Tutorial Notes: Reasoning about Logic Programs*

Alan Bundy
bundy@edinburgh.ac.uk

Department of Artificial Intelligence
University of Edinburgh

1 Introduction

In this tutorial we will describe techniques for reasoning about logic programs.

Why should we want to reason about logic programs?

We will see that reasoning about programs is an important software engineering tool. It can be used to improve the efficiency and reliability of computer programs. Such reasoning is particularly well suited to logic programs. In fact, the ease of reasoning with logic programs is one of their main advantages over programs in other languages.

The kind of reasoning tasks we will consider are as follows.

Verification: to verify that a program meets a specification of its behaviour.
Synthesis: to synthesise a program that meets a specification.
Transformation: to transform a program into a more efficient program meeting the same specification.
Termination: to show that a run of a program will always terminate.
Abstraction: to abstract from the program information about the types of its input/output, its modes of use, *etc.*

In conventional, imperative, programming languages these four tasks are very different, but in logic programming some of them merge together. The reason is that specifications are usually written as logical formulae. As such, they can be interpreted as logic programs. They may be very inefficient and they may not always terminate, but they can often be used as a prototype of the desired program. This means that synthesis and transformation are not different in kind, but only in degree. Moreover, synthesis can be seen as verification of a partially specified program. All this imparts a simplicity and unity to reasoning with logic programs.

* I would like to thank the members of the mathematical reasoning group at Edinburgh and members of the CompuLog Network for helpful advice and feedback on these notes. In particular, Ina Kraan, Andrew Ireland, Helen Lowe, Danny De Schreye and Michael Maher were especially helpful. Special thanks are due to Ian Green and Andrew Ireland for getting the Springer LaTeX style file to work. Some of the research reported in this paper was supported by Esprit BRA grant 3012 (CompuLog).

1.1 Semantics of Logic Programs

To reason formally about a computer program we must have a method of turning a question about programs into a mathematical conjecture. The program reasoning task can then be converted into a theorem proving task. The usual way to do this is to associate a *semantics* with the programming language. By 'semantics' we mean an assignment of a mathematical expression to each program statement. The mathematical expression is often thought of as the *meaning* of the program statement.

To a first approximation it is unnecessary to give a semantics to a logic program; it is already a mathematical expression, namely a clause in first-order logic. Unfortunately, this is only true of *pure* logic programs. In practice, logic programs, *e.g.* Prolog programs, often contain impure features, *e.g.* negation as failure, `assert/retract`, `var`, the cut operator, the search strategy, *etc.* These cannot be *directly* interpreted in logical terms.

There are (at least) three solutions to this problem.

1. Provide a semantics for the particular logic programming language, *e.g.* Prolog, in which both the pure and impure features of the language are assigned a mathematical interpretation.
2. Work only within a pure subset of the language. Implement a logic programming language based on this pure subset.
3. Reason about specifications of logic programs. Introduce the impure features as necessary during a final compilation of the specification into your logic programming language of choice.

We will discuss the tradeoffs between these approaches in §2.

1.2 Specifications of Logic Programs

Specifications of programs are usually given as logical relations between input and output. For instance, a sort program takes in a list and outputs another list in which the same elements are put in order, *i.e.* the output is an ordered permutation of the input. This program can be specified by the relation[1]:

$$perm(IL, OL) \ \&\ ordered(OL)$$

where IL is the input list, OL is the output list and $perm$ and $ordered$ are defined appropriately.

Suppose `sort1` is a logic program for sorting lists, then to verify `sort1` we must prove as a theorem:

$$\forall IL \in list, \forall OL \in list.\ sort1(IL, OL) \leftrightarrow perm(IL, OL) \ \&\ ordered(OL) \quad (1)$$

This specification of `sort1` can itself be readily 'compiled' into a prototype sorting program `sort2`. For instance, in Prolog we could write:

[1] We will adopt the convention of using typewriter font for programs and maths font for specifications.

```
sort2(IL,OL) :- perm(IL,OL), ordered(OL).
```

The program `sort2` would be very inefficient — it would generate each permutation in turn and then test to see if it was ordered — but it would work. Now the proof of theorem (1) can be re-interpreted as a process of transformation of a very inefficient sorting program, `sort2` into a more efficient program `sort1` meeting the same specification.

By a specification of a program we will mean a formula of the form:

$$\forall \boldsymbol{Args} \in \boldsymbol{types}.\ head(\boldsymbol{Args}) \leftrightarrow body(\boldsymbol{Args})$$

where:

- *head* is the name of the program being specified;
- $\boldsymbol{Args}$ is some sequence of distinct variables and $\boldsymbol{types}$ is a pairwise corresponding sequence of their types; and
- $body(\boldsymbol{Args})$ is an arbitrary first-order logic formula, with no defined functions, whose free variables are in $\boldsymbol{Args}$.

In the interests of readability we will sometimes reduce clutter by omitting the $\forall \boldsymbol{Args} \in \boldsymbol{types}$ part.

The condition excluding defined functions from the body means that only undefined or constructor functions are allowed. That is, we can use functions to form data-structures, *e.g.* $s(0)$ or $[Hd|Tl]$, but not to define relations between objects, *e.g.* $x + y$.

1.3 Partial Evaluation of Logic Programs

This process of verifying/transforming logic programs can be regarded in two ways:

- as the proof of a mathematical theorem (*e.g.* (1)) using the definitions of the programs and specifications as axioms; or
- the symbolic and partial execution of the logic program.

Because of the second interpretation this process it is often called *partial evaluation*. We will see that to reason with recursive logic programs we will have to add *mathematical induction* to partial evaluation. Because we use induction we must work with a typed logic, *cf.* theorem (1) above in which $\forall IL \in list$ means for all objects, IL, of type *list*.

1.4 Abstract Interpretation of Logic Programs

It is also possible to reason with abstractions of logic programs, *i.e.* with programs in which some details of the program are generalised. For instance, we can reason with an abstraction in which the parameters to each procedure are replaced with labels representing their types or with their input/output mode, [Bruynooghe & De Schreye 88]. The purpose of such reasoning is to deduce the implied type or mode of some parameters from others. For instance, suppose we know that the parameters of our `sort2` program are both of type list. Its definition can be abstracted to:

```
sort2(list,list) :- perm(list,list), ordered(list).
```

From which we can deduce that the parameters of `perm` and `ordered` are also all lists (or some more general type).

The method of reasoning with abstract programs is the same as that used for concrete programs, *e.g.* partial evaluation, but the reasoning tends to be simpler because of the loss of detail caused by abstraction.

2 What Shall we Reason About?

In this section we discuss the tradeoffs between reasoning with impure programs, pure programs or specifications. There are advantages and disadvantages with each approach.

2.1 Incomplete Information

A disadvantage of reasoning directly with either pure or impure logic programs is that their declarative meaning does not quite correspond with their intended meaning. To see why not consider, for instance, the Prolog test for list membership, `member/2`.

```
member(El,[El|Tl]).
member(El,[Hd|Tl]) :- member(El,Tl).
```

Note that this program says nothing about the case when the list is empty. If you try to find out about this case by calling the goal:

```
?- member(X,[]).
```

then you will find that it fails. The standard interpretation of this failure is to say that `member(X,[])` is false for all `X`. But this is to go beyond the declarative meaning of the `member/2` program. It is to adopt the *closed world assumption*: anything that is not true is false.

One way to formalise this assumption is to say that the meaning of a logic program is not the immediate declarative meaning of its clauses, but the *Clark completion*. To form the Clark completion of the set of clauses defining a predicate we replace them all with a single equivalence. This equivalence states that a program head is true if *and only if* its body is true. For instance, the Clark completion of the `member/2` program is[2]:

$$\begin{aligned} member(El, L) \leftrightarrow\ & (\exists Tl.\ L = [El|Tl]) \vee \\ & (\exists Hd, Tl.\ L = [Hd|Tl]\ \&\ member(El, Tl)) \end{aligned}$$

Note the following general properties of Clark completions.

[2] Maths font is used here because we intend to use typed versions of Clark completions as the specifications of the programs they complete.

- All the clauses for `member/2` are replaced with one equivalence. Each of the old clauses corresponds to one disjunct in the body of this new equivalence.
- The head of the new equivalence consists of the predicate with distinct variables as its arguments. The head arguments in the old clauses are represented by equalities in each disjunct between these old arguments and the new variables, *e.g.* $L = [El|Tl]$.
- Each variable that appears in the body of the equivalence but not the head is existentially quantified in the body, *e.g.* Hd and Tl.

When reasoning about logic programs it is sometimes necessary to use this missing information. It is, therefore, more convenient to reason directly with the Clark completions than with the programs themselves.

A further piece of missing information is the types of the expressions in the programs. Consider the goal clause:

```
?- member(X,42).
```

We normally want to regard this call, not as false, but as ill-formed. Furthermore, if we want to reason using mathematical induction it will be vital to circumscribe the types of argument that a program can take. To do this we must add type declarations to our language. For instance, we might declare the type of *member* as:

$$member \in Type \times list(Type)$$

This is to be read as saying that *member* can take objects of any type as its first argument but must take lists of objects of this type as its second argument.

The types of the variables in an equivalence can be declared as $Obj \in Type$ statements within the quantifier declarations. For instance, we can enrich the Clark completion of `member/2` as follows.

$$\begin{array}{l} \qquad\qquad \forall El \in Type, L \in list(Type). \\ member(El, L) \leftrightarrow (\exists Tl \in list(Type).\ L = [El|Tl]) \vee \\ \qquad\qquad (\exists Hd \in Type, Tl \in list(Type).\ L = [Hd|Tl]\ \&\ member(El, Tl)) \end{array}$$

Note that this extended version of the Clark completion of `member/2` fits the logical form of a program specification as defined in §1.2. In fact, we can use an extended Clark completion as the specification of the program it completes. We can reason directly with extended Clark completions, compiling them into their corresponding programs when the reasoning is complete. In this way we can reason with complete information and only 'forget' it when it is no longer required. This provides an argument for reasoning at the specification level.

Notation Henceforth, we will use the term *completion* to mean extended Clark completion.

2.2 What does ‘Equivalent’ Mean?

The simplest kind of reasoning with programs is to transform one program into an equivalent one. Unfortunately, there are many rival senses of ‘equivalent’. A transformation that is legal under one sense of ‘equivalent’ may be illegal under another.

Impure Logic Programs Consider, for instance, the Prolog clauses:

```
p1(X) :- q(X), r(X).
p2(X) :- r(X), q(X).
q(a).
r(b) :- r(X).
r(a).
```

The programs `p1` and `p2` differ only in the order of the two literals in their body. Logically these literals are conjoined. Thus logically the two programs are equivalent. Unfortunately, they have very different operational behaviour. The goal `p1(X)` will succeed with `X=a` and stop, but the goal `p2(X)` will loop for ever with no success. Thus `p1` and `p2` are logically equivalent but operationally different.

The operational differences are due to the Prolog search strategy. It always evaluates literals left to right and applies clauses top to bottom. If it evaluated clauses bottom to top then `p2(X)` would also succeed with `X=a`, but would then backtrack and loop for ever. If it evaluated literals right to left then it would be `p2(X)` that would succeed with `X=a` and `p1(X)` that would loop for ever.

The most popular form of program transformation is partial evaluation. For instance, we can transform the program `p1` by partially evaluating the goal `p1(X)`. We resolve the literal `q(X)` with the unit clause `q(a)` and then the literal `r(a)` with the unit clause `r(a)`. This transforms the clause `p1(X) :- q(X), r(X).` into the logically and operationally equivalent clause `p1(a).` We can apply the same process to the program `p2` to transform clause `p2(X) :- r(X), q(X).` into `p2(a).` However, in this case we get a clause that is logically but not operationally equivalent. For instance, `p2(X)` will not now loop for ever.

If we want to preserve the operational behaviour of Prolog programs then we must restrict partial evaluation to use the same search strategy as Prolog. This will permit the above partial evaluation of `p1` but prevent the one of `p2`. To partially evaluate the clause for `p2` we must resolve the literals in its body in left/right order. The literal `r(X)` can be resolved with either of the two clauses for `r`. This produces the partially evaluated clauses:

```
p2(b) :- r(X), q(b).
p2(a) :- q(a).
```

If we continue to partially evaluate the first clause then we will get into a loop, so we should stop now. The second clause can be further partially evaluated to `p2(a)`.

Such a restriction throws away a major reason for using logic programs: their logical semantics. It also makes the reasoning machinery dependent on the details of the Prolog interpreter. Small 'enhancements' of the interpreter might render the reasoning machinery obsolete.

Pure Logic Programs One reaction to this problem is to define conditions that a pure logic programming language must fulfil and then assume these conditions are met when designing reasoning machinery. One such condition is to insist on a *fair* search strategy, *i.e.* that each node in the search space is considered at some point during search. Prolog's search strategy is not fair; it can easily get trapped down an infinite branch of the search space and never return, *cf.* `p2(X)` above. This means that we must reject Prolog as an implementation of logic programming and confine ourselves to more restrictive implementations.

Unfortunately, even if we restrict ourselves to pure logic programs, there are still rival notions of equivalence. For instance, [Maher 87] considers ten different definitions of equivalence and shows that only two pairs of these give the same notion. This leaves eight distinct notions of equivalence. Consider, for instance, the following pure logic programs[3] from [Maher 87].

$$\begin{aligned} &even1(0) \\ even1(s(s(N))) &\leftarrow even1(N) \\ &odd1(s(0)) \\ odd1(s(s(N))) &\leftarrow odd1(N) \end{aligned}$$

$$\begin{aligned} &even2(0) \\ even2(s(N)) &\leftarrow odd2(N) \qquad (2) \\ odd2(s(N)) &\leftarrow even2(N) \end{aligned}$$

At first sight the definitions of $even1/odd1$ and $even2/odd2$ look like alternative, but equivalent programs for testing for even and odd numbers. [The natural numbers are represented in unary notation, where, for instance, 3 is represented by $s(s(s(0)))$.]

However, these programs are not logically equivalent, *i.e.* it is not the case that:

$$\begin{aligned} even1(N) &\leftrightarrow even2(N) \\ odd1(N) &\leftrightarrow odd2(N) \end{aligned}$$

To see this consider the clause:

$$even2(s(0)) \leftarrow odd2(0) \tag{3}$$

This clause is a logical consequence of the definitions of $even2/odd2$ — it is an instance of clause (2) above. If the two programs were equivalent then the

[3] We will use maths font for pure logic programs.

implication:

$$even1(s(0)) \leftarrow odd1(0) \tag{4}$$

would also be a logical consequence of the definitions — but it is not. To see this consider the non-standard model in which $even1(N)$ is true for all even numbers N, but $odd1(N)$ is true for *all* numbers, both odd and even. In this model all the clauses of the program are true, but clause (4) is false.

However, there is a notion of equivalence in which these two definitions are equivalent: their completions are logically equivalent[4].

$$\begin{aligned} even1(N) &\leftrightarrow N = 0 \vee \exists M \in nat.\ N = s(s(M))\ \&\ even1(M) \\ odd1(N) &\leftrightarrow N = s(0) \vee \exists M \in nat.\ N = s(s(M))\ \&\ odd1(M) \end{aligned}$$

$$\begin{aligned} even2(N) &\leftrightarrow N = 0 \vee \exists M \in nat.\ N = s(M)\ \&\ odd2(M) \\ odd2(N) &\leftrightarrow \exists M \in nat.\ N = s(M)\ \&\ even2(M) \end{aligned}$$

Both clauses (3) and (4) are logical consequences of this definition because $odd1(0) \leftrightarrow odd2(0) \leftrightarrow false$.

Thus, whether we regard these two programs as equivalent depends on whether we take logical equivalence of programs, logical equivalence of completions or one of the other six rival notions, as the definition of equivalence.

2.3 Summary

We can summarise these different tradeoffs as follows.

Impure Programs

Pros It is possible to reason directly with existing programs.

Cons There are a wide variety of different transformation schemes depending on what properties of the program it is desired to preserve. A different semantics is required to define each kind of preservation and to justify the corresponding transformations. These transformations are very complex and restricted. The semantics and the transformations are sensitive to small changes in the definition of the programming language. The logical and intended meaning of the program do not coincide. This means that some information required to reason with the programs is not represented directly.

Pure Programs

[4] Our use of 'logically equivalent' is non-standard in that we use more than just rules of logic in proofs of equivalence; we also use mathematical induction.

Pros The kinds of transformation are general across a wide range of programming languages.

Cons There are still several different notions of program equivalence and, hence, several different transformation schemes. These transformations do not preserve the operational behaviour of impure programs, *e.g.* Prolog programs. The mismatch between the logical and intended meaning of the program causes problems during reasoning.

Specifications

Pros There is only one notion of equivalence. The logical and intended meaning of the specification coincide.

Cons Specifications of programs must be available for transformation. A specification can be 'compiled' into alternative programs which are not operationally equivalent.

Conclusion It seems to us that the benefits of logic programs can best be realised by a combination of reasoning with specifications and with impure programs. The basic algorithm is best determined by reasoning at the specification level, where the notion of equivalence is unambiguous and the reasoning invariant under changes in programming language. However, various implementational aspects can only be dealt with at the programming language level, so some tuning transformations at this level must also be catered for. The rest of these tutorial notes will assume this viewpoint.

3 Logic Specifications and Programs

Adopting this viewpoint makes it vital that we can translate freely between specifications and the programs they specify. In this section we describe two algorithms: one for lifting programs into specifications and one for compiling specifications into programs.

3.1 From Programs to Specifications

In this section we describe an algorithm for translating a logic program into a specification of itself. We will call this the *Clark completion algorithm.* It works by constructing the program's completion and using this as the specification.

Definition 1 (The Clark Completion Algorithm) *The algorithm consists of four stages.*

1. *Rewrite the program into logical notation,* e.g. *for Prolog clauses turn* `:-` *into* $\leftarrow$*s, commas into* $\&$*s, semi-colons into* $\vee$*s, negation as failure into* $\neg$*s, provide definitions for system predicates,* etc.

2. *Make the head arguments into distinct variables. This is done by replacing each clause of the form:*

$$p(s_1, \ldots, s_n) \leftarrow body$$

with the clause:

$$p(X_1, \ldots, X_n) \leftarrow X_1 = s_1 \ \& \ \ldots \ \& \ X_n = s_n \ \& \ body$$

where the X_i are new variables. The same X_is are used for each clause for p.
Note that if s_i is already a variable distinct from s_j for $j < i$ then rather than add $X_i = s_i$ to the body we can replace s_i by X_i throughout. In practice, we will adopt this option when possible, since it will lead to simpler completions.
3. *Existentially quantify each variable which occurs in the body but not in the head. Replace each clause of the form:*

$$p(X_1, \ldots, X_n) \leftarrow body(Y_1, \ldots, Y_m)$$

where $X_i \neq Y_j$, for all i and j, and Y_j occurs in $body(Y_1, \ldots, Y_m)$, with the clause:

$$p(X_1, \ldots, X_n) \leftarrow \exists Y_1 \in t_1, \ldots, \exists Y_m \in t_m.\ body(Y_1, \ldots, Y_m)$$

where t_i is the type of Y_i. We postpone the problem of discovering these types to §9.
4. *Combine the clauses for each predicate into a single equivalence. Suppose there are k clauses defining p, each of the form:*

$$p(X_1, \ldots, X_n) \leftarrow body_i$$

We replace these k clauses with:

$$p(X_1, \ldots, X_n) \leftrightarrow body_1 \vee \ldots \vee body_k$$

This algorithm is adapted from [Lloyd 87][§14], in which further details may be found.

Example: Constructing the completion of *subset* Here is a simple example of the algorithm. Consider the following program for `subset/2`.

```
subset([],J).
subset([Hd|Tl],J) :- member(Hd,J), subset(Tl,J).
```

A goal of the form `subset(I,J)` succeeds if `I` is a subset of `J`, where sets are represented as lists.

The first stage of the algorithm is to rewrite the program into logical notation.

$$subset([\,], J)$$
$$subset([Hd|Tl], J) \leftarrow member(Hd, J) \ \& \ subset(Tl, J)$$

The second stage is to make the head arguments into distinct variables.

$$\begin{aligned}
&subset(I, J) \leftarrow I = [\,] \\
&subset(I, J) \leftarrow I = [Hd|Tl] \;\&\; member(Hd, J) \;\&\; subset(Tl, J)
\end{aligned}$$

Note that in both clauses we exercised our option not to replace J.

The third stage is to existentially quantify each variable which occurs in the body but not in the head.

$$\begin{aligned}
&subset(I, J) \leftarrow I = [\,] \\
&subset(I, J) \leftarrow \exists Hd \in Type, \exists Tl \in list(Type). \\
&\qquad\qquad\qquad\qquad I = [Hd|Tl] \;\&\; member(Hd, J) \;\&\; subset(Tl, J)
\end{aligned}$$

The fourth stage is to combine these two clauses into a single equivalence.

$$\begin{aligned}
&subset(I, J) \leftrightarrow I = [\,] \vee \\
&\qquad\qquad\quad \exists Hd \in Type, Tl \in list(Type). \\
&\qquad\qquad\qquad\qquad I = [Hd|Tl] \;\&\; member(Hd, J) \;\&\; subset(Tl, J)
\end{aligned}$$

3.2 From Specifications to Programs

In this section we describe an algorithm for compiling a specification into the logic program it directly specifies.

One way to make such a compilation algorithm would be to reverse the Clark completion algorithm. Unfortunately, this would not be a general-purpose compilation algorithm. This is because the class of specifications is much larger than the class of completions. For instance, the equivalence:

$$subset(I, J) \leftrightarrow (\forall El \in Type.\ member(El, I) \rightarrow member(El, J)) \tag{5}$$

is a specification, but is not a completion. Our algorithm must cover these non-completions too.

Our algorithm will compile any specification. We will call it the *Lloyd-Topor compilation algorithm*.

Definition 2 (The Lloyd-Topor Compilation Algorithm) *The key idea of the algorithm is to put the specification into clausal form. It consists of three stages.*

1. *Turn the main $\leftrightarrow$ into a $\leftarrow$.*
2. *Put the specification into clausal form using the Lloyd-Topor rules. They are too complicated to give in full here. We have illustrated the general idea by giving some examples in figure 3.2. The complete set is given in [Lloyd 87][p113].*

Name	Input Clause	Output Clause(s)
$\exists$	$head \leftarrow \ldots \ \& \ \exists \boldsymbol{X} \in \boldsymbol{t}.\ body \ \& \ \ldots$	$head \leftarrow \ldots \ \& \ body \ \& \ \ldots$
$\vee$	$head \leftarrow \ldots \ \& \ (a \vee b) \ \& \ \ldots$	$head \leftarrow \ldots \ \& \ a \ \& \ \ldots$ $head \leftarrow \ldots \ \& \ b \ \& \ \ldots$
$\neg \rightarrow$	$head \leftarrow \ldots \ \& \ \neg(a \rightarrow c) \ \& \ \ldots$	$head \leftarrow \ldots \ \& \ a \ \& \ \neg c \ \& \ \ldots$
$\forall$	$head \leftarrow \ldots \ \& \ \forall \boldsymbol{X} \in \boldsymbol{t}.\ body \ \& \ \ldots$	$head \leftarrow \ldots \ \& \ \neg\exists \boldsymbol{X} \in \boldsymbol{t}.\ \neg body \ \& \ \ldots$
$\neg\exists$	$head \leftarrow \ldots \ \& \ \neg\exists \boldsymbol{X} \in \boldsymbol{t}.\ body \ \& \ \ldots$	$head \leftarrow \ldots \ \& \ \neg q(Y_1, \ldots, Y_k) \ \& \ \ldots$ $q(Y_1, \ldots, Y_k) \leftarrow body$

In the last rule, q is a new predicate symbol and the Y_i are the free variables in $\exists \boldsymbol{X} \in \boldsymbol{t}.\ body$.
These rules are applied as rewrite rules to the specification until no more apply. The specification is then in clausal form.

Fig. 1. Selected Lloyd-Topor Transformation Rules

3. *Turn the logical symbols into program symbols,* i.e. *invert stage 1 of the Clark completion algorithm.*

We can now see the need for the restriction, given in §1.2, to exclude defined functions from specifications. There is no provision in the Lloyd-Topor algorithm to turn these defined functions into predicate definitions. Consider, for instance:

$$plus(X, Y, Z) \leftrightarrow (X = 0 \ \& \ Z = Y) \vee \\ \exists X' \in nat.\ X = s(X') \ \& \ Z = s(X' + Y)$$

Lloyd-Topor compiles this into the Prolog program:

```
plus(X,Y,Z) :- X=0, Z=Y.
plus(X,Y,Z) :- X=s(X'), Z=s(X'+Y).
```

But this program will fail for non-zero `X` because `X'+Y` is undefined. There is no provision in Prolog to define it as a function nor in Lloyd-Topor to compile it into a predicate. This problem could be solved by a suitable modification of the Lloyd-Topor algorithm.

The Lloyd-Topor algorithm is a near inverse of the Clark completion algorithm.

– Stage 4 of Clark completion is inverted by Stage 1 and rule $\vee$ of Lloyd-Topor.
– Stage 3 of Clark completion is inverted by rule $\exists$ of Lloyd-Topor.
– Stage 1 of Clark completion is inverted by stage 3 of Lloyd-Topor.

We could also invert stage 2 of Clark completion by introducing an additional rule into stage 2 of Lloyd-Topor which removed $X_i = s_i$ literals from the body be replacing X_i by s_i in the head. Unfortunately, it would be impossible to prevent this rule from removing X_i literals that were present in the original program. Similarly, rule $\vee$ can remove disjunctions that were present in the original program, if the programming language allows them.

Note that Clark completion is not an inverse of Lloyd-Topor, since the original specification may not be a completion.

Example: Translating a *subset* specification into a program We will illustrate the Lloyd-Topor algorithm on the specification of *subset*, (5) above. The first stage is to turn the $\leftrightarrow$ into a $\leftarrow$.

$$subset(I, J) \leftarrow (\forall El \in Type.\ member(El, I) \rightarrow member(El, J))$$

The second stage is to use the Lloyd-Topor compilation to put this into clausal form. The rules from figure 3.2 apply as follows:

$$subset(I, J) \leftarrow \neg\exists El \in Type.\ \neg(member(El, I) \rightarrow member(El, J))$$

$$\begin{aligned} subset(I, J) &\leftarrow \neg not\_subset(I, J) \\ not\_subset(I, J) &\leftarrow \neg(member(El, I) \rightarrow member(El, J)) \end{aligned}$$

$$\begin{aligned} subset(I, J) &\leftarrow \neg not\_subset(I, J) \\ not\_subset(I, J) &\leftarrow member(El, I)\ \&\ \neg member(El, J) \end{aligned}$$

The third stage is to rewrite the clauses into program notation.

```
subset(I,J) :- not not_subset(I,J).
not_subset(I,J) :- member(El,I), not member(El,J).
```

4 Equivalence of Specifications

The problem we will consider in this section is how to prove that two logic program specifications are equivalent. That is, suppose $Spec_1$ and $Spec_2$ are two logic program specifications. We will consider how to prove:

$$\forall \boldsymbol{Args} \in \boldsymbol{Types}.\ Spec_1 \leftrightarrow Spec_2 \qquad (6)$$

In the interests of readability we will sometimes omit the $\forall \boldsymbol{Args} \in \boldsymbol{Types}$ part.

We saw in §3 that each of these two specifications compiles directly into a logic program. Some of these logic programs are more practical than others. Suppose $Prog_1$ is the logic program corresponding to $Spec_1$ and $Prog_2$ to $Spec_2$. If $Prog_2$ is an impractical program and $Prog_1$ is a practical program then we can view the proof of equivalence (6) as verification of $Prog_1$ in terms of $Spec_2$. If both $Prog_1$ and $Prog_2$ are practical programs then we can view this proof as the transformation of $Prog_2$ into $Prog_1$. Typically, $Prog_1$ will be more efficient than $Prog_2$.

Thus, when reasoning at the level of specifications, the processes of verification and transformation coalesce; they differ in degree not kind. Of course, the real challenge in transformation is to be given an inefficient program, $Prog_2$, and to construct a more efficient program, $Prog_1$. This is essentially the same problem as synthesising a practical program, $Prog_1$, which meets a given specification, $Spec_2$. We will tackle this joint problem in §5 below.

Example: Equivalence of two *subset* definitions Here is a simple example. Consider the verification theorem:

$$\forall I \in list(Type), J \in list(Type). \qquad subset(I,J) \leftrightarrow (\forall El \in Type.\ member(El,I) \rightarrow member(El,J)) \tag{7}$$

where *subset* and *member* are defined by the specifications:

$$\begin{aligned} subset(I,J) \leftrightarrow\ & I = [\,] \vee \\ & \exists Hd \in Type, Tl \in list(Type). \\ & \quad I = [Hd|Tl]\ \&\ member(Hd,J)\ \&\ subset(Tl,J) \\ member(El,L) \leftrightarrow\ & (\exists Hd \in Type, Tl \in list(Type).\ L = [El|Tl]) \vee \\ & (\exists Hd \in Type, Tl \in list(Type).\ L = [Hd|Tl]\ \&\ member(El,Tl)) \end{aligned}$$

The quantification of the outer universal variables has been omitted to reduce clutter.

Using the Lloyd-Topor compilation, the left and right hand sides of equivalence (7) compile into the two Prolog programs:

```
subset_1([],J).
subset_1([Hd|Tl],J) :- member(Hd,J), subset_1(Tl,J).

subset_2(I,J) :- not not_subset(I,J).
not_subset(I,J) :- member(El,I), not member(El,J).
```

The details of these compilations were given in §3. Program `subset_1/` will run in any input mode, but `subset_2` will flounder in all input modes except `subset_2(+,+)`. The proof of equivalence (7) can thus be regarded as an exercise in verification of `subset_1`.

To prove (7) we must use induction on the recursive structure of lists with I as the induction variable. This generates a base case and a step case.

The base case is obtained from (7) by substituting the empty list for I.

$$subset([\,],J) \leftrightarrow (\forall El \in Type.\ member(El,[\,]) \rightarrow member(El,J))$$

Applying the definitions of *subset* and *member* this reduces to the problem of proving:

$$true \leftrightarrow (\forall El \in Type.\ false \rightarrow member(El,J))$$

which further reduces to $true$, using the rules of predicate logic.

The step case consists of an induction conclusion which can be proved with the aid of an induction hypothesis. The induction hypothesis and conclusion are obtained from (7) by substituting tl and $[hd|tl]$, respectively, for I.

$$\begin{aligned}&subset(tl, J) \leftrightarrow (\forall El \in Type.\ member(El, tl) \rightarrow member(El, J))\\ &\vdash subset([hd|tl], j) \leftrightarrow (\forall El \in Type.\ member(El, [hd|tl]) \rightarrow member(El, j))\end{aligned}$$

The turnstile symbol $\vdash$ indicates that the induction hypothesis on the left can be assumed when proving the induction conclusion on the right. Note that the universal variable J in the induction hypothesis can be instantiated, if necessary, whereas it should not be instantiated in the induction conclusion. We have ensured this by representing J by a free variable (upper case) on the left and a constant j (lower case) on the right.

Using the definitions of *subset* and *member* the induction conclusion can be reduced to:

$$\begin{aligned}&member(hd, j)\ \&\ subset(tl, j)\\ &\leftrightarrow (\forall El \in Type.\ (El = hd \lor member(El, tl)) \rightarrow member(El, j))\end{aligned}$$

which, using the rules of predicate logic, can be rewritten as follows:

$$\begin{aligned}member(hd, j)\ \&\ subset(tl, j) &\leftrightarrow (\forall El \in Type.\ (El = hd \rightarrow member(El, j))\ \&\\ &\qquad\qquad (member(El, tl) \rightarrow member(El, j)))\\ member(hd, j)\ \&\ subset(tl, j) &\leftrightarrow ((\forall El \in Type.\ El = hd \rightarrow member(El, j))\ \&\\ &\quad\ (\forall El \in Type.\ member(El, tl) \rightarrow member(El, j)))\\ member(hd, j) &\leftrightarrow ((\forall El \in Type.\ El = hd \rightarrow member(El, j))\\ member(hd, j) &\leftrightarrow member(hd, j)\\ &true\end{aligned}$$

This completes the proof of the step case. The whole of equivalence (7) is now proved.

5 Synthesis of Specifications

The problem we will consider in this section is how, given an initial specification, we can synthesise an equivalent new one. That is, suppose we are given a specification $Spec_2$, how can we construct a specification $Spec_1$ such that:

$$\forall \boldsymbol{Args} \in \boldsymbol{Types}.\ Spec_1 \leftrightarrow Spec_2$$

As discussed in §4 above, this will enable us to synthesise a practical program, $Prog_1$, from a specification $Spec_2$ and/or to transform an inefficient program, $Prog_2$, into an efficient program, $Prog_1$.

The essential idea underlying our solution to this problem will be to proceed with the proof of the equivalence theorem as if $Spec_1$ were known, and to pick up clues as to its definition as we proceed. This is best illustrated with an example.

Example: Synthesis of *subset* We will repeat the proof of equivalence (7), given in §4 above, but with *subset* left undefined. This will leave us with some unprovable subgoals, which we can use to form the definition of *subset*.

We repeat below the equivalence to be proved.

$$subset(I, J) \leftrightarrow (\forall El \in Type.\ member(El, I) \rightarrow member(El, J))$$

Many of the steps in the previous proof of this equivalence can be repeated, but the proof cannot be completed due to the absence of the definition of *subset*. The residue of subgoals is:

$$\begin{aligned} subset([\,], J) &\leftrightarrow true \\ subset([hd|tl], J) &\leftrightarrow member(hd, J)\ \&\ subset(tl, J) \end{aligned}$$

This residue can be used to suggest a definition of *subset*. We regard the residue as a logic program and take its completion, which we then adopt as the definition. In this case the definition this suggests is:

$$\begin{aligned} subset(I, J) \leftrightarrow I = [\,] \lor \\ &\exists Hd \in Type, Tl \in list(Type). \\ &\quad I = [Hd|Tl]\ \&\ member(Hd, J)\ \&\ subset(Tl, J) \end{aligned}$$

as required. The residue of subgoals can be readily proved from this definition, so the proof is completed.

6 Automated Theorem Proving

The equivalences between specifications are usually straightforward to prove. However, it does require some facility with mathematical ideas to find these proofs. It is desirable to automate this proof discovery process as much as possible to reduce the burden on the program developer. In this section we discuss how this can be done.

The main technical problem in automated theorem proving is guiding the search for a proof. It is easy to represent the theorem and the rules for manipulating it. It is particularly easy in a logic programming language since the necessary data-structures are provided directly. It is also easy to write a program to apply these rules exhaustively until the required proof is found. Unfortunately, for all but trivial theorems, this process rapidly becomes bogged down in an explosion of partially generated proofs. This phenomenon is called the *combinatorial explosion*.

To defeat the combinatorial explosion we need to use heuristic methods to guide the proof building process along the most promising paths. To illustrate such heuristic methods, consider the problem of rewriting the induction conclusion so that the induction hypothesis may be applied to it. The form of the step case of an inductive proof is:

$$P(tl) \vdash P(\boxed{[hd|\underline{tl}]}^{\uparrow})$$

Note that the induction conclusion on the right differs from the induction hypothesis on the left by inclusion of the induction term $[hd|\ldots]$. We have emphasised this by drawing a box around the induction term and underlining the induction variable, tl inside it. We call this boxed sub-expression a *wave-front*. The arrow, $\uparrow$, represents the direction of movement of the wave-front: upwards (or outwards depending on your point of view) through the induction conclusion.

The presence of this wave-front prevents us from using the induction hypothesis to prove the induction conclusion. To enable the induction hypothesis to be used we need to move the wave-front to the outside of the induction conclusion, *i.e.* we need to rewrite the induction conclusion into the form:

$$P(tl) \vdash \boxed{Q(hd, tl, \underline{P(tl)})}^{\uparrow}$$

so that the induction hypothesis can be used, giving:

$$P(tl) \vdash Q(hd, tl, true)$$

We call this rewriting process *rippling*. It consists of applying rewrite rules which move the wave-fronts outwards but leave the rest of the induction conclusion unchanged. Rewrite rules of this form are called *wave-rules*. They have the form:

$$f(\boxed{g(\underline{X})}^{\uparrow}) \Rightarrow \boxed{h(\underline{f(X)})}^{\uparrow}$$

Some examples are given in figure 6. For more information about rippling see [Bundy *et al.* 91].

Example: Rippling in the *subset* proof To illustrate rippling consider the proof of equivalence (7) from §4. We start by putting a wave-front around each occurrence of the induction term and then we ripple these outwards.

$$subset(\boxed{[hd|\underline{tl}]}^{\uparrow}, j) \leftrightarrow$$
$$(\forall El \in Type.\ member(El, \boxed{[hd|\underline{tl}]}^{\uparrow}) \rightarrow member(El, j))$$

$$subset(\boxed{[hd|\underline{tl}]}^{\uparrow}, j) \leftrightarrow$$
$$(\forall El \in Type.\ \boxed{El = hd \vee \underline{member(El, tl)}}^{\uparrow} \rightarrow member(El, j))$$

$$subset(\boxed{[hd|\underline{tl}]}^{\uparrow}, j) \leftrightarrow$$
$$(\forall El \in Type.\ \boxed{El = hd \rightarrow member(El, j)\ \&\ \underline{member(El, tl) \rightarrow member(El, j)}}^{\uparrow})$$

$$\boxed{member(hd, j)\ \&\ \underline{subset(tl, j)}}^{\uparrow} \leftrightarrow$$
$$\boxed{\forall El \in Type.\ El = hd \rightarrow member(El, j)\ \&\ \underline{\forall El \in Type.\ member(El, tl) \rightarrow member(El, j)}}^{\uparrow}$$

$$\boxed{member(hd, j) \leftrightarrow \forall El \in Type.\ El = hd \rightarrow member(El, j)\ \&}$$
$$\boxed{\underline{subset(tl, j) \leftrightarrow \forall El \in Type.\ member(El, tl) \rightarrow member(El, j)}}^{\uparrow}$$

The wave-rules that enable this rewriting are given in figure 6.

Rippling is now complete and a copy of the induction hypothesis is embedded within the induction conclusion. The induction hypothesis can now be used to replace this copy with *true*. This leaves the subgoal:

$$member(hd, J) \leftrightarrow \forall El \in Type.\ El = hd \rightarrow member(El, J) \qquad \&\ true$$

which is readily proved. This completes the step case.

There are many other search control problems in inductive inference, *i.e.* choosing an induction rule, guiding the base case, deciding on case splits, constructing witnesses of existential variables, generalising the induction formula, conjecturing suitable lemmas. Rippling is often the key to solving these problems. For instance, an induction rule can be chosen by a look-ahead mechanism to see which choice will most facilitate subsequent rippling. Unfortunately, there is insufficient space to explore these issues further here. The interested reader is referred to [Bundy *et al.* 90].

7 Termination of Logic Programs

The problem we will consider in this section is how to prove that a logic program terminates. To solve this problem we must reason about programs rather than specifications. This is because whether a program terminates depends on both the code and the interpreter, that is, we must consider how the search strategy evaluates the code. There is a wider variation of interpreters in logic programming than in other kinds of programming. This factor makes termination a more complicated problem for logic programs than it is for other kinds of programs.

$$
\begin{aligned}
\mathit{subset}_1(\boxed{[Hd|\underline{Tl}]}^{\uparrow}, J) &\Rightarrow \boxed{\mathit{member}(Hd, J) \ \& \ \underline{\mathit{subset}_1(Tl, J)}}^{\uparrow} \\
\mathit{member}(El, \boxed{[Hd|\underline{Tl}]}^{\uparrow}) &\Rightarrow \boxed{El = Hd \vee \underline{\mathit{member}(El, Tl)}}^{\uparrow} \\
\boxed{A \vee \underline{B}}^{\uparrow} \rightarrow C &\Rightarrow \boxed{(A \rightarrow C) \ \& \ (\underline{B \rightarrow C})}^{\uparrow} \\
\forall X \in T.\ \boxed{A \ \& \ \underline{B}}^{\uparrow} &\Rightarrow \boxed{\forall X \in T.\ A \ \& \ \underline{\forall X \in T.\ B}}^{\uparrow} \\
\boxed{(A_1 \ \& \ \underline{B_1})}^{\uparrow} \leftrightarrow \boxed{(A_2 \ \& \ \underline{B_2})}^{\uparrow} &\Rightarrow \boxed{(A_1 \leftrightarrow A_2) \ \& \ (\underline{B_1 \leftrightarrow B_2})}^{\uparrow}
\end{aligned}
$$

If the left-hand side of a wave-rule matches a sub-expression of a goal then this sub-expression can be replaced by the instantiated right-hand side of the wave-rule. In this match the wave-fronts in the rule and the goal must be aligned. Note that this causes the wave-front to move outwards through the goal leaving the rest of the goal unchanged.
Wave-rules can be derived from the step parts of recursive definitions, e.g. *the subset and member rules above. They can also come from distributive laws,* e.g. *the other three rules above, from associative laws, substitution laws and many other sources. Our proofs run backwards from the theorem to the axioms. Thus, in wave-rules based on implications, the direction of rewriting is opposite to the direction of implication,* cf. *the last rule above.*

Fig. 2. Wave-Rules for the *subset* Example

Another complication is caused by the relational nature of logic programs, *i.e.* the fact that they work in different input modes and that they can return alternative outputs on backtracking. Termination is no longer a simple concept. A program may terminate in some input modes, but not in others. Hence, whether a program terminates depends on the goal. A program may return one or more outputs, but then fail to terminate on backtracking. We must define different kinds of non-termination.

Universal termination: the search space is finite.
Existential termination: if the search space is infinite, then it has success branches at finite depth.

Note that existential termination includes universal termination as a special case.

Whether a search space is finite depends, among other things, on the order in which subgoals are solved. For instance, consider the clause defining $p2$ below:

$$
\begin{array}{l}
p2(X) \leftarrow r(X) \ \&\ q(X). \\
q(a). \\
r(b) \leftarrow r(X). \\
r(a).
\end{array}
$$

If $r(X)$ is solved before $q(X)$ then the search space will be infinite, but if $q(X)$ is solved first then the search space is finite.

In the discussion below we will describe a simple termination technique that establishes universal termination of definite programs[5] Our technique is to associate a well-founded measure with each procedure call and to show that this strictly decreases as computation proceeds. A well-founded measure is one that cannot decrease for ever. This ensures that the computation cannot proceed for ever. An example of a well-founded measure is the natural numbers[6] ordered by $>$. This is well-founded because there is no infinite sequence of the form: $n_1 > n_2 > n_3 > \ldots$. Eventually, one of the n_i would be 0, and there is no natural number smaller than that. This natural number, well-founded measure will suffice for our purposes. To prove termination we will associate a natural number with each procedure call and show that the number associated with the current procedure call strictly decreases as the computation proceeds.

Example: The Termination of *subset* A simple example will illustrate this technique.

Consider the pure logic *subset* program in input mode $subset(+,+)$.

$$subset([\,], J). \tag{8}$$

$$subset([Hd|Tl], J) \leftarrow member(Hd, J) \ \&\ subset(Tl, J). \tag{9}$$

[5] *I.e.* those without negation as failure.
[6] *I.e.* the non-negative integers: 0,1,2,...

$$member(El, [El|Tl]). \tag{10}$$
$$member(El, [Hd|Tl]) \leftarrow member(El, Tl). \tag{11}$$

We will define a measure on procedure calls of the form $subset(I, J)$ and $member(El, J)$, where both I and J are ground.

Definition 3 (List length norm) *The* list length norm *is a function from a list, L, to a natural number, $len(L)$, defined recursively as:*

$$len([\,]) = 0 \quad len([Hd|Tl]) = len(Tl) + 1$$

Definition 4 (procedure call measure) *The* procedure call measure *is a function, $|\ldots|$, from a literal, $subset(I, J)$ or $member(El, J)$, where I and J are ground, to a natural number defined as:*

$$\begin{aligned} |subset(I, J)| &= len(I) + len(J) \\ |member(El, J)| &= len(J) \end{aligned}$$

We now show that the procedure call measure is strictly decreased as the computation proceeds. Each step of the computation is a resolution with one of the clauses (8) — (11). Each resolution replaces a call of the head literal with calls to each of the body literals. We show that the measure of the head literal is strictly greater than the measure of each of the body literals. In the case of the clauses (8) and (10) this is trivial, since their bodies contain no literals. In the case of clauses (9) and (11) the following calculations establish our claim.

$$\begin{array}{rcccl} |subset([Hd|Tl], J)| &=& len([Hd|Tl]) + len(J) &=& len(Tl) + 1 + len(J) \\ &>& len(J) &=& |member(Hd, J)| \\ |subset([Hd|Tl], J)| &=& len([Hd|Tl]) + len(J) &=& len(Tl) + 1 + len(J) \\ &>& len(Tl) + len(J) &=& |subset(Tl, J)| \\ |member(El, [Hd|Tl])| &=& len([Hd|Tl]) &=& len(Tl) + 1 \\ &>& len(Tl) &=& |member(El, Tl)| \end{array}$$

Note that this argument breaks down if any of the lists in the procedure call arguments are non-ground. Indeed, a procedure call $member(a, L)$, for instance, will not terminate in the universal sense. It will return an infinite number of results of the form:

$$[a|Tl], \quad [Hd1, a|Tl], \quad [Hd1, Hd2, a|Tl], \quad \ldots$$

Similar remarks hold for $subset(I, J)$ where either I or J is non-ground. These are examples of existential termination.

An introduction to more elaborate techniques for proving termination can be found in [Hogger 90][Theme 59] and [De Schreye & Verschaetse 92].

8 Tuning Impure Programs

Suppose we have specified a logic program, transformed this into an equivalent specification and then compiled this new specification into a Prolog program using the Lloyd-Topor algorithm. We have seen that the completion of this Prolog program is logically equivalent to the original specification. Unfortunately, due to the impure aspects of Prolog, this program may not behave operationally as desired. For instance,

- It may not terminate in its intended input mode.
- Some of its uses of negation as failure may flounder.
- Its side effects may occur in the wrong order.

For a complete list of the possible problems see [Deville 90][p240].

Sometimes these problems can be fixed by tuning the program.

Examples: Reordering clauses and literals We can illustrate this with some examples.

Consider the following clauses for `r/1`:

```
r(b) :- r(X).
r(a).
```

These clauses do not existentially terminate in input mode `r(-)`. This problem can be readily fixed by reordering the two clauses to:

```
r(a).
r(b) :- r(X).
```

This gives a logically equivalent,but operationally different program. The program for `r/1` now existentially terminates.

Now consider the following clauses:

```
p2(X) :- r(X), q(X).
q(a).
r(a).
r(b) :- r(X).
```

The program for `p2/1` existentially terminates in input mode `p2(-)`. We can transform this into a program that universally terminates by reordering the literals in the `p2` clause to give:

```
p2(X) :- q(X), r(X).
```

As we have seen, in §2.2, this clause can be partially evaluated to the logically and operationally equivalent clause:

```
p2(a).
```

9 Abstract Interpretation of Logic Programs

The techniques we have discussed above can be used not only to reason with programs in their original form, but also with abstractions of these programs. For instance, we can infer mode and type information about programs, or we can discover whether variables are independent or aliasing during execution. This information can be used, for instance, to optimise programs. Abstract reasoning can be automated and incorporated into optimising compilers.

To apply abstract reasoning to a logic program we must define the abstraction as follows.

Abstract the domain: We must describe the objects that can appear as arguments to the program. For instance, for reasoning with modes we will want objects to represent a free variable and an instantiated variable. We might also want objects for various degrees of instantiation, *e.g.* totally ground *vs* partially instantiated. For reasoning about types we will choose abstract objects to represent the various types of concrete objects, *e.g.* numbers, strings, lists, *etc.*
In addition, to these basic abstract objects we must usually specify how to form the least upper bound of any two objects. This is because finding the least upper bound is often used to combine different instantiations of the same variable. This is usually done by putting the objects in a lattice with a top element, *e.g.* any mode, any type, *etc.*

Abstract the programs: We must describe how the program operates on this abstract domain. For instance, we must define the built-in predicates over the abstract domain. We must adapt the interpreter, as necessary. For instance, if a variable is instantiated in different ways on different branches of the search space these values might be combined into one by taking the least upper bound. We must describe how to unify abstract objects.
Non-termination is far more common with abstract programs, because distinct concrete procedure calls are often identical when abstracted. Therefore, it is usually necessary to modify the interpreter to trap looping and ensure that an output is returned.

Example: Inferring the Types of Predicates We will illustrate these ideas by the use of a simple example: the inference of the types of a whole program, given some partial information about types.

Our abstract domain will consist of the simple lattice of types given in figure 3.

Suppose we want to infer the types of `subset` and `member` as defined by the clauses:

```
subset([],J).
subset([Hd|Tl],J) :- member(Hd,J), subset(Tl,J).

member(El,[El|Tl]).
member(El,[Hd|Tl]) :- member(El,Tl).
```

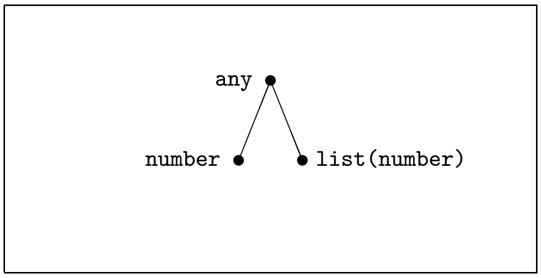

Fig. 3. Simple Lattice of Types

given that the type of the first argument of `subset` is known to be *list*(*number*), say.

To infer the remaining types we partially evaluate the procedure call:

```
?- subset(list(number),X).
```

If we resolve this goal against the first clause for `subset` then we must unify `list(number)` with `[]` and `X J`. We should define abstract unification so that these unifications both succeed, but this resolution does not tell us more than we already knew.

More interesting is the resolution against the second clause for `subset`. We should define abstract unification so that unifying `list(number)` and `[Hd|Tl]` succeeds, instantiating `Hd` to `number` and `Tl` to `list(number)`. The new procedure calls are:

```
?- member(number,X), subset(list(number),X).
```

We can now repeat this process for the `member(number,X)` procedure call. This gives us two instantiations of `X`, both of which are `list(number)`. The new procedure call generated by the second clause is:

```
?- member(number,list(number)).
```

but this is subsumed so by the earlier call, so can tell us nothing new. Our loop detection mechanism should note this and terminate this call.

The remaining procedure call is:

```
?- subset(list(number),list(number)).
```

which is also subsumed by an earlier call and should also be terminated.

The abstract interpretation is now complete. We have inferred the types of our predicates to be:

```
subset(list(number),list(number))
member(number,list(number))
```

10 Conclusion

In these notes we have outlined various techniques for reasoning about logic programs. These techniques can be combined in various ways to form a methodology for logic program development. Ideally, for the reasons summarised in §2.3, this methodology should take logic program specifications as its central representation. The original description of the desired program should take the form of a specification, as defined in §1.2.

This specification can be used to synthesise a more efficient specification using the techniques of §5. It may seem odd to discuss the 'efficiency' of a specification. One way to measure this is to compile the specification into a logic program using the Lloyd-Topor algorithm (see §3.2) and then to measure the efficiency of this program. However, there are some aspects of efficiency that are independent of the particular target programming language, *e.g.* the complexity associated with any forms of recursion used in the specification. These aspects can be measured more directly.

Having synthesised an acceptable specification, this can be compiled into a program using the Lloyd-Topor algorithm. The termination, mode and similar properties of this program can be analysed using the techniques of §7 and 9. If necessary, the program can then be tuned using the techniques of §8.

If a program, rather than a specification, is available, then this can be lifted into a specification using the Clark completion algorithm, §3.1. The above methodology is then applicable.

All these techniques can be automated to a greater or lesser extent. Automation makes possible machine aids to program development that remove some of the tedium and error from the proof, compilation and analysis steps.

Recommended Reading

In these notes it has only been possible to give a rough outline of the range and complexity of the techniques available. If you want to find out more, here are some suggestions for further reading.

Elementary and very readable introductions to the ideas outlined in these notes can be found in the two books by Chris Hogger, [Hogger 84, Hogger 90]. John Lloyd's book, [Lloyd 87], is the standard reference for the theoretical background on logic programming. A more detailed account of verification and synthesis of logic programs can be found in Yves Deville's book [Deville 90]. Deville adopts the same position that we have on reasoning with specifications, wherever possible, rather than with programs. A discussion of the different notions of logic program equivalence can be found in the paper by Michael Maher, [Maher 87]. An introductory survey of termination proving techniques can be found in the tutorial notes by Danny De Schreye and Kristof Verschaetse, [De Schreye & Verschaetse 92], and a similar survey of work on abstraction can be found in the tutorial notes by Maurice Bruynooghe and Danny De Schreye, [Bruynooghe & De Schreye 88].

References

[Bruynooghe & De Schreye 88] M. Bruynooghe and D. De Schreye. Tutorial notes for: abstract interpretation in logic programming. Technical report, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, 1988. Tutorial given at ICLP-88, Seattle. Notes available from the authors. Department of Computer Science, Katholieke Universiteit Leuven, Celestijnenlaaan 200A, 3001 Heverlee, Belgium.

[Bundy *et al.* 90] A. Bundy, A. Smaill, and J. Hesketh. Turning eureka steps into calculations in automatic program synthesis. In S. L.H. Clarke, editor, *Proceedings of UK IT 90*, pages 221–6. IEE, 1990. Also available from Edinburgh as DAI Research Paper 448.

[Bundy *et al.* 91] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. Research Paper 567, Dept. of Artificial Intelligence, University of Edinburgh, 1991. In the Journal of Artificial Intelligence.

[De Schreye & Verschaetse 92] D. De Schreye and K. Verschaetse. Termination of logic programs: Tutorial notes. Technical Report CW-report 148, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, 1992. Also in the proceedings of Meta-92.

[Deville 90] Y. Deville. *Logic programming: systematic program development.* Addison-Wesley Pub. Co., 1990.

[Hogger 84] C. J. Hogger. *Introduction to logic programming.* Academic Press, 1984.

[Hogger 90] C. J. Hogger. *Essentials of Logic Programming.* Oxford University Press, 1990.

[Lloyd 87] J. W. Lloyd. *Foundations of Logic Programming.* Symbolic Computation. Springer-Verlag, 1987. Second, extended edition.

[Maher 87] M. J. Maher. Equivalences of logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann, 1987.