Transforming Specifications of Observable Behaviour into Programs

David Gilbert[1], Christopher Hogger[2], Jiří Zlatuška[3]

[1] City University, Northampton Square, London EC1V 0HB, U.K.
drg@cs.city.ac.uk
[2] Imperial College, 180 Queens Gate, London SW7 2BZ, U.K.
cjh@doc.ic.ac.uk
[3] Masaryk University, Burešova 20, 602 00 Brno, Czech Republic
zlatuska@informatics.muni.cz

A methodology for deriving programs from specifications of observable behaviour is described. The class of processes to which this methodology is applicable includes those whose state changes are fully definable by labelled transition systems, for example communicating processes without internal state changes. A logic program representation of such labelled transition systems is proposed, interpreters based on path searching techniques are defined, and the use of partial evaluation techniques to derive the executable programs is described.

1 Motivations

Our methodology provides a means for deriving executable programs from specifications of the observable behaviour of a restricted class of systems. The systems which are tractable by this methodology are those whose observations are discrete fine-grained steps which progressively construct data objects, expressed as terms in our approach. We give the characterisation of these systems by the use of a language based on labelled transition systems, and identify a class of interpreters derived from rewriting and path searching algorithms on the graphs induced by the labelled transition systems. Furthermore, we are able to derive programs by partially evaluating such interpreters with respect to the rules in the language of the labelled transition systems. We also provide a formalism which permits the transformation of the labelled transition systems into the target programs within the framework of computational logic.

The class of computations in which we are interested contains those whose result is incrementally constructed, whilst at the same time the partial results are being output as *observations* which are accessible to the environment of the computing agent. For some of these computations it is natural also to *specify* such processes in terms of their observable behaviour. We believe this may be closer to the user's understanding of the system to be programmed, when the external behaviour of the system and the sequence in which the result is produced (i.e. a trace history) is an essential part of the activity of the process. For the sake of simplicity, we assume that the observation of the external behaviour is tightly linked to the internal computation steps of such a system, i.e. each step of the computation strictly extends the resulting data structure which it constructs.

Our method regards observations of the progress of a computation as *extrinsic specifications* which can be represented as directed acyclic graphs. Each computation that the system can perform is represented by a path through the graph, which in turn can be described in first order logic. Logic programs can be derived from these first order logic descriptions by standard transformations.

We view the long-term objective of our method to be the construction of reactive concurrent systems. As a first step towards this objective we present a working framework for sequential systems. From a more abstract point of view, we can understand the procedure for constructing programs from graphs representing observable behaviour as a compiler of a graph-based production language. A particular strategy for the generation of computations from such a graph can be linked to the particular strategy for the search of the tree representing the trace history of the execution of the corresponding program. In general, the method as presented in this paper makes no assumptions about the sequential or concurrent behaviour of the programs which have been generated; such behaviour is a result of the execution mechanism for these programs.

2 Summary of the Approach

The outline of our approach can be given as follows: first, give a specification of all the possible sequences of the observable behaviour of the system to be constructed. This is done by determining the elementary transitions between the states of the program which produce the observables, and taking these transitions as a definition of a labelled transition system which can generate all the possible state changes. Since we assume that every change of state of a program is immediately reflected in production of an observable (i.e. an output visible to the external observer), we identify internal states with their associated observations.

Second, take a general *interpreter* of the resulting labelled transition system expressed in a suitable language. This interpreter is a path searching algorithm which explores the graph of transitions generated by the system. The structure of observations, labelled transitions, and the path searching procedure can all be formalised in the language of first-order Horn clause logic. Labels of the initial transformation system are identified with suitable variables, and partial objects built as a result of partial execution of the program. These objects can be identified with terms containing variables in places at which the structure will be later extended during further program execution. Within the language of logic, generation of a particular observable corresponds to substituting the term representing this observable for the free variable at the location where the new observable occurs. Hence the term which is being constructed during a program run corresponds to a tree of a particular history of observable state transitions, represented as a term.

When specifying path-searching procedures working over the state space of a particular labelled transition system represented in logic, *substitution* plays the the rôle of the basic operation performed. Each transition of the initial labelled transition system thus corresponds to an atomic substitution, and sequences of transitions correspond to compositions of atomic substitutions of this kind. Based on this, the path-searching procedure works over compositions of substitutions. Therefore the resulting program defined by such a system amounts to a generator of substitutions. These substitutions in turn correspond to changes of program observables, i.e. to program state transitions. The substitution generator therefore works as an interpreter working over the labelled transition system.

Finally, generate a program which implements the labelled transition system by partially interpreting the path searching algorithm applied to the set of transition rules. We employ a partial evaluator for logic programs for this and hence use logic programming for all three steps.

The scheme outlined above depends on the feasibility of realising each of the steps involved so that the goal of generating the program from the specification can in fact be achieved. In this paper we describe a particular formal framework which permits this goal to be accomplished. We start with a simple definition of a labelled transition system defined as a system for synchronously rewriting several labels during one transition step. The intuitive meaning of this definition is that several processing agents can act synchronously within the computational environment. We then embed these systems into clauses defining transitions of the system. From this point on, all of the construction is performed in a logic programming language, Prolog in our test implementation, starting from data structures, path-searching algorithms and generation of state-change histories i.e. terms generated by subsequent applications of substitutions corresponding to observable changes. The partial interpretation needed is therefore just a general logic programming partial interpreter (Mixtus [21] in our case).

Within each of the steps we discuss the data structures involved and the simplifications which can be employed. Note that because of the meta-programming features of our approach which is based on a path-searching interpreter, we need to specify substitutions as operations at the meta-level, rather than to rely on substitutions performed by the underlying engine of the logic programming language used for implementation. If the method is to be practically usable, the implementation of the manipulation of substitutions has to be substantially simplified in order that the generation of the final programs by partial evaluation terminates. Special discussion is therefore devoted to using the general properties of the substitutions which can possibly occur during the process of path generation, and to designing a modified definition of substitution suitable for this step.

3 Specifying Observable Changes

We use a labelled transition system (LTS) to describe possible changes of observables in the system. Such a system is given by a set of transition rules of the form

$$(x_1, \ldots, x_n) \mapsto (t_1, \ldots, t_n) \qquad \text{where } \; n \geq 1$$

Note that we permit more than one label on the left-hand side of the transition, enabling us to describe systems where more than one observables may change concurrently. $x_1, \ldots, x_n$ are the labels, or identifiers representing observables, and $t_1, \ldots, t_n$ are general expressions built over the labels and other atoms. These latter denote the resulting configuration after observable change, and may include the observable identifiers again as a proper subpart of any of them. The expressions on the right-hand side correspond to fragments of the trees (terms) of trace histories associated with observable data generation.

An example of a labelled transition rule which describes the generation of a list is

$$(x) \mapsto (a.x)$$

We may extend this to the description of a system which counts the number of items in a list:

$$(x, y) \mapsto (a.x, succ(y))$$

The informal motivation is to consider the labels as states, and to take each transition rule as a definition of a state change, possibly acting synchronously over several processes (if $n > 1$). The expressions on the right-hand side permit the definition of both the observable output and the resulting change of the state, including termination or splitting into several processes. One can think of the expressions generated by systems of this kind as snapshots of trace histories of processes which are represented by labels. Transitions can be applied to any of those labels in order to expand the structure representing the current partial trace-history. Observables produced by the system correspond to functors (atoms) occurring in the expressions generated by the LTS. (In the logic programming representation, these will be functors of the language.)

The descriptive power of the formalism is most easily understood by considering the class of processes which can be determined by a LTS as a language generated by a grammar derived from it. On an abstract level, any LTS corresponds to a *grammar* whose nonterminal symbols represent the labels of the LTS, and whose terminal symbols correspond to data structures. Thus the nonterminals actually correspond to states of a computation (sequential or parallel) represented by expanding the starting state. Even in the sequential case, the resulting pattern is different from just recursive descent due to the treatment of all the non-terminals produced in an expansion step as a partial process output. This reflects our interest in focussing primarily on generating/specifying *traces* of computations as sequences of process outputs, not just the resulting (data) structure given by the words generated by the grammar.

If the transitions of the LTS transitions only have one label expanded at each step the resulting grammar is at most a context free grammar, with all the inherent limitations of CFGs, which for example cannot represent the concurrent update of more than one label. The treatment of the class of systems which we consider within our framework contains transition rules which can concurrently transform several labels at the same time, permitting us to describe concurrent systems, and leads to a sub-class of context grammars which is strictly larger than CFGs.

4 Target Program Structure

The processes specified by this class of LTS can be represented in various ways, depending on the actual programming paradigm selected. In our approach, representation as a logic program is chosen, because of the declarative nature of this paradigm. This permits us to develop a framework for program synthesis which is independent of the particular implementation of the processes it defines, either sequential or concurrent. When the resulting logic programs are coupled with a corresponding evaluation strategy, this is effectively equivalent to a program in a procedural programming language, yet the particular level of abstraction permits a more succinct representation of the problem.

The observables of a logic program are the logical variables in the initial goal associated with it. Unification is the finest level of granularity which is useful to observe, and thus unification steps are taken to be the atomic events which are observable. Communication in a logic programming system occurs via bindings made to *shared* variables, and our assumption is that an observer can detect the *incremental* bindings made to the variables in the initial goal (i.e. to external variables). The observations made are posets of binding sets; we can represent these posets as directed acyclic graphs, due to the write-once nature of the logic variable. The bottom element of such a set represents the initial unbound state of the observable variables. Each path through the graph from the minimum vertex to a maximum vertex comprises the observations of one computation and the union of the sets associated with all such paths comprises the instantiation set of the observational variable(s).

An example is the instantiation set of the following directed graph for the variable x. Nodes are labelled with the term to which x is bound, and an arc from node A to node B is labelled with the substitution which when applied to the term at A results in the term at B.

$$\begin{tikzcd}
 & f(t_1,t_2) & \\
f(t_1,z) \arrow[ur, "\{z/t_2\}"] & & f(y,t_2) \arrow[ul, "\{y/t_1\}"'] \\
 & f(y,z) \arrow[ul, "\{y/t_1\}"] \arrow[ur, "\{z/t_2\}"'] \arrow[uu, "\{y/t_1\ \ z/t_2\}"] & \\
 & x \arrow[u, "\{x/f(y,z)\}"'] &
\end{tikzcd}$$

Fig. 1.

$$\{x/f(y,z), x/f(t_1,z), x/f(t_1,t_2)\} \cup \{x/f(y,z), x/f(y,t_2), x/f(t_1,t_2)\} \cup$$
$$\{x/f(y,z), x/f(t_1,t_2)\} = \{x/f(y,z), x/f(t_1,z), x/f(y,t_2), x/f(t_1,t_2)\}$$

Given the choice of logic variables as observables, the actual data items produced by a running program are represented by the functors of the language of terms. Therefore changes of a process state are manifested by assignment of data to the tuple of logic variables representing the process. The particular syntactic structure is just a consequence of our use of logic terms to represent data structures. During the transformation of terms into prefix/postfix notation, the non-variable symbols in them correspond straightforwardly to terminal symbols of the grammar associated with the original LTS. The use of logic variables permits the representation of trace histories as simple variable bindings, with no additional formalism being needed.

An example of an LTS is the pair of transition rules:

$$\begin{array}{c}(x) \mapsto (a.x)\\(x) \mapsto (nil)\end{array}$$

These rules represent an LTS which describes the behaviour of a process which binds a variable to list and whose trace history is itself a list of the binding states of that variable.

There is a regular grammar corresponding to this system,

$$\begin{array}{l}X \to aX\\X \to \epsilon\end{array}$$

characterizing the set of process behaviour as the corresponding regular set of sequences of data items a of arbitrary length.

The logic-variable representation uses just one type, X, for the variables of the LTS. The tuples on the right-hand sides of the production rules can be represented by terms $a(x)$, where x is a variable of the type X and a a unary function symbol, and by a nullary functor symbol nil, respectively.

5 Logic Program Representation

When specifying a logic program, we choose to identify the observables by logic variables. As far as these correspond to distinct labels of the LTS, or distinct non-terminals of the grammar, there is a need to ensure that the correspondence of every variable with its associated label is preserved. Without such correspondence substitutions may be applied to incorrect variables. This correspondence can be achieved by partitioning the sets of labels and identifiers using tags. On the level of specification, we choose to work in a *multi-sorted* logic, where types can be used to perform this tagging function. Each variable is therefore associated with a unique fixed *type*. All the usual properties of logic programs are preserved within this, and the only change to the underlying machinery required is that of modifying unification so that it fails whenever an attempt to bind a variable of certain type to a term of *different* type is made.

Thus, for example, in the rule

$$(x, y) \mapsto (a.x, succ(y))$$

we consider that x and y are of different types.

The type scheme resulting from the use of labels of the LTS as types of the system permits simple static type checking, for example that of Gödel [11]. Note that with static type checking it is sufficient to verify type constraints at the level of the source program code, and so at run-time it is possible to use type-less logic programming language, such as Prolog, and hence not to refer to types. In our case this corresponds to the need to ensure proper type constraints when writing the interpreter, as proposed by Hill and Lloyd [10], but the actual programs generated by partial evaluation are ordinary type-less logic programs.

The idea of using typed terms is just a syntactic means for avoiding the use of dynamic predicate-based type checking. The untyped predicate logic is expressible enough to define all that is needed for this, but requires the use of a more complex clause structure for the representation of the transition rules. Specifically we need to introduce predicates for *dynamic* type checking into each of the clauses of the interpreter. The framework of typed terms seems more natural in our context for two reasons. Firstly because of the example of the successful use of types in logic programming which has been set by Gödel, and secondly because the use of types simplifies the representation of transition rules as clauses by effectively moving the type-checking predicates out of such clauses into the code of a general-purpose interpreter.

For the representation of the LTS, the left-hand side of a rule becomes a tuple of logic variables, and the right-hand side is represented by a tuple of terms containing new versions of the variables, all of the variables being typed by the appropriate LTS label types. The version of the above example would be

$$(x, y) \mapsto (a.x', succ(y'))$$

where x and x' are of the same type, and so are y and y'.

In order to implement the above process in Prolog, we choose a representation of variables in which each variable carries its source observable id as a type associated with it. This observable id tag controls the possible variable occurrences which may or may not match with the variables resulting from a labelled transition rule. Obviously, when using a typed logic programming language such as Gödel, the representation could be made simpler.

Observable changes can now be described by the successive instantiation of variables, a characteristic feature being the possibility of binding variable to a term containing yet more variables. Non-linear structures can be generated in such a way, with several new observables being generated as a result, for example the generation of tree structures.

Instantiations of variables are carried out by substitutions, defined as morphisms on terms, fully described by their result on variables. In the case of *finite* substitutions (which only change a finite number of variables), the usual notation

$$\theta = [x_1/t_1, \ldots, x_n/t_n]$$

describes a mapping defined as

$$t\theta = \begin{cases} t_i & \text{if } t = x_i \text{ for } x_i/t_i \in \theta; \\ t & \text{if } t \text{ is a variable, not occurring as } t/u \in \theta \text{ for any } u; \\ f(t_1\theta, \ldots, t_n\theta) & \text{for } t = f(t_1, \ldots, t_n), n \geq 0. \end{cases}$$

Substitutions define state-changing operations on the processes, and the program-generating process developed later in this paper is based on building a meta interpreter which combines substitutions in a suitable way.

The process of observables transformation leads to the composition of substitutions defined as function compositions. On finite substitutions this this gives the following standard definition:

$$\theta\sigma = [x/t\sigma | x/t \in \theta \text{ and } x \neq t\sigma] \cup [y/s \in \sigma \text{ and for every } t,\ y/t \notin \theta]$$

Note that the second operand of the union allows us to eliminate those changes to variables defined by σ which are ineffective because of a previous elimination of suitable variable occurences by θ. We will employ this fact in the following section to simplify our working definition of composition.

At this point, the LTS can be transformed into substitutions: for each rule

$$(x_1, \ldots, x_n) \mapsto (t_1, \ldots, t_n)$$

generate a set of substitutions of the form

$$[x_1/t'_1, \ldots, x_n/t'_n]$$

with identifiers expressed as logic variables. Moreover, within each pair x_i/t'_i, t'_i is formed from t_i by renaming all variables into fresh ones. As noted above, we assume the existence of typed variables, and hence the framework of a multi-sorted language. When actually implementing this operation in a language lacking strict type discipline (such as Prolog), some extra care must be taken in the actual code to ensure that the types of the variables are preserved when renaming them.

Now the program specification part can be viewed as the set of rules describing the accumulation of substitutions: input substitution is composed with the observable-changing substitution in order that the resulting substitution is a new configuration of the system.

6 Instantiation Steps

Our method describes computations as ones which progressively instantiate variables to terms. We represent terms explicitly by substitution sets, and describe the instantiation of a term t to a more specialised form t' by the relation $\mathsf{compose}(x, y, z)$ where

x is the substitution set associated with t
z is the substitution set associated with t'
y is the substitution set whose composition with x results in z.

For example, consider the following set of possible instances of a variable x
$\{x, f(y,z), f(t_1,z), f(y,t_2), f(t_1,t_2)\}$
which corresponds to the set of *atomic substitutions* illustrated by Figure 1 above. From this poset we may extract, by closure over the arcs, one of the possible binding histories, e.g.
$[x/f(y,z)]..[x/f(t_1,z)]..[x/f(t_1,t_2)]$
which we illustrate in Figure 2 below

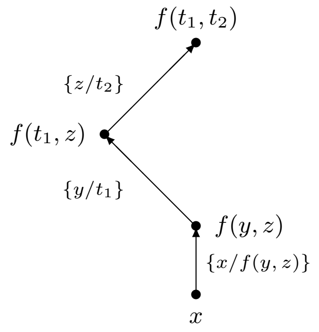

Fig. 2.

We can then associate the following compositions with the binding history:
compose$([x/f(y,z)], [y/t_1], [x/f(t_1,z), y/t_1])$,
compose$([x/f(t_1,z), y/t_1], [z/t_2], [x/f(t_1,t_2), y/t_1, z/t_2])$

We will need to add some restrictions to the standard definition of substitution in order to refine our technique. The first is linked to our basic assumptions about the class of systems we are interested in. The acyclicity of the underlying LTS reflects the intuition that during the process of computation there is always some visible non-empty output after each step. In terms of bindings, this means that there is a *progression* in the binding sequence which prohibits simple renaming occurring as local instantiation steps. This permits the elimination of useless compositions such as compose$([x/y], [y/z], [x/z, y/z])$ and hence prohibits the inclusion of x/y where x and y are variables in the substitution set at the second argument of compose/3

Our top-level definition is

compose(A, B, C) $\leftarrow$ progress(A, B, C) , full-compose(A, B, C)

where full-compose/3 is the relation corresponding exactly to the general mathematical definition of substitution composition without any additional restrictions.

Here progress(A, B, C) means that C is not a variant of A, i.e. B must be some non-trivial binding corresponding to an observable state change.

The above-outlined model is too general and contains excessive checks on the substitutions which result from composition. The checks for idempotent substitution pairs and for the elimination of variables from the standard definition of substitution make the partial evaluation of the target program unnecessarily complicated. Pragmatically we found the definition to be too general, and the partial evaluation did not terminate under Mixtus. The picture can be simplified by the consideration of the constraints result from the source LTS structure and the way substitutions are generated from it.

First, variables on the right-hand side of substitution pairs can be renamed, so that $x_i \notin t_i$ for any x_i/t_i. Also, for any $x_i/t_i \in \sigma$ there is no $y_i/u_j \in \sigma$ such that $x_i \in u_j$ for any substitution pair θ, σ, which can possibly be considered a result from composing substitution θ with substitution σ resulting from LTS. This is because variables in u_j have been generated as fresh, not yet occurring elsewhere. As a result, the idempotency check can be omitted from the definition of composition.

Second, in substitutions θ and σ, for any $x_i/t_i \in \theta$ there is no $y_j/u_j \in \sigma$ such that $x_i = y_i$. Again, this is a result of state-changing substitution being generated from LTS rules. Consequently, the elimination check can be omitted as well.

The resulting definition of substitution needed for describing observable changes therefore reads as follows:

$$\theta\sigma = [x_i/t_i\sigma | x_i/t_i \in \theta] \cup \sigma.$$

This allows us to simplify the partial-evaluation phase significantly (see Section 9), and to provide for a manageable program generator out of the source LTS.

7 Constraining Instantiation Steps

The definition of composition as given above is too general for our purposes and does not constrain instantiation to any particular discrete steps. The concept of expanding the underlying LTS-related grammar corresponds to performing computations using the LTS-based specification which leads to the identification of computational steps with the expansion of non-terminals, i.e. variable instantiation. Such an instantiation is limited to terms of the structure which correspond to the left-hand sides of transformation rules, i.e. such an instantiation step typically replaces variables either by constants or by trees of a restricted depth. Hence we are only interested in systems which instantiate terms in such a minimal manner, and thus introduce the notion of a (non-null) atomic substitution set Y whose application to a substitution set X by compose(X,Y,Z) satisfies the following constraints:

- There is at least one non-ground t_i of $v_i/t_i \in$ X which is further instantiated by $v_j/t_j \in$ Y (where t_j is not a variable).
- v_j/t_j is minimal in some sense to the particular application

We can define atomic/1 which is part of the meta-interpreter by:

atomic(Y) $\leftarrow$ lts(P $\mapsto$ Q), transform(P $\mapsto$ Q, Y)

where lts/1 is the object level program and transform/2 performs the transformation operation from the LTS into substitutions referred to in Section 5. For example, we may take for lists the pair of transition rules represented by lts/1:

lts($(x) \mapsto (a.x)$)
lts($(x) \mapsto (nil)$)

which are transformed to $[x/a.x']$ and $[x/nil]$ respectively.

We now can give a definition of a new predicate inst(X,Z) which relates a substitution set X, about a term T, to a substitution set Z, about the immediate successor T' of T, as determined by some applicable substitution set Y:

inst(X,Z) $\leftarrow$ atomic(Y), compose(X,Y,Z)

8 Node Traversal of Instantiation Graphs

The inst/2 predicate allows us to describe only one individual step, or edge, in the instantiation graph which represents the graph of transitions, whereas we ultimately intend to describe the graph as a whole. More specifically, having provided a definition or program for 'inst' in respect of some particular application, we want to incorporate it within some encompassing program which *traverses* the DAG determined by 'inst'. The classic path traversal method, for example as shown by Kowalski [13] relates nodes (in this case substitution sets) to their subsequent states. Consider any node N already generated; after one or more atomic steps, various paths will have been developed to some further node N. We can define the following path-finding programs by the transitive closure of inst/2:

Program A1
path(N, F) $\leftarrow$ inst(N, F)
path(N, F) $\leftarrow$ inst(N, N$'$) , path(N$'$, F)
inst(N, F) $\leftarrow$ compose(N, Y, F) , atomic(Y)

Program A2
path(N, F) $\leftarrow$ inst(N, F)
path(N, F) $\leftarrow$ inst(N$'$, F) , path(N, N$'$)
inst(N, F) $\leftarrow$ compose(N, Y, F) , atomic(Y)

Which program is used is determined by the input-output mode with which path/2 is queried; the entire graph can be traversed by inputting N as the bottom node and using path1 to seek all reachable nodes F, or vice-versa.

We can now follow the method of Gilbert and Hogger [8] which derived path exploration programs which computed only the *differences* between nodes. In the context in which compose/3 appears, constrained together with atomic/1, it cannot be used in any way which would not satisfy the following conditions, as defined by Brough and Hogger [4]:

(1) $(\forall \mathrm{N}\ \forall \mathrm{Y}\ \exists \mathrm{F})(\mathrm{compose(N,Y,F)})$
Some result F be *defined* for any Y applied to any N.
(2) $(\exists \mathrm{I}\ \forall \mathrm{Y}\ \forall \mathrm{F})(\mathrm{compose(I,Y,F)} \leftrightarrow \mathrm{Y=F})$
compose to have at least one *left-identity* I.
(3) $(\forall \mathrm{N}\ \forall \mathrm{Y}\ \forall \mathrm{Y'}\ \forall \mathrm{F})\ ((\mathrm{compose(N,Y',F)} \leftrightarrow \mathrm{Y'=Y}) \leftarrow \mathrm{compose(N,Y,F)})$
$(\forall \mathrm{N}\ \forall \mathrm{Y}\ \forall \mathrm{F}\ \forall \mathrm{F'})\ ((\mathrm{compose(N,Y,F')} \leftrightarrow \mathrm{F'=F}) \leftarrow \mathrm{compose(N,Y,F)})$
compose to be *functional* in two of its modes.
(4) $(\forall \mathrm{N}\ \forall \mathrm{N'}\ \forall \mathrm{Y}\ \forall \mathrm{Y'}\ \forall \mathrm{Y''}\ \forall \mathrm{F})\ ((\mathrm{compose(N,Y,F)} \leftrightarrow \mathrm{compose(N',Y',F)})$
$\leftarrow (\mathrm{compose(Y'',Y',Y)} \wedge \mathrm{compose(N,Y'',N')}))$
compose to be *associative*.

This new path exploration relation path/1 is defined by

Program B1
```
path(Y) ← atomic(Y)
path(Y) ← compose(Y″, Y′, Y) , atomic(Y″) , path(Y′)
```

Program B2
```
path(Y) ← atomic(Y)
path(Y) ← compose(Y″, Y′, Y) , atomic(Y′) , path(Y″)
```

We should note that all the four path searching programs above act as interpreters for programs defined by the lts/1 relation.

9 Program Derivation by Partial Evaluation

Programs A1, A2, B1 and B2 can be used as general templates to describe a class of programs which incrementally instantiate observable variables during the course of their execution; specific instances of programs are determined by the definition of atomic/1, determined by lts/1. We have coded the relations for compose/3, atomic/1 and both path/1 and path/2 in SICStus Prolog in order to explore the possibilities of transforming the generic path programs into specialised forms for given LTS's. A design decision was taken early on to distinguish between meta-level and object-level variables in the Prolog code by using the ground term representation, in order to preserve the semantics of the definitions.

We then use partial evaluation in order to produce a program which incorporates the path searching interpreter and our compose/3 relation with a specific labelled transition system as defined by lts/1. Partial evaluation of logic programs is an optimisation technique which has been described in logic programming terms by Lloyd and Shepherdson [15] as follows: "Given a program P and a goal G, partial evaluation produces a new program P' which is P 'specialised' to the goal G. The intention is that G should have the same (correct and computed) answers w.r.t. P and P', and that G should run more efficiently for P' than for P". Both folding and unfolding are techniques used in partial evaluation:

- *logical folding* is the replacement of a goal that is an instance of the body of a clause by the corresponding instance of the head of the clause;
- *logical unfolding* of the goal X_i in the clause
  $H \leftarrow X_1, \ldots, X_{i-1}, X_i, X_{i+1}, \ldots, X_n$
  where X_i is defined by $X \leftarrow B_1, \ldots, B_m$ is defined by the following transformation:
  $\{(H \leftarrow X_1, \ldots, X_{i-1}, B_1, \ldots, B_m, X_{i+1}, \ldots, X_n)\theta \mid \text{mgu}(X, X_i) \wedge \theta \neq \text{false}\}$

Although partial evaluation can be done by hand, we have used Mixtus [21], the excellent partial evaluator for Prolog developed by Dan Sahlin, and have obtained good initial results.

The choice of Prolog permits the simplification of the relationship between the data structures used, the meta-interpreter of the substitution-changing relation, and the resulting synthesized code. This is because the same language is used to represent the observables, the program in the LTS language and the interpreter based on graph searching. The methodology itself is nonetheless applicable to any implementation language, but the amount of code actually needed may be significantly greater.

In the case of an implementation in logic programming, the method naturally does not provide any universal mechanism for synthesizing arbitrary logic programs. By the nature of the initial assumptions chosen, only programs whose state changes result in bindings to variables in an initial query are expressible by this method. On the general level of process description this corresponds to systems whose change of state is always visible to the outside environment, e.g. via observable communication between processes.

10 Comparison with other work

The algebraic structure of atomic formulae, including the lattice properties of the instantiation order relation, were described independently in the early seventies by Reynolds [18] and Plotkin [16]. Both authors were also interested in mechanical theorem proving. Reynolds showed in his paper that the refutation of "transformational systems" (sets of clauses containing only unit clauses and clauses with one positive and one negative literal) was in effect path searching, but that there was no decision procedure for such systems. Plotkin discussed the use of induction to find least generalizations of clauses or literals and showed that there is an algorithm to find the least generalization of any pair of literals or terms. It is interesting to note that in his paper Plotkin considered the possibilities of automated induction; the path searching algorithms described in our work relate to induction, but it is not our goal to try to enhance the specification by generalisation.

Belia and Occhiuto [2] have developed an explicit calculus of substitutions which extends Reynold's gci [18] and combines Robinson's unification [19] and term instances. Their work aims to avoid the drawback of the gap between the theory and current implementations of logic programming languages represented by the use of metalevel structures, like substitutions, and mechanisms, like mgu and instantiation, to deal with the substitution rule. Their calculus of c-expressions permits structures to be dealt with explicitly at the object level which would otherwise typically be hidden at the metalevel. They put the substitution rule as an additional operator, mgi, of the language of terms, and provide c-expressions as programs. We prefer to work initially at the metalevel, and to take a classical approach based on interpretation [13, Chapter 12] and partial evaluation (see for example [15, 3]). C-expressions which use integers as tuple indexes do not exploit the tree structure of terms; Belia and Occhiuto are investigating a different calculus using paths instead of integers, which may be closer to our approach.

The language of Associons [17] was developed by Martin Rem as a program notation without variables; the motivation was to develop a language model which employed more concurrency than traditional languages based on assignment. An associon is a tuple of names defining a relation between entities represented by these names. The state of a computation can be changed by a forward chaining process based on the "closure statement", which creates new associons that represent new relations deduced from the already existing ones. The language of associons is essentially deterministic and is based on sets. In fact the language does have logical variables and also universal quantifiers over closure statements. These statements are effectively normal program clauses (i.e. they can contain negated conditions) [14], and also contain guards. Our programs are expressed as *definite* program clauses, without negated conditions, and in contrast to Rem's language our approach is based on the backward-chaining principle of logic programming and permits the expression of all-solutions non-determinism since our language does not contain guards. A similarity between our approach and that of Rem is that both formalisms permit the construction of programs which are inherently concurrent.

Banâtre and Le Métayer have developed the Gamma language [1] which also permits the construction of programs which are inherently parallel in their operation. A Gamma program is essentially a multiset transformer operating on all the data at once; it 'reacts' on multisets of data by replacing a subset whose elements satisfy a given property by the result of the application of a function on that subset. Gamma is an intermediate language between specifications in first order logic and traditional programming languages; programs in Gamma describe the logic of an algorithm and are transformed into executable programs by expressing lower-level choice such as data representation and execution order. Central to the Gamma language is non-determinism, expressed as the choice between several subsets which are candidates for reaction, and the locality principle, which permits independent and simultaneous reactions on disjoint subsets. The multiset is seen as a representation of the state of a system in Gamma; although our approach is based on sets rather than multisets, there is a similarity in that the set of bindings is regarded as the state of the system. There is no recursive data structure definition in Gamma, and data has to be represented as flat multisets of items; for example trees are represented by nodes and leaves associated with parenthood information. We preserve the tree structure of terms due to our use of term substitution, but do employ types to indicate the position of subterms within a term. In both Gamma and our formalism this means that all components of a data structure are directly accessible, independently of their position in the structure. However, in contrast with Gamma, our technique does not have direct equivalents to the operations of data expansion and data reduction.

The work of Gilbert and Hogger [8, 9] described the instantiation of a given term X by a given substitution Y to give a new term Z by a relation subst(X,Y,Z) where any substitution Y was constructed by applying a tupling <> to some set $\{Y_1, \ldots, Y_n\}$ of simpler substitutions. Y was represented by means of a *substitution-tree*, and a special symbol Δ used exclusively to describe variables. A major drawback of this method is that it was only applicable systems which generated lists, since the non-inclusion of S-trees in the Herbrand universe required the transformation of the subst/3 relation into one which does operate over Herbrand terms, but which is not associative.

Our work is closely related to the area of partial evaluation, in particular as applied to logic programs. Work in this field has been carried out by Lloyd et al. [15, 3] amongst others, who have given a strong theoretical foundation for partial evaluation in logic programming, and Dan Sahlin, who has constructed a robust partial evaluator for Prolog [20, 21]. Partial evaluation for concurrent logic languages has been explored by Fujita et al. [6] for GHC programs, and by Huntbach [12] for Parlog programs.

Furthermore, our method based on a graph traversal template can be regarded as an interpreter for graph-based computations, and in this sense is related to the work by Gallier et al. [7] on graph-based interpreters for general Horn clauses.

11 Conclusions

The work reported in this paper reconstructs the method of Gilbert and Hogger for deriving logic programs from the expected observations of program behaviour. We replace their concept of substitution trees (S-trees) by binding sets, and show that a more general method can be developed based on substitution mappings as the basis for the theory. We have formulated path exploration programs which act as generalised program schemata for certain classes of systems, and have derived specialised instances of these programs by partial evaluation of the schemata and specifications of program behaviour given in terms of labelled transition systems. The partial evaluation stage has been successfully mechanised using Mixtus [21], a partial evaluator for Prolog.

Acknowledgements

This work was partially supported by PECO Fellowship grants CT931675 (David Gilbert) and CT926844 (Jiří Zlatuška) provided by the European Community under the scheme for Scientific and Technical Cooperation with Central and Eastern Europe.

References

1. J-P. Banâtre and D. Le Métayer. Programming by Multiset Transformation. *Communications of the ACM*, 36(1):98–111, 1993.
2. M. Belia and M. E. Occhiuto. C-expressions: a variable–free calculus for equational logic programming. *Theoretical Computer Science*, 107:209–252, 1993.
3. K. Benkerimi and J. W. Lloyd. A partial evaluation procedure for logic programs. In Debray and Hermenegildo [5], pages 343–358.
4. D. R. Brough and C. J. Hogger. Compiling associativity into logic programs. *The Journal of Logic Programming*, 4(4):345–360, December 1987.
5. S. Debray and M. Hermenegildo, editors. *Proceedings of the 1990 North American Conference on Logic Programming*, Austin, 1990. ALP, MIT Press.
6. H. Fujita, A. Okumura, and K. Furukawa. Partial evaluation of GHC programs based on the UR-set with constraints. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 924–941, Seatle, 1988. ALP, IEEE, The MIT Press.
7. J. H. Gallier and S. Raatz. Hornlog: A graph-based interpreter for general Horn clauses. *The Journal of Logic Programming*, 4(2):119–156, June 1987.
8. D. R. Gilbert and C. J. Hogger. Logic for representing and implementing knowledge about system behaviour. In V Mařík, O Štěpánková, and R Trappl, editors, *Proceedings of the International Summer School on Advanced Topics in Artificial Intelligence*, pages 42–49, Prague, Jul 1992. Springer Verlag Lecture Notes in Artificial Intelligence No. 617.
9. D. R. Gilbert and C. J. Hogger. Deriving logic programs from observations. In Jean-Marie Jacquet, editor, *Constructing Logic Programs*. John Wiley, 1993.
10. P. M. Hill and J. W. Lloyd. Analysis of Meta-programs. Technical Report CS-88-08, Department of Computer Science, University of Bristol, Bristol, UK, June 1988.
11. P. M. Hill and J. W. Lloyd. *The Gödel programming language*. MIT Press, 1993.
12. M. Huntbach. Meta-interpreters and partial evaluation in parlog. *Formal Aspects of Computing*, 1(2):193–211, 1989.
13. R. A. Kowalski. *Logic for problem solving*. North Holland, 1979.
14. J. W. Lloyd. *Foundations of Logic Programming*. Spinger-Verlag, Berlin, second edition, 1987.
15. J. W. Lloyd and J. C. Sheperdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11(3 & 4):217–242, October/November 1991.
16. G. D. Plotkin. A note on inductive generalization. In B. Meltzer and D. Mitchie, editors, *Machine Intelligence*, pages 153–165, 1970.
17. M. Rem. Associons: A Program Notation with Tuples instead of Variables. *ACM Transactions on Programming Languages and Systems*, 3(3):251–262, Jul 1981.
18. J. C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. In B. Meltzer and D. Mitchie, editors, *Machine Intelligence*, pages 135–151, 1970.
19. J. A. Robinson. A machine-orientated logic based on the resolution principle. *Journal of the ACM*, 12(1):23 – 49, Jan 1965.
20. D. Sahlin. The mixtus approach to automatic partial evaluation of full Prolog. In Debray and Hermenegildo [5], pages 377–398.
21. D. Sahlin. *An Automatic Partial Evaluator for Full Prolog*. PhD thesis, Swedish Institute of Computer Science, Mar 1991.