}, T_2, \overline{A}_{12}), F(\overline{A}_{13}).\\
&\underline{\overline{X}}_2
\end{aligned}$$

Note that when a schema pattern is finally matched to a fragment of Prolog program, different instances of the same schema variable name must match with Prolog-unifiable pieces of the program, so we must be careful to ensure that schema variable names are different where we do not wish this to be the case.

B.2 Schema Development

Define a new predicate with the first two atoms from the body of $c1$. The new predicate should have the same arguments as the atoms in its body, although we are free to reorder them and remove repeated occurrences as we see fit. The new predicate name must not already be in use, so $\Phi_0 = \{New \notin \{\underline{\overline{X}}_1, c1, c2, c3, c4, c5, \underline{\overline{X}}_2\}\}$. The new predicate name is instantiated when the schema is applied to a program.

$$c6 : New(L, \overline{Y}_1, \overline{Y}_2, \overline{Z}_1, \overline{Z}_2) \leftarrow \underline{P_1(\overline{Y}_1, L, \overline{Y}_2)}, \underline{P_2(\overline{Z}_1, L, \overline{Z}_2)}.$$

Fold the first two atoms of the body of $c1$ using $c6$. Since $c6$ is a new predicate, and $c1$ is old, the fold is allowed. The unifier is trivial so there is no specialisation.

$$c7 : P(\overline{X}_1) \leftarrow New(L, \overline{Y}_1, \overline{Y}_2, \overline{Z}_1, \overline{Z}_2), \overline{R}.$$

Unfold the first literal of $c6$ using $c2$ and $c3$. Since this is a leftmost unfold, it is correct as long as we follow the five unfolding steps defined in §2:

1. Generalise the heads of the unfolding clauses $(c2, c3)$:

$$\begin{aligned}
\mathcal{H} &= P_1(\overline{B}_1, M, \overline{B}_2)\ \theta_1 = \{nil/M, \overline{A}_1/\overline{B}_1, \overline{A}_2/\overline{B}_2\}\\
A &= P_1(\overline{Y}_1, L, \overline{Y}_2)\ \ \theta_2 = \{L/M, \overline{Y}_1/\overline{B}_1, \overline{Y}_2/\overline{B}_2\}
\end{aligned}$$

2. Find Θ such that $A\Theta = \mathcal{H}\Theta$. Here, we can choose $\Theta = \{\overline{B}_1 = \overline{Y}_1, M = L, \overline{B}_2 = \overline{Y}_2\}$. Other possible unifiers correspond to cases where the list L appears more than once in the head of P_1 or in the first literal of $c6$.
3. Apply Θ to the source predicate $c6$. Since Θ is a renaming, this step is trivial and the development history is undisturbed.
4. Produce the new clauses, $(c8, c9)$:

$$\begin{aligned} c8: &\mathit{New}(L, \overline{Y}_1, \overline{Y}_2, \overline{Z}_1, \overline{Z}_2) \leftarrow (M = \mathit{nil}, \overline{A}_1 = \overline{B}_1, \overline{A}_2 = \overline{B}_2), \\ &\quad \underline{P_2(\overline{Z}_1, L, \overline{Z}_2)}.\{M = L, \overline{B}_1 = \overline{Y}_1, \overline{B}_2 = \overline{Y}_2\} \\ c9: &\mathit{New}(L, \overline{Y}_1, \overline{Y}_2, \overline{Z}_1, \overline{Z}_2) \leftarrow (M = \mathit{cons}(H_1, T_1), \overline{A}_3 = \overline{B}_1, \overline{A}_4 = \overline{B}_2), \\ &\quad P_1(\overline{A}_3, T, \overline{A}_4), \underline{P_2(\overline{Z}_1, M, \overline{Z}_2)}.\{M = L, \overline{B}_1 = \overline{Y}_1, \overline{B}_2 = \overline{Y}_2\} \end{aligned}$$

5. Solve the introduced equations. This gives:

$$\begin{aligned} c8': &\mathit{New}(\mathit{nil}, \overline{Y}_1, \overline{Y}_2, \overline{Z}_1, \overline{Z}_2) \leftarrow \overline{A}_1 = \overline{Y}_1, \overline{A}_2 = \overline{Y}_2, \underline{P_2(\overline{Z}_1, \mathit{nil}, \overline{Z}_2)}. \\ c9': &\mathit{New}(\mathit{cons}(H_1, T_1), \overline{Y}_1, \overline{Y}_2, \overline{Z}_1, \overline{Z}_2) \leftarrow \overline{A}_3 = \overline{Y}_1, \overline{A}_4 = \overline{Y}_2, \\ &\quad P_1(\overline{A}_5, T, \overline{A}_6), \underline{P_2(\overline{Z}_1, \mathit{cons}(H_1, T_1), \overline{Z}_2)}. \end{aligned}$$

Now we unfold the newly introduced clauses $(c8', c9')$ using $(c4, c5)$.

First unfold $c8'$ using $(c4, c5)$:

$$\begin{aligned} \mathcal{H} &= P_2(\overline{C}_1, N, \overline{C}_2) \;\; \theta_1 = \{\overline{A}_7/\overline{C}_1, \mathit{nil}/N, \overline{A}_8/\overline{C}_2\} \\ A &= P_2(\overline{Z}_1, \mathit{nil}, \overline{Z}_2) \;\; \theta_2 = \{\overline{A}_9/\overline{C}_1, \mathit{cons}(H_2, T_2)/N, \overline{A}_{10}/\overline{C}_2\} \end{aligned}$$

Find Θ such that $A\Theta = \mathcal{H}\Theta$. One possible unifier is $\{\overline{Z}_1/\overline{C}_1, \mathit{nil}/N, \overline{Z}_2/\overline{C}_2\}$. This is a renaming of the unfolding atom, so no program modification is necessary.

Apply Θ to $c8'$ and produce the two new clauses:

$$\begin{aligned} c10: &\mathit{New}(\mathit{nil}, \overline{Y}_1, \overline{Y}_2, \overline{Z}_1, \overline{Z}_2) \leftarrow (\overline{A}_7 = \overline{C}_1, \overline{A}_8 = \overline{C}_2, N = \mathit{nil}), \overline{A}_1 = \overline{Y}_1, \\ &\quad \overline{A}_2 = \overline{Y}_2, \{\overline{Z}_1 = \overline{C}_1, \mathit{nil} = N, \overline{Z}_2 = \overline{C}_2\}. \\ c11: &\mathit{New}(\mathit{nil}, \overline{Y}_1, \overline{Y}_2, \overline{Z}_1, \overline{Z}_2) \leftarrow (\overline{A}_9 = \overline{C}_1, \mathit{cons}(H_2, T_2) = N, \overline{A}_{10} = \overline{C}_2), \\ &\quad \overline{A}_1 = \overline{Y}_1, \overline{A}_2 = \overline{Y}_2, \{\overline{Z}_1 = \overline{C}_1, \mathit{nil} = N, \overline{Z}_2 = \overline{C}_2, \} \end{aligned}$$

Clearly the body of $c11$ contains an inconsistent substitution and so is finitely failing and can be omitted from the final program.

Since $\overline{C}_1$, $\overline{C}_2$ and N do not appear in the original program schema — they were only introduced during the pattern unification process — we can eliminate them from $c10$ above to give:

$$c10': \mathit{New}(\mathit{nil}, \overline{Y}_1, \overline{Y}_2, \overline{Z}_1, \overline{Z}_2) \leftarrow \overline{Z}_1 = \overline{A}_7, \overline{Z}_2 = \overline{A}_8, \overline{A}_1 = \overline{Y}_1, \overline{A}_2 = \overline{Y}_2.$$

Next, unfold $c9'$:

$$\begin{aligned} \mathcal{H} &= P_2(\overline{C}_1, N, \overline{C}_2) & \theta_1 &= \{\overline{A}_7/\overline{C}_1, \mathit{nil}/N, \overline{A}_8/\overline{C}_2\} \\ A &= P_2(\overline{Z}_1, \mathit{cons}(H_1, T_1), \overline{Z}_2) & \theta_2 &= \{\overline{A}_9/\overline{C}_1, \mathit{cons}(H_2, T_2)/N, \overline{A}_{10}/\overline{C}_2\} \end{aligned}$$

Find Θ such that $A\Theta = \mathcal{H}\Theta$. One unifier is $\{\overline{Z}_1/\overline{C}_1, cons(H_1, T_1)/N, \overline{Z}_2/\overline{C}_2\}$. This is a renaming of the unfolding atom, so no program modification is necessary.

Apply Θ to $c9'$ and produce the two new clauses:

$$\begin{aligned} c12 : & \, New(cons(H_1,T_1), \overline{Y}_1, \overline{Y}_2, \overline{Z}_1, \overline{Z}_2) \leftarrow (\overline{A}_7 = \overline{C}_1, nil = N, \\ & \quad \overline{A}_8 = \overline{C}_2), \overline{A}_3 = \overline{Y}_1, \overline{A}_4 = \overline{Y}_2, P_1(\overline{A}_5, T_1, \overline{A}_6), \{\overline{Z}_1 = \overline{C}_1, \\ & \quad N = cons(H_1,T_1), \overline{Z}_2 = \overline{C}_2\}. \\ c13 : & \, New(cons(H_1,T_1), \overline{Y}_1, \overline{Y}_2, \overline{Z}_1, \overline{Z}_2) \leftarrow (\overline{A}_9 = \overline{C}_1, cons(H_2,T_2) = N, \\ & \quad \overline{A}_{10} = \overline{C}_2), \overline{A}_3 = \overline{Y}_1, \overline{A}_4 = \overline{Y}_2, P_1(\overline{A}_5, T_1, \overline{A}_6), P_2(\overline{A}_{11}, T_2, \overline{A}_{12}), \\ & \quad F(\overline{A}_{13}), \{\overline{Z}_1 = \overline{C}_1, N = cons(H_1,T_1), \overline{Z}_2 = \overline{C}_2\}. \end{aligned}$$

Clearly the body of $c12$ contains an inconsistent substitution and so is finitely failing and can be omitted from the final program.

As before, we can eliminate the variables $\overline{C}_1$, $\overline{C}_2$, and N which were introduced during the pattern unification process.

$$\begin{aligned} c13' : & \, New(cons(H_1,T_1), \overline{Y}_1, \overline{Y}_2, \overline{Z}_1, \overline{Z}_2) \leftarrow \overline{Z}_1 = \overline{A}_9, H_1 = H_2, T_1 = T_2, \\ & \quad \overline{Z}_2 = \overline{A}_{10}, \overline{A}_3 = \overline{Y}_1, \overline{A}_4 = \overline{Y}_2, P_1(\overline{A}_5, T_1, \overline{A}_6), P_2(\overline{A}_{11}, T_1, \overline{A}_{12}), F(\overline{A}_{13}). \end{aligned}$$

Now we fold $c13'$ using $c6'$. As noted in the second item of §7.3, we must check the unifying substitution. In general this condition can only be checked when the schema is instantiated with a Prolog program, but in this case we can easily see that $vars(bd(c6)) - vars(hd(c6)) = \{\}$. We produce the new clause:

$$\begin{aligned} c14 : & \, New(cons(H_1,T_1), \overline{Y}_1, \overline{Y}_2, \overline{Z}_1, \overline{Z}_2) \leftarrow \overline{Z}_1 = \overline{A}_9, H_1 = H_2, T_1 = T_2, \\ & \quad \overline{Z}_2 = \overline{A}_{10}, \overline{A}_3 = \overline{Y}_1, \overline{A}_4 = \overline{Y}_2, New(T_1, \overline{A}_5, \overline{A}_6, \overline{A}_{11}, \overline{A}_{12}), F(\overline{A}_{13}). \end{aligned}$$

The final program is made up from clauses $c7$, $c10'$, $c14$, $c2$, $c3$, $c4$, $c5$. By tracing through the substitutions we can eliminate the intermediate variables introduced in the presentation above to produce an equivalent list of clauses using variable names from the original schema:

$$\begin{aligned} c7 : & \, P(\overline{X}_1) \leftarrow New(L, \overline{Y}_1, \overline{Y}_2, \overline{Z}_1, \overline{Z}_2), \overline{R}. \\ c10' : & \, New(nil, \overline{Y}_1, \overline{Y}_2, \overline{Z}_1, \overline{Z}_2) \leftarrow \overline{Z}_1 = \overline{A}_7, \overline{Z}_2 = \overline{A}_8, \overline{A}_1 = \overline{Y}_1, \overline{A}_2 = \overline{Y}_2. \\ c14 : & \, New(cons(H_1,T_1), \overline{Y}_1, \overline{Y}_2, \overline{Z}_1, \overline{Z}_2) \leftarrow \overline{Z}_1 = \overline{A}_9, H_1 = H_2, T_1 = T_2, \\ & \quad \overline{Z}_2 = \overline{A}_{10}, \overline{A}_3 = \overline{Y}_1, \overline{A}_4 = \overline{Y}_2, New(T_1, \overline{A}_5, \overline{A}_6, \overline{A}_{11}, \overline{A}_{12}), F(\overline{A}_{13}). \\ c2 : & \, P_1(\overline{A}_1, nil, \overline{A}_2) \leftarrow . \\ c3 : & \, P_1(\overline{A}_3, cons(H_1,T_1), \overline{A}_4) \leftarrow P_1(\overline{A}_5, T_1, \overline{A}_6). \\ c4 : & \, P_2(\overline{A}_7, nil, \overline{A}_8) \leftarrow . \\ c5 : & \, P_2(\overline{A}_9, cons(H_2,T_2), \overline{A}_{10}) \leftarrow P_2(\overline{A}_{11}, T_2, \overline{A}_{12}), F(\overline{A}_{13}). \end{aligned}$$

The transformation schema is made up of the initial and final schema patterns. Note that the schema we have derived has some generality. For example, the use of vectors in the heads of P_1 and P_2 means that we do not require the list argument to be in any particular position.

B.3 Application of the Schema to a Program: An Example

Consider the program from [6][p.264]:

$c1 : average(L, A) \leftarrow length(L, N), sumlist(L, S), div(S, N, A).$
$c2 : length(nil, 0) \leftarrow .$
$c3 : length(cons(H1, T1), s(N1)) \leftarrow length(T1, N1).$
$c4 : sumlist(nil, 0) \leftarrow .$
$c5 : sumlist(cons(H2, T2), S1) \leftarrow sumlist(T2, S2), sum(H2, S2, S1).$

Matching the input schema with the program above gives:

$P = average$	$P_1 = length$	$P_2 = sumlist$	$\overline{A}_1 = ()$
$\overline{A}_2 = 0$	$\overline{A}_3 = ()$	$\overline{A}_4 = s(N1)$	$\overline{A}_5 = ()$
$\overline{A}_6 = N1$	$\overline{A}_7 = ()$	$\overline{A}_8 = 0$	$\overline{A}_9 = ()$
$\overline{A}_{10} = S1$	$\overline{A}_{11} = ()$	$\overline{A}_{12} = S2$	$\overline{A}_{13} = (H2, S2, S1)$
$\overline{R} = div(S, N, A)$	$\overline{X}_1 = (L, A)$	$\overline{Y}_1 = ()$	$\overline{Y}_2 = N$
$\overline{Z}_1 = ()$	$\overline{Z}_2 = S$	$L = L$	$H_1 = H1$
$H_2 = H2$	$T_1 = T1$	$T_2 = T2$	

Applying this substitution (omitting trivial variable assignments) to the final program schema gives the following program, as expected:

$c7 : average(L, A) \leftarrow new(L, N, S), div(S, N, A).$
$c10 : new(nil, N, S) \leftarrow S = 0, N = 0.$
$c14 : new(cons(H1, T1), N, S) \leftarrow H1 = H2, T1 = T2, s(N1) = N, S = S1,$
$\qquad new(T1, N1, S2), sum(H2, S2, S1).$
$c2 : length(nil, 0) \leftarrow .$
$c3 : length(cons(H1, T1), s(N1)) \leftarrow length(T1, N1).$
$c4 : sumlist(nil, 0) \leftarrow .$
$c5 : sumlist(cons(H2, T2), S1) \leftarrow sumlist(T2, S2), sum(H2, S2, S1).$