1

Deriving Logic Programs from Observations

DAVID GILBERT, CHRISTOPHER HOGGER

This chapter describes a method for deriving logic programs from observations that can be made of the progress of a computation. Such observations can be regarded as *extrinsic specifications*, and 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. In this chapter we chose to derive sequential logic programs, but the method is equally applicable to concurrent systems.

This chapter is organised as follows. Section 1.1 introduces the basic concepts of observations of logic programming systems as posets, section 1.2 introduces the terminology required for the description of computations in terms of graph theory and section 1.3 describes the graph properties of sets of instantiation states of variables during computations. In section 1.4 we define a relation between states and substitutions and use it to describe paths through graphs of the instantiation states of variables, each path through such a graph representing one computation. In section 1.5 we show how this relation can be specialised to describe substitutions over recursively defined data structures such as lists, and also terms representing the natural numbers. Section 1.6 introduces the concept of *restricted substitutions*, and in particular *atomic substitutions*. Our method for deriving logic programs from the recursive definition of paths through an instantiation graph, making use of the atomic substitution relation, is described in section 1.7, and section 1.8 presents a worked example for summing the elements of a list of natural numbers. We set our methodology within the wider context of our research programme in section 1.9, and also refer to earlier work carried

Constructing Logic Programs, J.-M. Jacquet (editor)

out by other researchers which provides some of the theoretical underpinning to our own research.

1.1 INTRODUCTION

A logic programming system can be viewed as a black box, whose task is to compute bindings; an observer is only permitted to perceive its input-output behaviour. Communication in such a system occurs via bindings made to *shared* variables. Our assumption is that during the execution of the system an observer can detect the *incremental* bindings made to the variables in the initial goal (i.e. to external variables). Bindings may be made to internal variables, but these are not detectable by the observer.

The observations that can be made of these bindings comprise sets which are partially ordered by relations determined by the topologies of the structures to which the shared variables are bound. The bottom element $\perp$ of such a set represents the initial unbound state of the observable variables. The sets of observations can also be described as directed acyclic graphs, where each edge corresponds to a single unification step. Due to the 'write once' nature of logic variables, no such step can undo any bindings already made and thus such a graph is a directed acyclic graph. Each path through the graph from the minimum vertex to a maximum vertex comprises the observations of one computation. First order logic is a suitable formalism for the description of these sets since they are characterized by an associated order relation.

1.2 COMPUTATIONS, PATHS AND STATES

The *execution* of a system or process comprises one or more computations. Graph theory provides a useful basis for describing these sets of computations. In this chapter we describe terminating systems (finite computations) since our reasoning is based on *finite* graphs, and only consider *successful* computations.

Definition 1 *A* computation *is any derivation beginning from an initial state which can be generated during the execution of a system. There is only one initial state of a system.*

A computation can be represented as a sequence of state changes

$$S_0 \longrightarrow S_1 \longrightarrow S_2 \longrightarrow \ldots$$

where S_0 is the initial state. A *terminating* computation can be represented by a sequence of the form

$$S_0 \longrightarrow S_1 \longrightarrow S_2 \longrightarrow \ldots \longrightarrow S_n$$

where S_0 is the initial state and S_n the final state. We may also represent the terminating computation above as the graph $G = (X, U)$ where

$$\begin{aligned} X &= \{S_0, S_1, S_2, \ldots, S_n\} \\ U &= \{(S_0, S_1), (S_1, S_2), \ldots, (S_{n-1}, S_n)\} \end{aligned}$$

Such a graph comprises just one path

$$[(S_0, S_1), (S_1, S_2), \ldots, (S_{n-1}, S_n)]$$

The order relation on the graph is defined by U, since for every (S_i, S_j) in U we may write $S_i \prec S_j$.

In general, a system may terminate in different ways, or may reach the same state in a variety of ways, resulting in more than one path through the graph. Each path represents one computation. Such a graph is

i) *directed*: since $(S_i, S_j) \in U$ means that state S_j is the next state after state S_i,
ii) *acyclic*: since once a state has been reached, it can never be returned to,
iii) *partially ordered*: since the reflexive, antisymmetric and transitive relation $\leq$ on terms of chapter 1 can be defined by $\prec$.

1.3 CHANNELS

We now look more closely at the data structures to which variables can be bound during a computation. The bindings may occur incrementally and the binding states of a variable can be depicted as a directed acyclic graph. In this section we introduce the use of first order logic to describe such graphs.

Definition 2 *A* channel *is a collection of states of a variable that occur during one or more computations, i.e. the set of instantiations that can be made to that variable.*

Such a set of instantiations to a variable is partially ordered by a prefix relation over the data structure to which the variable becomes bound. This relation which we shall denote by $\leq$ is reflexive, antisymmetric and transitive; X $\leq$ Y means that Y is an instance of X, or X is equal to Y (See definition ?? in chapter ??). The instantiation of a variable is effected by a process, and thus the channel can be described as a *graph* ordered by $\leq$, through which there are one or more paths, each representing a distinct computation. If there is more than one data structure to which the variable can be bound then the graph is an inf-semilattice graph, with one bottom element, i.e. the initial unbound state of the variable. The greatest element in each path is V/t_n where t_n is the final term to which the variable V is bound in that branch of the computation. In the case that all computations bind the variable to the same structure T the channel is a complete lattice with the top element $\top$ being V/T. For each data structure to which the variable can be bound there is an ordering relation based on a corresponding predecessor relation.

Variables are bound during unification to logic terms, and subterms in a data structure can be instantiated in parallel or sequentially. An element in the channel poset can have more than one successor if a variable is bound to a complex term. For example, the channel {V, f(A,B), f(p,B), f(A,q), f(p,q)} describes the possible binding states of variable V with final binding state V/f(p,q). The channel can be more comprehensively described by the directed acylic graph $G = (X, U)$ where

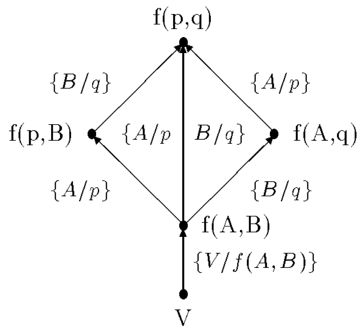

Figure 1.1 Representation of a channel, final state V/f(p,q)

$$
\begin{aligned}
X &= \{V, f(A,B), f(p,B), f(A,q), f(p,q)\} \\
U &= \{ \; (V, f(A,B), (f(A,B), f(p,B)), (f(A,B), f(A,q)), (f(A,B), f(p,q)), \\
&\qquad (f(p,B), f(p,q)), (f(A,q), f(p,q)) \; \}
\end{aligned}
$$

This graph has the complete lattice property, with bottom node V and top node $f(p,q)$.

We define the immediate predecessor relation $\prec$ as follows:

Definition 3 *If t and u are terms each of which represents a state of a variable, then $t \prec u$ iff u is derived from t either by instantiating at least one variable in the variable set of t to a term $f(V_1, \ldots, V_n)$ where $n \geq 1$ and $V_1, \ldots, V_n$ are variables, or by binding t to a constant.*

Thus the immediate predecessor relation $\prec$ on X is described by:

$$
\begin{aligned}
\{ \; & V \prec f(A,B),\ f(A,B) \prec f(p,B),\ f(A,B) \prec f(A,q),\ f(A,B) \prec f(p,q), \\
& f(p,B) \prec f(p,q),\ f(A,q) \prec f(p,q) \; \}
\end{aligned}
$$

Figure 1.1 is a graph diagram of the channel. The nodes are labelled with the state of the variable, and the arcs are labelled with the bindings made during the transition from one state to the next.

We define $\leq$ using $\prec$ and first order logic:

$$X \leq Y \Leftrightarrow (X = Y) \vee (X \prec Y) \vee \exists\, Z (X \prec Z \wedge Z \leq Y)$$

The path P between two nodes A and B in a graph $G = (X, U)$ can be described as a partial list using the infix binary functor "." .

$$
\begin{aligned}
path(A,B,U,P) \quad \Leftrightarrow \quad & ((A,B) \in U \wedge P = A.B) \vee \\
& \exists\, C, P'((A,C) \in U \wedge P = A.P' \wedge path(C,B,U,P'))
\end{aligned}
$$

Since we can characterize a graph by its order relation we may also define a path from A to B in a graph of $\prec$ by

$$path(A, B, P) \;\Leftrightarrow\; (A \prec B \wedge P = A.B) \vee \\ \exists\, C, P'(A \prec \;\wedge\, CP = A.P' \wedge path(C, B, P'))$$

where $\prec/2$ is defined by:

$$X \prec Y \;\Leftrightarrow\; (X = V \wedge Y = f(A, B)) \vee \ldots \vee (X = f(p, B) \wedge Y = f(p, q))$$

where A, B and V are variables.

1.4 PATHS

Our method describes computations which progressively instantiate variables to *terms*. We denote variables by the special symbol Δ which is used for no other purpose, and call a term which is not fully ground an *open term*. By analogy we call a term which is ground a *closed term*, and henceforth prefer to use *open* and *closed* rather than *non-ground* and *ground* respectively.

In the applications that we are going to consider later on in this chapter we shall have need to represent how a term becomes instantiated, and thus define a relation `subst/3` between states and substitutions. If `subst(X,Y,Z)` holds, then Z is the term constructed by further instantiating the term X using the tuple of terms Y. In the following we will represent any n-ary term by $F(T_1, \ldots, T_n)$ where F is a variable ranging over functor names and $T_1, \ldots, T_n$ are terms; if $n = 0$ then $F()$ stands for the constant F. For convenience we write $\langle\rangle\,(S_1,\ldots,S_n)$ as $\langle S_1,\ldots,S_n\rangle$ to represent an n-ary tuple containing the terms (or tuples) $S_1,\ldots,S_n$; if $n = 0$ then $\langle\rangle$ is a symbol representing $\langle\rangle\,()$, the *empty tuple*. Note that

i) the first two clauses permit substitution over *partially open* terms where some leaves are open and others are closed;

ii) the second argument to `subst/3` *must* be a tuple (of tuples), while the first and third arguments to the relation can be any term, including tuples.

```
subst(X,⟨⟩,X)
subst(Δ,⟨T⟩,T)
subst(F(T₁,...,Tₙ),⟨S₁,...,Sₙ⟩,F(T′₁,...,T′ₙ)) ←
    subst(T₁,S₁,T′₁), ..., subst(Tₙ,Sₙ,T′ₙ)
```

For example, consider instantiating the term f(X,g(b,Y)) to f(a,g(b,c)). This is described by

```
subst( f(X,g(b,Y)), ⟨a,⟨⟨⟩,c⟩, f(a,g(b,c)) ).
```

The second argument to `subst/3` is effectively an *outline* term of the same form as the first and third arguments with all nodes consisting of the symbol $\langle\rangle$.

1.5 SPECIALISING THE SUBSTITUTION RELATION

In the case that the term over which `subst/3` is defined is a *list* then we can dispense with the tuple form of the second argument to `subst/3`, whose definition is then similar to that of traditional `append/3`. We assume firstly that we only describe open lists and secondly that only the tail of such a list can be instantiated.

```
subst-list(X,⟨⟩,X)                      (i)
subst-list(Δ,⟨Y⟩,Y)                     (ii)
subst-list(X.Xs,⟨⟨⟩,T⟩,X'.Zs) ←         (iii)
    subst-item(X,⟨⟩,X'),
    subst-list(Xs,T,Zs)
```

We do not define the relation `subst-item/3` since lists can contain any data structures as items. Partially evaluating this call w.r.t. clause (i) above (i.e. effectively dropping the explicit universal type description for list items) results in the following definition:

```
subst-list(Δ,⟨Y⟩,Y)                     (i)
subst-list(X.Xs,⟨⟨⟩,T⟩,X.Zs) ←          (ii)
    subst-list(Xs,T,Zs)
```

A call to this version of `subst-list/3` only succeeds when its argument is a a term in which the only significant data is that stored at the deepest right leaf as $\langle \mathrm{T} \rangle$. Using this observation, we may modify the definition of `subst-list/3` so that the second argument is a list instead of a tuple $\langle \mathrm{T} \rangle$ or $\langle\langle\rangle, \mathrm{T}\rangle$ and thus we obtain a form close to the 'traditional' form of `append/3` which is only valid over *open lists*:

```
append(Δ,Y,Y)
append(X.Xs,Y,X.Zs) ←
    append(Xs,Y,Zs)
```

Similarly we can derive a variant of the `subst/3` relation specialized to the data structure `s/1` which we will use to represent natural numbers:

```
sum(Δ,Y,Y)
sum(s(X),Y,s(Z)) ←sum(X,Y,Z)
```

1.6 RESTRICTED SUBSTITUTIONS

The definition of `subst/3` relation is too general when we want to describe the behaviour of systems which progress in discrete steps. We therefore formulate a restricted relation `inst/2` which describes a *minimal* substitution in some sense over terms which must replace at least one open leaf in the term in question. This relation is effectively $\prec$ (immediate predecessor), defined in definition 3.

```
inst(X,Z) ←  subst(X,Y,Z), atomic(Y)
```

where we define `atomic/1` by:

$$\texttt{atomic}(\langle Z_1,\ldots,Z_n\rangle) \leftarrow$$
$$\exists X_k\ (Z_1 = F_1(X_1,\ldots,X_p) \wedge (X_k = \Delta \vee X_k = G_1()))\ \vee \ldots \ \vee$$
$$\exists Y_j\ (Z_n = F_n(Y_1,\ldots,Y_q) \wedge (Y_j = \Delta \vee Y_j = G_1()))$$

We will later require a definition of `inst/2` which distinguishes between atomic substitutions which result in an open, minimally extended term and those which produce a closed (ground) term whose structure has not been extended:

```
inst(X,Z) ←
    subst(X,Y,Z), atomic-open(Y)        Z̃ is open
inst(X,Z) ←
    subst(X,Y,Z), atomic-close(Y)       Z̃ is closed
```

,

```
atomic-open(⟨Z1,...,Zn⟩) ←
    Z1 = F1(X1,...,Xp), X1 = Δ, ..., Xp = Δ
atomic-open(⟨Z1,...,Zn⟩) ←
    Z1 = Δ
...
atomic-open(⟨Z1,...,Zn⟩) ←
    Zn = Fn(Y1,...,Yq), Y1 = Δ, ..., Yq = Δ
atomic-open(⟨Z1,...,Zn⟩) ←
    Zn = Δ

atomic-close(⟨Z1,...,Zn⟩) ←
    close(Z1), ..., Zk = Fk(), ..., close(Zn)

close(F())
close(⟨⟩)
```

where $F()$ and $F_k()$ stand for constants.

The definitions for `atomic-open/1` and `atomic-close/1` in the case of lists are:

```
atomic-open-list(X.Δ)
atomic-close-list(nil)
```

and for natural numbers:

```
atomic-open-nat(s(Δ))
atomic-close-nat(0)
```

1.7 THE METHOD

The basis of our method is to relate the initial state (i.e. unbound) of the variables in a query to their state at a subsequent stage in a computation. This state may be the final state or alternatively a 'frontier' state in a computation which has not yet terminated. Thus our method can be applied both to terminating and non-terminating computations. Our method comprises the following steps:

(1) Define the partial order relation $\mathcal{R}/2$ on the instantiation set of n-tuples $\mathcal{S}$ of an initial set of variables by the transitive closure of the `inst/2`. $\mathcal{R}(X,Y)$ describes the relation between any element Y $\in \mathcal{S}$ and a frontier element X $\in \mathcal{S}$:

$$\mathcal{R}\texttt{(Frontier,Node)} \leftarrow \texttt{inst(Node}'\texttt{,Node)}, \mathcal{R}\texttt{(Frontier,Node}'\texttt{)}$$
$$\mathcal{R}\texttt{(Frontier,Node)} \leftarrow \texttt{inst(Node,Frontier)}$$

(2) Rewrite $\mathcal{R}$`/2`, explicitly describing each member of the set as an n-tuple, s.t. $\texttt{Frontier=(F}_1\texttt{,}\ldots\texttt{,F}_n\texttt{)}$ and $\texttt{Node=(N}_1\texttt{,}\ldots\texttt{,N}_n\texttt{)}$

$$\begin{aligned}&\mathcal{R}\texttt{((F}_1\texttt{,}\ldots\texttt{,F}_n\texttt{),(N}_1\texttt{,}\ldots\texttt{,N}_n\texttt{))} \leftarrow\\ &\quad\texttt{inst((N}'_1\texttt{,}\ldots\texttt{,N}'_n\texttt{),(N}_1\texttt{,}\ldots\texttt{,N}_n\texttt{))},\\ &\quad\mathcal{R}\texttt{((F}_1\texttt{,}\ldots\texttt{,F}_n\texttt{),(N}'_1\texttt{,}\ldots\texttt{,N}'_n\texttt{))}\end{aligned}$$

$$\begin{aligned}&\mathcal{R}\texttt{((F}_1\texttt{,}\ldots\texttt{,F}_n\texttt{),(N}_1\texttt{,}\ldots\texttt{,N}_n\texttt{))} \leftarrow\\ &\quad\texttt{inst((N}_1\texttt{,}\ldots\texttt{,N}_n\texttt{),(F}_1\texttt{,}\ldots\texttt{,F}_n\texttt{))}\end{aligned}$$

(3) Rearrange the $2n$ elements in the two n-tuples in the definition of $\mathcal{R}$`/2` to form n pairs of the form $\texttt{F}_k\texttt{-N}_k$ (whose significance will be explained in the next step below) to define $\mathcal{R}/n$

$$\begin{aligned}&\mathcal{R}\texttt{(F}_1\texttt{-N}_1\texttt{,}\ldots\texttt{,F}_n\texttt{-N}_n\texttt{)} \leftarrow\\ &\quad\texttt{inst((N}'_1\texttt{,}\ldots\texttt{,N}'_n\texttt{),(N}_1\texttt{,}\ldots\texttt{,N}_n\texttt{))},\\ &\quad\mathcal{R}\texttt{(F}_1\texttt{-N}'_1\texttt{,}\ldots\texttt{,F}_n\texttt{-N}'_n\texttt{)}\end{aligned}$$

$$\begin{aligned}&\mathcal{R}\texttt{(F}_1\texttt{-N}_1\texttt{,}\ldots\texttt{,F}_n\texttt{-N}_n\texttt{)} \leftarrow\\ &\quad\texttt{inst((N}_1\texttt{,}\ldots\texttt{,N}_n\texttt{),(F}_1\texttt{,}\ldots\texttt{,F}_n\texttt{))}\end{aligned}$$

(4) Assign to the predicate `subst(X,Y,Z)` the interpretation "Y represents the difference between Z and X". Thus we define $\mathcal{R}/2n$ by associating a tuple X_k with each pair $F_k - N_k$ and adding the corresponding call $\texttt{subst(}N_k\texttt{,}\ X_k\texttt{,}\ F_k\texttt{)}$ to the body of the enhanced definition for $\mathcal{R}/2n$. Although the conjunctions on this line are technically redundant, we will need them in a later step when we perform replacement of calls.

$$\begin{aligned}&\mathcal{R}\texttt{(F}_1\texttt{-N}_1\texttt{,X}_1\texttt{,}\ldots\texttt{,F}_n\texttt{-N}_n\texttt{,X}_n\texttt{)} \leftarrow\\ &\quad\texttt{inst((N}'_1\texttt{,}\ldots\texttt{,N}'_n\texttt{),(N}_1\texttt{,}\ldots\texttt{,N}_n\texttt{))},\\ &\quad\texttt{subst(N}_1\texttt{,X}_1\texttt{,F}_1\texttt{)}, \ldots, \texttt{subst(N}_n\texttt{,X}_n\texttt{,F}_n\texttt{)},\\ &\quad\texttt{subst(N}'_1\texttt{,X}'_1\texttt{,F}_1\texttt{)}, \ldots, \texttt{subst(N}'_n\texttt{,X}'_n\texttt{,F}_n\texttt{)},\\ &\quad\mathcal{R}\texttt{(F}_1\texttt{-N}'_1\texttt{,X}'_1\texttt{,}\ldots\texttt{,F}_n\texttt{-N}'_n\texttt{,X}'_n\texttt{)}\end{aligned}$$

$$\begin{aligned}&\mathcal{R}\texttt{(F}_1\texttt{-N}_1\texttt{,X}_1\texttt{,}\ldots\texttt{,F}_n\texttt{-N}_n\texttt{,X}_n\texttt{)} \leftarrow\\ &\quad\texttt{inst((N}_1\texttt{,}\ldots\texttt{,N}_n\texttt{),(F}_1\texttt{,}\ldots\texttt{,F}_n\texttt{))}\ ,\\ &\quad\texttt{subst(N}_1\texttt{,X}_1\texttt{,F}_1\texttt{)}, \ldots, \texttt{subst(N}_n\texttt{,X}_n\texttt{,F}_n\texttt{)}\end{aligned}$$

(5) Rewrite calls of the form

$$\texttt{inst((N}_1\texttt{,}\ldots\texttt{,N}_n\texttt{),(N}'_1\texttt{,}\ldots\texttt{,N}'_n\texttt{))}$$

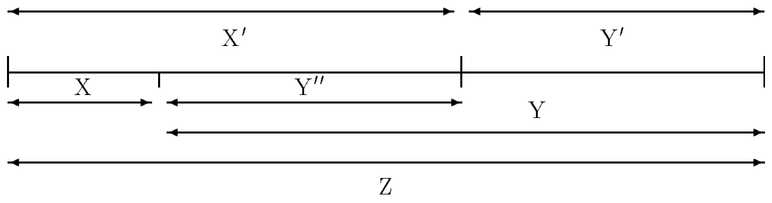

Figure 1.2 Composing substitutions

by the conjunction

$$\texttt{inst(}N_1\texttt{,}N'_1\texttt{)}, \ldots, \texttt{inst(}N_n\texttt{,}N'_n\texttt{)}$$

and partially evaluate the resulting definition of $\mathcal{R}/2n$ w.r.t. the definition for `inst/2`, i.e. expanding a call of the form `inst(N,N')` to the conjunction

$$\texttt{subst(N,Y,N')}, \texttt{atomic-open(Z)}.$$

Note that in this refinement we further assume that the *Frontier* represented by $F_1, \ldots, F_n$ is a maximal element (i.e. that the computation in question is *terminating*), and hence expand calls of the form `inst(N,Y,F)` in the second clause of `subst/2` to the conjunction `subst(N,Y,F)` , `atomic-close(Z)`. Such an assumption is not central to our argument but permits us to derive the termination case for terminating computations.

$$\begin{array}{l}
\mathcal{R}\texttt{(F}_1\texttt{-N}_1\texttt{,X}_1\texttt{,}\ldots\texttt{,F}_n\texttt{-N}_n\texttt{,X}_n\texttt{)} \leftarrow \\
\quad \texttt{subst(N}_1\texttt{,Y}_1\texttt{,N}'_1\texttt{)}, \texttt{atomic-open(Y}_1\texttt{)}, \ldots, \\
\quad \texttt{subst(N}_n\texttt{,Y}_n\texttt{,N}'_n\texttt{)}, \texttt{atomic-open(Y}_n\texttt{)}, \\
\quad \texttt{subst(N}_1\texttt{,X}_1\texttt{,F}_1\texttt{)}, \ldots, \texttt{subst(N}_n\texttt{,X}_n\texttt{,F}_n\texttt{)}, \\
\quad \texttt{subst(N}'_1\texttt{,X}'_1\texttt{,F}_1\texttt{)}, \ldots, \texttt{subst(N}'_n\texttt{,X}'_n\texttt{,F}_n\texttt{)}, \\
\quad \mathcal{R}\texttt{(F}_1\texttt{-N}'_1\texttt{,X}'_1\texttt{,}\ldots\texttt{,F}_n\texttt{-N}'_n\texttt{,X}'_n\texttt{)}
\end{array}$$

$$\begin{array}{l}
\mathcal{R}\texttt{(F}_1\texttt{-N}_1\texttt{,X}_1\texttt{,}\ldots\texttt{,F}_n\texttt{-N}_n\texttt{,X}_n\texttt{)} \leftarrow \\
\quad \texttt{subst(N}_1\texttt{,Y}_1\texttt{,F}_1\texttt{)}, \texttt{atomic-close(Y}_1\texttt{)}, \ldots, \\
\quad \texttt{subst(N}_n\texttt{,Y}_n\texttt{,F}_n\texttt{)}, \texttt{atomic-close(Y}_n\texttt{)}, \\
\quad \texttt{subst(N}_1\texttt{,X}_1\texttt{,F}_1\texttt{)}, \ldots, \texttt{subst(N}_n\texttt{,X}_n\texttt{,F}_n\texttt{)}
\end{array}$$

(6) Where possible remove X and Z to leave Y and a relation about Y :

(a) Recursive case: replace

```
subst(X,Y,Z),subst(X',Y',Z),subst(X,Y'',X')
```

by

```
subst(Y'',Y',Y)
```

Note that since `subst(X,Y,Z)` effectively describes *substitutions*, we can regard the above equivalence as describing the *composition of substitution*. The parameter `Y'` is the difference between `Y''` and `Y`, illustrated by figure 1.2.

(b) Base case: replace

$$\mathtt{subst(N_k,X_k,F_k)},\ \mathtt{subst(N_k,Y_k,F_k)}$$

by $\mathtt{X_k = Y_k}$

(7) Remove tuples $\mathtt{F_k}$-$\mathtt{N_k}$ ($1 \leq k \leq n$) throughout to derive the relation `computation/n`

```
computation(X1,...,Xn) ←
    subst(Y1,X'1,X1), atomic-open(X1), ...,
    subst(Yn,X'n,Xn), atomic-open(Xn),
    computation(X'1,...,X'n)

computation(X1,...,Xn) ←
    atomic-close(X1), ..., atomic-close(Xn)
```

Note that the number of clauses defining `computation/2` is related to the number of clauses in the definitions of `inst/2`, `atomic-open/1` and `atomic-close/1`. One or more *base-cases* are derived from the definitions of `inst/2` and `atomic-close/1` whereas the recursive cases are derived from the definitions of `inst/2` and `atomic-open/2`.

1.8 EXAMPLE — SUMMING THE ELEMENTS OF A LIST

1.8.1 The 'inst' relation

This example illustrates the derivation of the relation `sum-list(X, Y)` where `X` is a list of natural numbers and `Y` is the sum of the elements of `X`. Both `X` and `Y` are *closed* structures, i.e. all the members of `X` are themselves closed. We represent lists using the '•/2' functor and the constant `nil` to represent the empty list. Integers are represented by the `s/1` functor and `0` represents zero. Thus the list **1 2 3** is represented by the structure `s(0)•s(s(0))•s(s(s(0)))•nil`. Since the items in the list can represent natural numbers of any size, we require that `inst/2` be defined as atomic only for lists and not for natural numbers. We give the inst relation for summing the elements of a closed list as

```
inst((List,Sum),(List',Sum')) ←
    append(List,DiffList,List'),
    atomic-open-list(DiffList),
    sum(Sum,DiffSum,Sum'),
    DiffList = Item.Δ,
    constrain(DiffSum,Item)

inst((List,Sum),(List',Sum')) ←
    append(List,DiffList,List'),
    atomic-close-list(DiffList),
    sum(Sum,DiffSum,Sum'),
    atomic-close-sum(DiffSum)
```

```
constrain(Δ,0)
constrain(s(Sum),s(Sum′)) ←
    constrain(Sum,Sum′)

append(Δ,Y,Y)
append(A.X,Y,A.Z) ←
    append(X,Y,Z)

atomic-open-list(X.Δ)
atomic-close-list(nil)

sum(Δ,Y,Y)
sum(s(X),Y,s(Z)) ←
    sum(X,Y,Z)

atomic-close-sum(0)
```

Note that we may rewrite inst/2 more succinctly as:

```
inst((List,Sum),(List′,Sum′)) ←
    append(List,Item.Δ,List′),
    sum(Sum,DiffSum,Sum′),
    constrain(DiffSum,Item)

inst((List,Sum),(List′,Sum′)) ←
    append(List,nil,List′),
    sum(Sum,0,Sum′)
```

1.8.2 Deriving the logic program

We follow the method given in section 1.7 to derive the following definition of $\mathcal{R}$/2 in step 5:

```
𝓡(FrontierList-NodeList,List,FrontierSum-NodeSum,Sum) ←
    append(NodeList,List,FrontierList),
    sum(NodeSum,Sum,FrontierSum),
    append(NodeList,Item.Δ,NodeList′),
    sum(NodeSum,DiffSum,NodeSum′),
    constrain(DiffSum,Item),
    append(NodeList′,List′,FrontierList),
    sum(NodeSum′,Sum′,FrontierSum),
    𝓡(FrontierList-NodeList′,List′,FrontierSum-NodeSum′,Sum′)
```

```
R(FrontierList-NodeList,List,FrontierSum-NodeSum,Sum) ←
    append(NodeList,List,FrontierList),
    sum(NodeSum,Sum,FrontierSum),
    append(NodeList,nil,FrontierList),
    sum(NodeSum,0,FrontierSum)
```

Then following step 6, noting that if in the first clause

```
    append(NodeList,List,FrontierList),
    sum(NodeSum,Sum,FrontierSum),
    append(NodeList,Item.Δ,NodeList′),
    sum(NodeSum,DiffSum,NodeSum′),
    constrain(DiffSum,Item),
    append(NodeList′,List′,FrontierList),
    sum(NodeSum′,Sum′,FrontierSum)
```

holds then

```
List = Item.List′, sum(Item,Sum′,Sum)
```

holds and that in the second clause if

```
    append(NodeList,List,FrontierList),
    sum(NodeSum,Sum,FrontierSum),
    append(NodeList,nil,FrontierList),
    sum(NodeSum,0,FrontierSum)
```

holds then

```
List = nil , Sum = nil
```

holds. Thus we derive `sumlist/2` from $\mathcal{R}$`/4` by omitting the list pair and sum pairs:

```
sumlist(List,Sum) ←
    List = Item.List′,
    sum(Item,Sum′,Sum),
    sumlist(List′,Sum′)

sumlist(List,Sum) ←
    List = nil, Sum = 0
```

1.9 CONCLUSION

We have outlined a method for deriving logic programs which implement systems from the observations that can be made of such systems. Such observations can be regarded as *extrinsic specifications*, and can be described by directed acyclic graphs. Each computation that graph 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. Our method can be used to describe both sequential and concurrent systems; in [Gil90] we have illustrated the concurrent case. The computational rule incorporated in the interpreter for the derived logic programs dictates the characteristics of the system implemented by the execution of these programs. We have designed a concurrent logic language SILCS whose operational semantics permit the construction of *synchronous* concurrent systems, and constructed a prototype interpreter for this language [Gil91]. In [GH92] we have described the properties required of the `subst` relation which permit transformations of the type described in this paper; we have also been automating our transformations using the partial evaluator for Prolog developed by Dan Sahlin [Sah91]. Currently we are seeking to apply our method to the description of systems which construct general data structures rather than just lists, and at the same time we are extending our method to derive constraint logic programs from observations of constraint systems.

The algebraic structure atomic formula of atomic formulae, including the lattice properties of the instantiation order relation, were described independently in the early seventies by Reynolds [Rey70] and Plotkin [Plo70]. 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.

1.10 EXERCISES

1. Using the definition in this chapter for `append/3`, define the `inst/2` relation for a system whose observable variable is bound to a list.
2. Use the method described in section 1.7 to derive the relation `isa-list/1`, where `isa-list(List)` defines `List` to be a closed (ground) list.
3. Define the relation `inst/2` for a system whose two observable variables are bound to a list and an natural number representing the length of the list.
4. Derive the relation `length-list/2`, where `length-list(List,Length)` relates a closed (ground) list to its length expressed as a closed (ground) natural number.
5. Define the relation `inst/2` for a system whose two observable variables are bound to two lists, one of which is a sub-list of the other.
6. Using this definition, derive the relation `sub-list/2` where `sub-list(List,Sub)` describes the relationship between any closed list `List` and any closed sublist `Sub` of that list.

1.11 ACKNOWLEDGMENTS

Thanks are due to Professor Ernest Edmonds of Loughborough University of Technology, whose support while Head of the Department of Computer Studies gave David Gilbert the time and resources needed to complete the PhD on which the work described in this chapter is based.

REFERENCES

[GH92] D.R. Gilbert and C.J. Hogger. Logic for Representing and Implementing Knowledge about System Behaviour. In V.editors, *Advanced Topics in Artificial Intelligence*, International Summer School, pages 42–49, Prague, Czechoslovakia, July 1992. Springer-Verlag.

[Gil90] D.R. Gilbert. *Specifying and Reasoning about Concurrent Systems in Logic*. PhD thesis, Department of Computing, Imperial College, London, United Kingdom, September 1990.

[Gil91] D.R. Gilbert. SILCS — Specification In Logic of Concurrent Systems. Preconference workshop on Concurrent Logic Programming, International Conference on Logic Programming, June 1991.

[Plo70] G.D. Plotkin. A Note on Indictive Generalization. *Machine Intelligence*, 5:153–165, 1970.

[Rey70] J.C. Reynolds. Transformational Ssystems and the Algebraic Structure of Atomic Formulas. *Machine Intelligence*, 5:135–151, 1970.

[Sah91] D. *An Automatic Partial Evaluator for Full Prolog*. PhD thesis, Swedish Institute of Computer Science, Kista, Sweden, March 1991.