From Programs to Object Code and back again using Logic Programming: Compilation and Decompilation

Jonathan Bowen

Oxford University Computing Laboratory
Programming Research Group
11 Keble Road
Oxford OX1 3QD, UK

Telephone: +44-865-272574
Fax: +44-865-273839
E-mail: Jonathan.Bowen@comlab.ox.ac.uk

February, 1993

From Programs to Object Code and back again using Logic Programming: Compilation and Decompilation

Summary

A compiler may be specified by a description of how each construct of the source language is translated into a sequence of object code instructions. It is possible to produce a compiler prototype almost directly from this specification in the form of a logic program. This defines a relation between allowed high-level and low-level program constructs. Normally a high-level program is supplied as input to a compiler and object code is returned. Because of the declarative nature of a logic program, it is possible for the object code to be supplied and the allowed high-level programs returned, resulting in a decompiler, provided enough information is available in the object code. This paper discusses the problems of adopting such an approach in practice. A simple compiler and decompiler are presented in full as an example in the logic programming language Prolog, together with some sample output. The possible benefits of using *constraint logic programming* are also considered. Potential applications include reverse engineering in the software maintenance process, verification of safety-critical object code, quality assessment of code and program debugging tools.

Keywords: Reverse Engineering Decompilation Logic Programming Prolog Compiler Prototyping Constraint Logic Programming

1 Background

A compiler may be specified as a relation

$$\mathcal{C}\; p\; o\; \Psi$$

where p represents a high-level source program, o are the low-level object code sequences that are allowed for the corresponding source program, and Ψ is a symbol table that relates the high-level variables to their positions in memory. It is possible to define an ordering on programs such that a program is considered 'better' than another if it is more deterministic and terminates more often. If the semantics of the object code is defined in terms of an interpreter in the high-level language then it is possible to conduct proofs in an algebraic style using laws about the language to transform the source program constructs into a corresponding interpreter and object code that is equivalent or better than the original code. Compilation specifications for individual programming constructs may be specified as theorems and proved using a set of algebraic laws (which may themselves be derived from a specification-oriented semantics).

The above approach is the one adopted on the ESPRIT Basic Research **ProCoS** project (Bjørner, 1992, Bjørner *et al.*, 1992), and is detailed more fully in (Hoare *et al.*, 1990b). The compilation theorems are in general in the form of Horn clauses allowing an almost direct implementation as a logic program in a language such as Prolog (Bowen, 1992). Since the clauses in a logic program in principle define a relation between the parameters, it is possible to derive both a compiler *and* a decompiler, depending on whether a source program or object code are supplied as input, with a little ingenuity to ensure termination and by restricting the programs returned to ones of interest (since the number can and is in general infinite) (Breuer *et al.*, 1992a, Breuer *et al.*, 1992c, Bowen *et al.*, 1993a).

Whilst the concept of a compiler is widespread, the idea of a decompiler is more novel. This allows defined object code sequences to be mapped back to high-level constructs. This could have applications in the following areas:

- It may be useful in the software maintenance process to reverse engineer existing object code back to a high-level representation to gain understanding of the code, particularly if the original source code has been lost.
- It could be beneficial in the validation or verification of safety-critical code. Here the compiler may not be considered trustworthy enough and it may be desirable to validate the low-level object code rather than the original source code since this is the code that actually defines the operation of the processor.
- It could be used in the quality assessment of code, for example, to discover how well structured it is, or a variety of other desired metrics.

- Currently debuggers routinely disassemble machine code back to a higher-level assembly language representation to aid the debugging process. It could be helpful to obtain an even higher-level representation of the code in the form of a structured high-level program. This could automatically search for structure in the code and display the most appropriate program to the software engineer to help his/her understanding of the operation of the code. This would in general require knowledge of the original compiler and also the symbol table of the program.

1.1 Existing work

Literature on decompilation is extremely hard to find. However, decompilation may be useful when reverse engineering is required for part of the software maintenance process to gain understanding of the object code. For example, the REFORM project has produced a maintainer's assistant which is based upon transformation of assembly language code into a wide-spectrum language (Ward *et al.*, 1989).

Decompilation has always been of interest when compilers have been used in safety-critical systems (Bowen *et al.*, 1993b), to add an extra level of confidence that the low-level object code produced by a compiler does indeed correspond correctly to the high-level program (Clutterbuck *et al.*, 1988). For example, IBM produced a decompiler for the NASA Space Shuttle software (Spector *et al.*, 1984). They worked on a tool to decompile memory images and compare the results with the original inputs. More recently, Nuclear Electric in the UK have used decompilation techniques to verify significant amounts of safety-critical code (Pavey *et al.*, 1992). They first disassemble Intel PL/M-86 compiler object code. This and the source code are converted into a common language (MALPAS IL) and the two can be compared for consistency using static analysis techniques. There are limitations to this approach, but it appears to be a practical method to increase the confidence in the correctness of the code produced by an unverified (commercially available) compiler.

1.2 Legal concerns

Decompilation may be considered illegal, especially when applied to a third party's product (Samuelson, 1990). For example, the desktop publishing package FrameMaker$^{\text{TM}}$ displays the following message in its on-screen licensing information:

> You may make and distribute identical copies of the FrameMaker
> distribution tape provided that you do not attempt to
> (1) decompile or decipher the software,
> (2) develop passwords for the software, or
> (3) otherwise enable the Save feature yourself,

and provided that you do not permit others to do so.

The introduction of law in this area is fraught with problems. For example, a new UK law, *The Copyright (Computer Programs) Regulations 1992* (UK Government, 1992), on software copyright came into force on 1st January 1993 to meet a European directive. It modifies the *Copyright, Designs and Patents Act 1988* which allowed for the "decompilation" of programs for research or private study. The new law could affect that right in future. Note that in the UK all programs are automatically copyrighted so changes in the law could have far-reaching and even unexpected implications.

2 Introduction

This paper demonstrates a possible method of reverse engineering of object code to a high-level representation of the program. This could be useful if, for example, the original source code has been lost. This is an extension of other ideas of reverse engineering conceived on the ESPRIT II REDO project (van Zuylen *et al.*, 1993), a collaborative project of 11 industrial and academic partners concerned with software maintenance (Breuer *et al.*, 1991, van Zuylen *et al.*, 1993). In particular, work on converting (high-level) code to Z specifications has been undertaken by others on the project (Lano *et al.*, 1990). Combining the two approaches could allow the specification of an object program to be generated, with some guidance from a software engineer.

The first part of the paper briefly introduces the mathematical foundations of logic programming, and Prolog in particular. Then a compiling specification for a simple imperative programming language is presented in the form of a set of theorems which may be verified formally if desired. A compiler and decompiler prototype in Prolog, based on this specification, are included virtually in their entirety, together with some sample output. A discussion on the possible benefits of using *constraint logic programming* (CLP) and a simple CLP compiler/decompiler are presented. Finally some conclusions are drawn on the possible applicability of the method in practice.

3 Logic programming and Prolog

As the name suggests, logic programming has a well established mathematical basis (Lloyd, 1987, Hogger, 1990, van Emden *et al.*, 1976). Prolog (Clocksin *et al.*, 1987) is the most widely available logic programming language. However, Prolog includes many non-logical features in an attempt to make it into a practical programming language. Even so, if the features used in Prolog are restricted, it is possible to use it in a logical manner. For example, the Prolog Horn clause

$$P \texttt{ :- } Q_1 \texttt{,} \ldots \texttt{,} Q_n \texttt{.}$$

is equivalent to the following formula in first order predicate logic:

$$\forall\, x, y, \ldots \cdot ((Q_1 \wedge \ldots \wedge Q_n) \Rightarrow P)$$

where $x, y, \ldots$ are all the free variables in the predicates P and Q_i $(1 \leq i \leq n)$. Note that the '`:-`' of Prolog can be considered as a reverse implication ($\Leftarrow$) in predicate logic. This theorem is in turn equivalent to:

$$\forall\, x_1, \ldots, x_l \cdot ((\exists\, y_1, \ldots, y_m \cdot Q_1 \wedge \ldots \wedge Q_n) \Rightarrow P)$$

where $x_1, \ldots, x_l$ are the free variables in P (normally mentioned in the Q_i predicates as well) and $y_1, \ldots, y_m$ are the variables mentioned in Q_i but not in P. The quantifiers may be omitted when there are no relevant free variables. Additionally, if there are no Q_i clauses, this part of the formula reduces to *true* and the implication may be omitted since $\mathit{true} \Rightarrow P = P$. Such formulae, in which all the variables are set to some specific value are known as *facts* (Clocksin *et al.*, 1987).

A set of such clauses form a program. Queries (or *goals*) may be posed to this program as the conjunction of a set of goal clauses

$$\texttt{?- } G_1 \texttt{,} \ldots \texttt{,} G_n \texttt{.}$$

This is equivalent to the following in predicate logic

$$\neg\ \exists\, y_1, \ldots, y_m \cdot (G_1 \wedge \ldots \wedge G_n)$$

The Prolog system searches for a contradiction to this clause. If it finds one (or more), these are output successively as they are discovered. A very simple left-to-right and depth first search of the database of clauses is used. This can result in non-termination in practice if a search is made down an infinite branch of the proof tree. Thus some judicious ordering of the clauses is often necessary to ensure that an answer is found in finite time.

Using the form of clauses described above results in a restricted form of predicate logic. Note in particular that none of the Q_i and G_i predicates may be negated (e.g., $\neg Q_i$). In practice this can be too much of a limitation sometimes, and the restricted form of negation, *negation by failure*, is normally allowed Prolog (Lloyd, 1987, Quintus, 1990). This type of negation normally limits the modes in which the logic program can be used (i.e., which variables must be instantiated, and which can be left uninstantiated). In practice this may not be too restrictive since the logic program can be designed knowing which variables are to be inputs and outputs. Where negation is necessary in the logic program presented in this paper, its use is justified informally.

3.1 An example

Prolog consists of Horn clauses, which are a special case of predicates specifying relations between their parameters. Because of this declarative nature of Prolog it is possible in theory to run clauses 'backwards' as well as 'forwards'. I.e., the inputs (supplied 'instantiated' parameters) and outputs (unknown 'uninstantiated' parameters) to a clause may be reversed. This works well for some simple examples. Consider the well-known `append` program in Prolog:

```
append([],T,T).
append([X|S],T,[X|U]) :- append(S,T,U).
```

The second parameter may be appended to the first parameter to form the third parameter:

```
| ?- append([a,b],[c],L).

L = [a,b,c] ;

no
```

The first or second parameter can equally well be uninstantiated instead:

```
| ?- append(L,[c],[a,b,c]).

L = [a,b] ;

no
| ?- append([a,b],L,[a,b,c]).

L = [c] ;

no
```

In fact both the first two parameters can be unknown and all possible combinations of lists which when appended together form the third list are returned:

```
| ?- append(L1,L2,[a,b,c]).

L1 = [],
L2 = [a,b,c] ;

L1 = [a],
L2 = [b,c] ;

L1 = [a,b],
L2 = [c] ;

L1 = [a,b,c],
L2 = [] ;

no
```

However more complicated programs can exhibit non-termination when run in certain modes due to the simplistic left-to-right and depth-first search strategy of Prolog. This can be alleviated by reordering clauses appropriately, or adding the infamous *cut* pseudo-predicate (‘!’) at suitable points in the program. Unfortunately the latter can destroy the logical meaning of the program if used inappropriately by removing logically valid answers. This paper avoids the use of cuts so they will be mentioned no further, although in a more realistic example their use would probably become a necessity for efficiency reasons.

3.2 Compiler writing in Prolog

The idea of using Prolog (Clocksin *et al.*, 1987) for the construction of compilers has been accepted for some time (Warren, 1980). Advantages include the fact that the code for the compiler can be very close to the compiling specification since Prolog is based on logic (Lloyd, 1987) and thus the confidence in its correctness is increased. It can be used as a prototype compiler and even as a ‘real’ compiler; the Prolog code itself may be compiled into optimized code for increased efficiency (Quintus, 1990). Prolog has been found to be particularly good for code generation and almost as efficient as using a high-level imperative programming language such as Pascal (Paakki, 1991).

In theory, a compiler written in Prolog could also be run backwards. In practice, much care needs to be taken if this approach is adopted. Even ‘pure’ Prolog is neither sound (there is no *occurs check* in the unification algorithm used) nor complete (because of its search strategy) despite its background in logic programming (Lloyd, 1987). These compromises have been made for efficiency reasons. However with judicious and sensible recoding and reordering of clauses it is possible to produce a decompiler from a Prolog compiler. Normally this will only work with the output of a particular compiler, although it would be possible to search for common control structures in arbitrary object code and return some sort of higher-level representation of these.

4 Compiling specification

A simple compiling specification is presented here for completeness. For further information and a proof of correctness of each of the theorems below, see (Hoare *et al.*, 1990a) from which this specification is extracted. Later sections in this paper present a compiler and decompiler based on this specification.

The object code is given by $m[s:f)$, where m is the code store, s is the starting address and f is the finishing address (immediately *after* the object code) for execution. Ψ is a symbol table consisting of a finite injection (a one-to-one function) from variable names to locations.

The object code instruction set consists of $\mathit{load}(n)$, $\mathit{store}(n)$ for loading and storing an accumulator from and to memory, and $\mathit{jump}(j)$ and $\mathit{cond}(j)$ for unconditional and conditional jumps. These are defined formally in (Hoare *et al.*, 1990a) and presented more informally as a high-level program in section 5.

Compiling specification theorems are presented as special cases of a relation $\mathcal{C}$ between a high-level program construct p, the matching object code $m[s:f)$, and the symbol table Ψ.

$$\mathcal{C}p\; m[s:f)\Psi$$

The program `SKIP` may be compiled to nothing:

Theorem 1 $\{$`SKIP`$\}$
$\mathcal{C}\,\mathtt{SKIP}\; m[s:s)\Psi$

Object code allowed for a particular high-level construct need not be unique. For example, `SKIP` may also be implemented by a forward jump:

Theorem 1a
If $s < f$ and $m[s] = \mathit{jump}(f)$ then $\mathcal{C}\,\mathtt{SKIP}\; m[s:f)\Psi$.

Sequential composition depends on compiling two subprograms contiguously in memory:

Theorem 2 {composition}
If $s \le j \le f$ and $\mathcal{C}p\; m[s:j)\Psi$ and $\mathcal{C}q\; m[j:f)\Psi$ then $\mathcal{C}(p;\, q)m[s:f)\Psi$

Conditional and looping constructs may be compiled in the traditional manner:

Theorem 3 {conditional}
If $s{+}2 \le j \le f$ and $m[s] = \mathit{load}(\Psi(b))$ and $m[s{+}1] = \mathit{cond}(j)$ and $m[j{-}1] = \mathit{jump}(f)$

and $\mathcal{C}p_1 m[s+2:j-1)\Psi$ and $\mathcal{C}p_2 m[j:f)\Psi$
then $\mathcal{C}(\,\texttt{IF}\,b\,\texttt{THEN}\,p_1\,\texttt{ELSE}\,p_2)m[s:f)\Psi$

Theorem 4 {while}
If $s+1 \leq f-1$ and $m[s] = \mathit{load}(\Psi(b))$ and $m[s+1] = \mathit{cond}(f)$
and $m[f-1] = \mathit{jump}(s)$ and $\mathcal{C}p\, m[s+2:f-1)\Psi$
then $\mathcal{C}(\,\texttt{WHILE}\,b\,\texttt{DO}\,p)m[s:f)\Psi$

Non-determinism allows the implementor to chose between more than one program:

Theorem 5 {non-determinism}
If $\mathcal{C}p\, m[s:f)\Psi$ then $\mathcal{C}(p \sqcap q)m[s:f)\Psi$

Theorem 5a
If $\mathcal{C}q\, m[s:f)\Psi$ then $\mathcal{C}(p \sqcap q)m[s:f)\Psi$

If the program aborts, any object code can be generated. Thus the memory is not constrained in any way and the symbol table is also immaterial:

Theorem 6 {$\texttt{ABORT}$}
$\mathcal{C}\,\texttt{ABORT}\, m[s:f)\Psi$

Each declared variable is distinct from all other global variables:

Theorem 7 {declaration}
If $\Phi = \{v\} \lhd \Psi$ and $\mathcal{C}p\, m[s:f)\Psi$ then $\mathcal{C}(\,\texttt{VAR}\ v\,;\ p\,;\ \texttt{END}\ v)m[s:f)\Phi$

Φ is the same as Ψ except that the entry for the variable v is removed.

A simple example of assignment is included here. In practice, expressions would normally be allowed on the right hand side.

Theorem 8 {assignment}
If $m[s] = \mathit{load}(\Psi(y))$ and $m[s+1] = \mathit{store}(\Psi(x))$ then $\mathcal{C}(x := y)m[s:s+2)\Psi$

Recursion is considerably more complex than the previously presented theorems, but is included here as an example of a more complicated construct. Extra instructions $\mathit{push}(f)$, $\mathit{subr}(s+1)$ (which is equivalent to $\mathit{push}(j+1);\mathit{jump}(s+1)$) and *return* are required to implement recursion using subroutine calls and returns, with the return addresses stored on a stack. These are formally defined in the appendix of (Hoare *et al.*, 1990a).

Theorem 9 {recursion}
If $m[s] = \mathit{push}(f)$ and $m[f-1] = \mathit{return}$ and
$\forall\, X \bullet (\forall\, j \in [s+1 : f-1) \bullet (m[j] = \mathit{subr}(s+1) \Rightarrow \mathcal{C} X\, m[j : j+1)\Psi)$
$\quad\quad \Rightarrow \mathcal{C} F(X) m[s+1 : f-1)\Psi\,)$
then $\mathcal{C}\, \mu\, X.F(X) m[s : f)\Psi$

This results in code of the following form:

$s-1$	s	s+1	j	j+1	$f-1$	f
...	$\mathit{push}(f)$	...	$\mathit{subr}(s{+}1)$	...	*return*	...

Note that the code is produced in-line and recursive calls re-enter the code at location $s+1$.

5 Compiler prototype

Most of the rest of this paper presents a logic program in the programming language Prolog (Clocksin *et al.*, 1987) which implements the previously presented compiling specification as a compiler and also a decompiler prototype. (It is assumed that the reader has a basic knowledge of logic programming, and Prolog in particular.) By restructuring the forward compiler slightly, it is possible to produce a decompiler. Some of the practical problems of this approach will be discussed.

Prolog allows operators to be defined with a specified precedence, position (prefix, postfix or infix) and associativity. This obviates the need for a parser for the high-level programming language, which may be coded in the following form, for example, for direct use by Prolog:

```
p := s;
while (s<=p) /\ (p<f) do
  if      m(p) = load(n)  then [a,p]:=[m(n),p+1]
  else if m(p) = store(n) then [m(n),p]:=[a,p+1]
  else if m(p) = jump(j)  then p:=j
  else if m(p) = cond(j)  then (if a then p:=p+1 else p:=j)
  else abort;
{p = f}
```

The above example shows an interpreter for the basic instruction set of the example object code used in this paper. ('p' is the program counter and 'a' is an accumulator.) The compiler presented in this paper does not accept the full syntax shown above, although it could easily be extended to do so.

5.1 Compiling clauses

Each theorem is coded as a separate clause. For this simple language, the form of each theorem in section 4 may be followed almost exactly. However Prolog provides *functors* (essentially trees) for data structures, with a special syntax for the convenience of constructing lists, so some data refinement is necessary to implement the compiler. The source program is defined using a number of infix and postfix operators to aid readability. The object code is modelled as a list of tuples, each of the form `n->`*instr* where n is a memory location (a natural number) and *instr* is the opcode at that location (e.g., `load(`x`)`, `store(`x`)`, etc.). The starting and finishing object code memory addresses are natural numbers. The symbol table is a finite injective function from variable names to locations. The actual locations of each variable are of no particular interest in this example; thus the function is simply modelled as a list of variables representing an ordered set. The location associated with a variable x is a unique location `psi(`x`)`, etc.

The theorems concerning `SKIP` are non-deterministic and thus several (in fact, an infinite number of) different object codes may be generated. In practice, these are each enumerated in turn, depending on the ordering of the clauses in the Prolog database.

`SKIP` may be compiled to the empty sequence of instructions. The symbol table is immaterial. In Prolog, the name '`_`' may be used for uninstantiated parameters which are only mentioned once in the clause.

Theorem 1

```
c(skip,[]/S:S,_).
```

Alternatively, a forward jump may also be used to implement `SKIP`.

Theorem 1a

```
c(skip,[S->jump(F)]/S:F,_) :-
  ensure(S<F).
```

Sequential composition and the conditional statement both require an implicitly existentially quantified intermediate location, `J`, at some position between the start and finish address. The object code from each program (`M1` and `M2`) is concatenated to form the entire program code for the compilation.

Theorem 2

```
c(P;Q,M/S:F,Psi) :-
  c(P,M1/S:J,Psi),
  c(Q,M2/J:F,Psi),
  ensure(S<=J), ensure(J<=F),
  concat([M1,M2],M).
```

The conditional and looping constructs require the symbol table to be accessed so that the location of the tested variable in memory is known.

Theorem 3

```
c(if B then P else Q,M/S:F,Psi) :-
  succ(S,S1), succ(S1,S2),
  psi(B->PsiB,Psi),
  c(P,M1/S2:J_1,Psi),
  succ(J_1,J),
  c(Q,M2/J:F,Psi),
  ensure(S2<J), ensure(J<=F),
  concat([[S->load(PsiB),S1->cond(J)],M1,[J_1->jump(F)],M2],M).
```

Theorem 4

```
c(while B do P,M/S:F,Psi) :-
  succ(S,S1), succ(S1,S2),
  psi(B->PsiB,Psi),
  c(P,M1/S2:F_1,Psi),
  succ(F_1,F),
  ensure(S2<=F_1),
  concat([[S->load(PsiB),S1->cond(F)],M1,[F_1->jump(S)]],M).
```

Non-deterministic compilation is achieved in Prolog by including more than one clause, any of which may be applied.

Theorem 5

```
c(P\/_,M/S:F,Psi) :-
  c(P,M/S:F,Psi).
```

Theorem 5a

```
c(_\/Q,M/S:F,Psi) :-
  c(Q,M/S:F,Psi).
```

Compiling `abort` does not constrain any of the other parameters.

Theorem 6

```
c(abort,_,_).
```

A fresh (unused) variable must be generated when a variable is declared. This will be discussed further in section 6.

Theorem 7

```
c(var V;P;end V,M/S:F,Phi) :-
  fresh(V,Phi,Psi),
  c(P,M/S:F,Psi).
```

In practice, assignment would allow an expression on the right hand side, but here we simply assign one variable to another for simplicity in this example.

Theorem 8

```
c(X:=Y,[S->load(PsiY),S1->store(PsiX)]/S:S2,Psi) :-
  succ(S,S1), succ(S1,S2),
  psi(Y->PsiY,Psi),
  psi(X->PsiX,Psi).
```

A number of extra predicates are needed in the clauses above (e.g., `concat`, `ensure`, `succ`, etc.) and these are detailed in section 6.

Procedurally, Prolog works from left to right and performs a depth-first search for solutions. Thus in practice it is convenient and more efficient to order clauses in a way which reduces backtracking in normal usage – i.e., when the source program, start address and symbol table are assumed to be inputs and the object code memory and finish address are outputs. Given a valid source program, matching object code and symbol table (i.e., using the program as a compiler *checker*), the program can always perform a validation. However, given an incomplete set of parameters, it is very easy to induce non-termination of the program because Prolog is attempting to search down at infinite branch of the proof tree. In particular it would be interesting to supply some object code and return (possibly several) programs which implement it. Unfortunately, in practice the program presented above will not normally terminate in this case. However with some judicious reordering of clauses and by applying the theorems in a sensible order it is possible to produce a decompiler in Prolog. This will be explored further in section 7.

Compilation of recursive programs

The theorem for a recursive program (Theorem 9) is slightly more complex than those previously presented. Here the assumption is made that a subroutine call to the recursive program can be made anywhere within that program. This can be achieved in Prolog by adding an extra compiling clause allowing the subroutine call to be made within the recursive program to the Prolog database. Ideally the clause could be asserted and retracted at the start and end of the compilation of the recursive program.

At first sight, an appealing way to implement such implication in Prolog is as follows:

```
P=>Q :- assert(P), call(Q), retract(P).
```

Then the universally quantified implication in Theorem 9 can be coded as:

```
(c(X,[J->subr(S1)]/J:J1,_) :-
   succ(J,J1), ensure(S1<=J), ensure(J1<=F_1))) =>
     c(P,M1/S1:F_1,Psi)
```

First the antecedent is 'asserted' (added to the database of clauses), the consequent is called and finally the antecedent is 'retracted' (removed from the database). This works fine in practice, unless backtracking occurs (i.e., in the case of a nondeterministic compilation in this application). Unfortunately `assert` and `retract` are not reversible; that is to say, backtracking through a `retract` clause does not reassert the clause and vice versa for an `assert` clause. It is possible to explicitly code such behaviour in Prolog as follows:

```
add(P) :- assert(P).
add(P) :- retract(P), fail.

remove(P) :- retract(P).
remove(P) :- assert(P), fail.

P=>Q :- add(P), call(Q), remove(P).
```

(`fail` is a built-in clause which can never succeed.) This allows multiple answers to be returned for recursive programs. The complete theorem for recursion may be encoded as follows:

Theorem 9

```
c(mu X:P,M/S:F,Psi) :-
  succ(S,S1),
  ident(X->S1),
  ((c(X,[J->subr(S1)]/J:J1,_) :- succ(J,J1),ensure(S1<=J)))
```

```
      => c(P,M1/S1:F_1,Psi),
    succ(F_1,F),
    concat([[S->push(F)],M1,[F_1->return]],M).
```

Note that the constraint that `J1<=F_1` cannot be usefully included in the asserted clause because the size of the compiled recursive code is not known at this point under the procedural reading of Prolog. Since the clause is retracted after use, this does not pose a problem. The scheme means that the name `X` must not clash with existing programs and thus that mutually recursive programs should have unique names, but this is not a problem in practice. In fact if two recursive programs with the same name are used then both possibilities are returned.

The clause `ident` ensures that the name of the recursive program is a string or uniquely names the program using its start address. The former avoids conflicts with exiting program constructs; the latter is particularly useful for decompilation (see section 7).

```
ident(X->_) :- atom(X).
ident($N->N).
```

Other logic programming systems based on intuitionistic logic (McCarty, 1988) allow the use of an implication on the right-hand side of a clause (e.g., (Miller, 1989)). Such systems would allow the direct coding of this theorem.

6 Supporting clauses

The symbol table Ψ is stored as a list which represents the set of variables since Ψ is a finite injection and the actual variable locations are of no particular interest in this example. The `psi` clause allows variables in the table to be accessed:

```
psi(V->psi(V),Psi) :-
  variable(V), table(Psi), member(V,Psi).
```

An encoding for $\Phi = \{v\} \lhd \Psi$ in theorem 7 must be selected. Since $v \in \operatorname{dom}\Psi$ and $v \notin \operatorname{dom}\Phi$ this is equivalent to $\Psi = \Phi \uplus \{v \mapsto \Psi(v)\}$. Negation in Prolog (`\+`), needed to implement $\notin$, is not sound in general (Lloyd, 1987); however if all the parameters to the clause are known at the time, it can be applied safely using *negation by failure*. The symbol table and the number of allowed variables is finite in this case so it is possible to ensure this; `variable(V), table(Phi)` below make sure that `V` and `Phi` are always fully instantiated when `\+ member(V,Phi)` is called thus avoiding any unsoundness problems.

```
fresh(V,Phi,Psi) :-
```

```
  variable(V),
  table(Phi), \+ member(V,Phi),
  setof(X,member(X,[V|Phi]),Psi),
  table(Psi), member(V,Psi).
```

(setof(x,y,z) implements set comprehension $\{x \mid y \bullet z\}$ for finite sets.)

For example purposes, a small finite number of variable names are allowed so that all the possible combinations of variable declaration may be displayed, particularly during decompilation.

```
variable(a).
variable(b).
variable(c).
    ⋮
```

In practical use, the `variable` clause would simply check for a valid variable name (using the built-in `atom` clause, for example).

A valid symbol table consists of a subset of these variables.

```
table(Psi) :-
  setof(V,variable(V),Vs), subset(Vs,Psi).
```

6.1 Memory addresses – natural numbers

Numerical memory addresses (natural numbers) may be handled in such a way that they need not necessarily be deterministically instantiated when a compiling clause is called. In the definition of `nat` below, if the parameter is already instantiated as a natural number, the clause succeeds immediately. If the parameter is not instantiated, then successive natural numbers are returned at each invocation. This allows non-deterministic compilation to take place.

```
nat(0).
nat(X) :- integer(X), 0<X.
nat(X) :- var(X), nat(X_1), X is X_1+1.
```

(`integer` is a standard Prolog clause which checks for an instantiated integer value and `var` checks for an uninstantiated variable.)

An equivalent simpler but less efficient encoding is:

```
nat(0).
nat(X) :- nat(X_1), X is X_1+1.
```

Here instantiated values are checked much less efficiently, especially for large numbers.

A successor function succ for natural numbers is also needed in the compiling clauses above. The following encoding allows the clause to be used for fully instantiated values, partially instantiated values and totally uninstantiated values, in which case successive successor pairs are returned.

```
succ(X,X1) :- integer(X), integer(X1), X+1=:=X1.
succ(X,X1) :- integer(X), var(X1), X1 is X+1.
succ(X,X1) :- var(X), integer(X1), X is X1-1.
succ(X,X1) :- var(X), var(X1), nat(X), X1 is X+1.
```

(=:= performs arithmetic comparison on two fully instantiated expressions; is unifies a variable to the value of a fully instantiated arithmetic expression.)

A comparison relation may be applied to any two variables by ensuring that the variables are instantiated to numerical values before applying the standard Prolog arithmetic comparison test.

```
ensure(X<=Y) :- nat(X), nat(Y), X=<Y.
ensure(X<Y)  :- nat(X), nat(Y), X<Y.
```

Many of the comparisons in the compiling clauses are in fact redundant in practice but are included anyway to match the compiling specification more closely.

Different encodings for nat, succ and <= can produce varying results in practice because the ordering of possible results returned could be affected.

6.2 Object code – sequences

Concatenation of object code may be achieved in two ways. If the target code is unknown as in the case of standard compilation then the constituent parts of code are simply concatenated to form a flattened version of the entire code. If the target code is already known then this code is split into all possible subsequences instead in order to match the code given in the compiling clauses. In each case, the structure of the constituent parts is known.

```
concat(S,T) :- nonvar(S), var(T), flatten(S,T).
concat(S,T) :- nonvar(S), nonvar(T), split(T,S).
```

(nonvar checks for a (possibly partially) instantiated variable.)

```
flatten([],[]).
flatten([X|S],T) :- flatten(S,U), append(X,U,T).
```

```
split([],[]).
split(T,[X|S]) :- append(X,U,T), split(U,S).
```

6.3 Sets

Sets may be modelled by lists. Set membership can be checked:

```
member(X,[X|_]).
member(X,[_|L]) :- member(X,L).
```

All possible subsets of a (finite) ordered set may be returned. The following implementation ensures that elements are not repeated in the list:

```
subset([],[]).
subset([_|L],R) :- subset(L,R).
subset([X|L],[X|R]) :- subset(L,R).
```

Further operations on sets can be encoded in Prolog, but these are all that are necessary here.

7 Decompilation

The *Dictionary of Computing* (Illingworth, 1990) gives the following (abridged) definitions for a *compiler* and *decompiler*:

> **compiler** A program that translates high-level language into absolute code ...
>
> **decompiler** A program that attempts to ... translate back from machine code to something resembling the source language. The task is difficult and not often attempted.

This section attempts to show instead that decompilation is almost as easy as forward compilation; the amount of Prolog code required is not significantly more complex.

7.1 Prolog clauses

Each Prolog clause, or set of clauses, specifies a relation between its parameters. In the case of the 'c' compiling clauses, the source program is normally an input and the object code is normally an output. However because of the backtracking properties of Prolog, there is no reason in theory why this operation should not be performed in reverse. In practice, it is difficult to ensure termination of the Prolog program for all possible values of parameters supplied to a Prolog clause because the depth first strategy of the Prolog interpreter may easily attempt to search down an infinite tree. Some reordering of clauses is necessary to avoid this problem.

Some built-in Prolog clauses (e.g., the '`is`' clause) are not invertible and expect instantiated values for some of the parameters. Here the clauses `nat` and `succ` have been defined to avoid such clauses and perform invertible arithmetic allowing some degree of flexibility in which parameters are instantiated.

Another problem in the case of the compiler is that the number of programs which could compile into a particular piece of object code is very large; in fact it is infinite. For example:

- `SKIP` can be combined using sequential composition at any point in a program without changing its semantics;
- Any piece of object code could be decompiled to the program `ABORT`; any validly compiled program construct could be replaced by `ABORT` during decompilation since `ABORT` is the bottom element of the refinement ordering of programs;
- Unused variables can always be declared with no change in object code;
- The non-deterministic compilation of $P \sqcap Q$ allows any program to be supplied as an alternative.

Thus it is sensible to apply the theorems selectively in a decompiler so that not *all* possible programs are returned (i.e., only an 'interesting' subset is selected) and the user is not overwhelmed with uninteresting possibilities.

For example, the theorem compiling `skip` to the empty sequence should be applied very selectively; certainly not in sequential composition to avoid infinite sequences of `skip`, but necessarily in the `if` and `while` constructs to allow for programs such as `if a then b:=c else skip`.

Theorem 1

```
cR1(skip,[]/S:S,_).
```

On the other hand, it is necessary to allow a forward jump to decompile to `skip` even when considering sequential composition in case the code includes such jumps.

Theorem 1a

```
cR2(skip,[S->jump(F)]/S:F,_) :-
  ensure(S<F).
```

For decompilation, it is sensible to split the memory first and then attempt pattern matching to avoid backtracking.

Theorem 3

```
cR2(if B then P else Q,M/S:F,Psi) :-
  concat([[S->load(PsiB),S1->cond(J)],M1,[J_1->jump(F)],M2],M),
  succ(S,S1), succ(S1,S2), succ(J_1,J),
  psi(B->PsiB,Psi),
  cR12(P,M1/S2:J_1,Psi),
  cR12(Q,M2/J:F,Psi),
  ensure(S2<J), ensure(J<=F).
```

Theorem 4

```
cR2(while B do P,M/S:F,Psi) :-
  concat([[S->load(PsiB),S1->cond(F)],M1,[F_1->jump(S)]],M),
  succ(S,S1), succ(S1,S2), succ(F_1,F),
  psi(B->PsiB,Psi),
  cR12(P,M1/S2:F_1,Psi),
  ensure(S2<=F_1).
```

Some clauses are identical (apart from the name) to the clauses used for forward compilation.

Theorem 8

```
cR2(X:=Y,[S->load(PsiY),S1->store(PsiX)]/S:S2,Psi) :-
  succ(S,S1), succ(S1,S2),
  psi(Y->PsiY,Psi),
  psi(X->PsiX,Psi).
```

Despite the problems of asserting an extra clause for recursive program compilation, this works equally well for decompilation; indeed, the extra condition `J1<=F_1` which could not be included in the forward compiler because the value of `F_1` was unknown, can be included here since `F_1` is known beforehand.

Theorem 9

```
cR2(mu X:P,M/S:F,Psi) :-
  concat([[S->push(F)],M1,[F_1->return]],M),
    succ(S,S1), succ(F_1,F),
    ident(X->S1),
    ((cR2(X,[J->subr(S1)]/J:J1,_) :-
        succ(J,J1),ensure(S1<=J),ensure(J<F_1)))
          => cR12(P,M1/S1:F_1,Psi).
```

During decompilation of sequential composition, it is desirable to avoid considering empty sequences of code since these can only decompile to `skip` and can do so infinitely often. In addition, this allows the pattern matching abilities of Prolog to ascertain the intermediate (implicitly existentially quantified) value of `J` directly from the code itself.

Theorem 2

```
cR2(P;Q,M/S:F,Psi) :-
  concat([[S->I1|M1],[J->I2|M2]],M),
  cR2(P,[S->I1|M1]/S:J,Psi),
  cR2(Q,[J->I2|M2]/J:F,Psi),
  ensure(S<=J), ensure(J<=F).
```

By applying the theorem for sequential composition last, all other object code patterns may be checked first, thus improving the efficiency of the search.

Adding variable declarations should only be done at the outer level and is thus specified as a separate clause.

Theorem 7

```
cR3(var V;P;end V,M/S:F,Phi) :-
  fresh(V,Phi,Psi),
  cR123(P,M/S:F,Psi).
```

In the last resort, `abort` may be returned for any (including invalid) code at the topmost level.

Theorem 6

```
cR4(abort,_,_).
```

The non-deterministic compilation of $P \sqcap Q$ is not useful in decompilation since it can add any arbitrary program; thus it is not included.

7.2 Combinations of theorems

The following combinations of theorems are used in the decompilation clauses above:

Theorems 1 & 1a, 3, 4, 8, 9, 2

```
cR12(P,M/S:F,Psi) :- cR1(P,M/S:F,Psi).
cR12(P,M/S:F,Psi) :- cR2(P,M/S:F,Psi).
```

Variable declarations are only added at the outermost level:

All previous theorems & theorem 7

```
cR123(P,M/S:F,Psi) :- cR12(P,M/S:F,Psi).
cR123(P,M/S:F,Psi) :- cR3(P,M/S:F,Psi).
```

At the topmost level, all the theorems, including that for the program `ABORT`, may be applied:

All previous theorems & theorem 6

```
cR(P,M/S:F,Psi) :- cR123(P,M/S:F,Psi).
cR(P,M/S:F,Psi) :- cR4(P,M/S:F,Psi).
```

7.3 An alternative approach

An alternative and interesting coding for the theorems is to record the contents of memory as extra assertions. Thus the result of running the compiler is to add the object code to the database of Prolog clauses using the Prolog `assert` clause. For decompilation, the object code must first be added to the Prolog database of clauses, and then decompiler is run to generates the equivalent high-level program(s)

For compilation, more care must be taken to ensure that the memory is asserted at a time when the relevant parameters are instantiated. This is only a problem for the `if` and `while` clauses where jump locations are not known until the intervening program has been compiled. Judicious reordering of the clauses (allowed because of the commutativity of conjunction) enables these to be available at the correct time, under the procedural reading of the Prolog program.

For further details of such an encoding, see (Hoare *et al.*, 1990b).

8 Example output

The following are sample runs of some of the previously presented code using Quintus Prolog (Quintus, 1990).

Given arbitrary uninstantiated parameters, the compiling clause tries to return all possible programs. In practice, since Prolog performs a depth-first search of its database of clauses and `skip` can be compiled in infinitely different ways, only programs for the `skip` theorems are returned.

```
| ?- c(P,M,Psi).

P = skip,
M = []/_590:_590,
Psi = _470 ;

P = skip,
M = [0->jump(1)]/0:1,
Psi = _470 ;

P = skip,
M = [0->jump(2)]/0:2,
Psi = _470
```

The symbol table Ψ is arbitrary. Prolog indicates this with `Psi = _`n where n is some internal variable number.

Consider a more specified program:

```
| ?- c(if a then b:=c else c:=b,M/0:F,[a,b,c]).

M = [0->load(psi(a)),1->cond(5),
     2->load(psi(c)),3->store(psi(b)),4->jump(7),
     5->load(psi(b)),6->store(psi(c))],
F = 7 ;

no
```

Here ‘`a`’, ‘`b`’ and ‘`c`’ are supplied as being in the symbol table and there is only one possible compilation allowed (ignoring the allocation of memory addresses for variables in the symbol table).

Code containing a `skip` can include gaps. For example, the third compilation in the list of possibilities below has no code at location 3:

```
| ?- c(while b do skip,M/0:F,_).

M = [0->load(psi(b)),1->cond(3),2->jump(0)],
F = 3 ;

M = [0->load(psi(b)),1->cond(4),2->jump(3),3->jump(0)],
F = 4 ;

M = [0->load(psi(b)),1->cond(5),2->jump(4),4->jump(0)],
F = 5
```

Give a non-deterministic choice of two programs, either may be compiled:

```
| ?- c((a:=b)\/(b:=a),M/0:F,[a,b]).

M = [0->load(psi(b)),1->store(psi(a))],
F = 2 ;

M = [0->load(psi(a)),1->store(psi(b))],
F = 2 ;

no
```

Variables can be (and normally are) explicitly declared in the program. For example:

```
| ?- c(var a;(var b; a:=b; end b);end a,M/0:F,[]).

M = [0->load(psi(b)),1->store(psi(a))],
F = 2 ;

no
```

It is possible to vary the parameters that are supplied and those which are calculated. For example, asking for a program to fit into a given size of memory results in Prolog searching for all such programs:

```
| ?- c(P,M/0:10,_).

P = skip,
M = [0->jump(10)] ;

P = skip;skip,
M = [0->jump(10)] ;

P = skip;skip;skip,
M = [0->jump(10)]
```

On compiling a recursive program, a new compiling clause is added to the database:

```
| ?- c(mu x:x,M/0:F,_).

M = [0->push(3),1->subr(1),2->return],
F = 3 ;

no
| ?- c(x,M/S:F,_).

M = [1->subr(1)],
S = 1,
F = 2 ;

M = [2->subr(1)],
S = 2,
F = 3 ;

M = [3->subr(1)],
S = 3,
F = 4
```

Decompiling this code gives:

```
| ?- cR(P,[0->push(3),1->subr(1),2->return]/_:_,_).

P = mu$1: $1 ;

P = var a;mu$1: $1;end a ;

P = var a;(var b;mu$1: $1;end b);end a
```

As can be seen above, arbitrary redundant variable declarations may be added during decompilation.

Decompilation will in general list a number of possibilities. For example, sequential composition can be bracketed in any order and ABORT is always a valid program:

```
| ?- cR(P,[0->load(psi(b)),1->store(psi(a)),
           2->load(psi(c)),3->store(psi(b)),
           4->load(psi(a)),5->store(psi(c))]/_:_,[a,b,c]).
```

```
P = a:=b;b:=c;c:=a ;

P = (a:=b;b:=c);c:=a ;

P = abort ;

no
```

9 Constraint Logic Programming

Some Prolog clauses are not reversible. For example, the `is` construct allows arithmetic expressions to be evaluated in a single direction. Here the clauses `nat` and `succ` have been defined to avoid this problem. However there is no way in Prolog to define a number (for example) which is constrained in some way (e.g., $x > 1$). The number must be enumerated for each instance which is valid (e.g., $x = 1$, $x = 2$, $x = 3$, etc.). Such enumerations can often be infinite (see for example, theorem 1a for the compilation of `SKIP` using a forward jump instruction. This can easily result in non-termination of the Prolog program if care is not taken.

This deficiency could be remedied by a newly emerging field, namely '*Constraint Logic Programming*' which is an extension of the standard Logic Programming paradigm. Unification is replaced by a more general mechanism of constraint satisfaction in which a set of (in)equations (constraints which may be added to each clause) must be solved. A given domain or set of domains (e.g., CLP($\Re$) for the real numbers) is covered by a particular CLP system. Such systems are now reasonably efficient and becoming more generally available (e.g., (Colmerauer, 1990)). A useful overview and a good up-to-date list of references to this field may be found in (Cohen, 1990).

9.1 Meta-level interpretation

Briefly, a core *logic programming* interpreter looks like the following (in Prolog):

```
solve([]).

solve([Goal|Restgoal]) :-
  solve(Goal),
  solve(Restgoal).
solve(Goal) :-
  clause(Goal,Body),
  solve(Body).
```

In this example, clauses of the form '`Goal :- Body.`' are assumed to be stored as `clause(Goal,Body)`.

In a constraint logic programming language, clauses have extra *constraints* associated with them:

```
Goal :- Body {Constraints}.
```

A *constraint logic programming* interpreter is augmented to handle these constraints as follows:

```
solve([],C,C).

solve([Goal|Restgoal],PrevC,NewC) :-
  solve(Goal,PrevC,TempC),
  solve(Restgoal,TempC,NewC).

solve(Goal,PrevC,NewC) :-
  clause(Goal,Body,Constraints),
  merge(PrevC,Constraints,TempC),
  solve(Body,TempC,NewC).
```

The sets of constraints must be maintained and updated as each clause is encountered. The `merge` clause conjoins the previous constraints with the constraints of the current clause. This is where most of the extra programming effort is required in implementing a CLP system in practice.

9.2 A CLP compiler/decompiler

Part of a simple CLP compiler/decompiler has been implemented using Prolog III:

```
cc(skip, <>, S, S, T) -> ;

cc(skip, <S,jump(F)>, S, F, T) ->
   {S<F};

cc(if(B,P,Q), <S,load(T,B), S+1,cond(J)>.M1.<J-1,jump(F)>.M2,
     S, F, T) ->
   cc(P, M1, S+2, J-1, T)
   cc(Q, M2, J, F, T)
   {S+2<J, J<=F} ;

cc(while(B,P), <S,load(T,B), S+1,cond(J)>.M.<F-1,jump(S)>,
     S, F, T) ->
   cc(P, M, S+2, F-1, T)
   {S+2<F};

cc(abort, M, S, F, T) -> ;

cc(assign(X,Y), <S,load(T,Y), S+1,store(T,X)>, S, S+2, T) -> ;

cc(mu(X, P), <S,push(F)>.M.<F-1,return>, S, F, T) ->
   assert(cc(X, <J,subr(S+1)>, J, J+1, T), [])
   cc(P, M, S+1, F-1, T)
   {S<J, J<F-1};

cc(<>, M, S, F, T ) ->
   cc(skip, M, S, F, T);

cc(<P>.R, M1.M2, S, F, T) ->
   cc(P, M1, S, J, T)
   cc(R, M2, J, F, T)
   {S<=J, J<=F};
```

These clauses do indeed execute in both directions without the necessity to have different clauses for the compiler and the decompiler. However the clauses should still be applied selectively in the reverse direction to ensure program termination.

9.3 Sample runs

The following examples demonstrate the output obtained by running the program shown above:

```
> cc(skip, M, S, F, T);
{M = <>, F = S}
{M = <S_2,jump(S_2 + S$5_2)>, S = S_2, F = S_2 + S$5_2,
        S$5_2 > 0 }

> cc(assign(x,y), M, S, F, T);
{M = <S_1,load(T,y),S_1 + 1,store(T,x)>, S = S_1, F = S_1 + 2}
```

```
> cc(P, <S_1,load(T,y), S_1+1,store(T,x)>, S, F, T);
{P = abort,
       S_1!num }
{P = assign(x,y), S = S_1, F = S_1 + 2}
```

In particular, programs that output an infinite number of answers by enumerating all the possible answers in turn (such as when SKIP is compiled non-deterministically), can output a finite answer, simply giving the constraints involved instead (see above).

10 Conclusions

The complete code for the example compiler and decompiler is presented here. It can be seen that the entire program for each is not a great deal longer than the original specification, even including the support routines; and certainly the reverse compiler is not significantly more complicated than the more normal forward compiler.

The subset of Prolog used is relatively pure (e.g., there are no cuts and negation is used very sparingly (just once!) and only when it is 'safe' to do so to maintain the soundness properties). Thus the declarative semantics of Prolog (Lloyd, 1987) may be assumed to hold and it would be possible to perform a formal proof of the correctness of the Prolog compiler and decompiler which are presented here. For example, the approach of program synthesis using the proofs as programs technique presented in (Bundy *et al.*, 1990) could be (somewhat laboriously) applied. Research is very active in the area of logic program synthesis and transformation (Clement *et al.*, 1991). Less pure features used in the code presented here include `var`, `nonvar`, `integer`, `assert` and `retract`, but these are used in a very restricted manner which is intended to maintain the logical semantics as far as possible.

This paper uses a very simple programming language and instruction set as an example. However some of the ideas presented have been applied to a subset of a real programming language and microprocessor, namely occam and the **transputer** (Bowen *et al.*, 1989, Hoare *et al.*, 1990b). Whilst there is no guarantee than the prototyping method will necessarily scale up to a full real language, we believe that the approach looks promising, particularly for safety-critical applications where optimization is avoided for safety reasons.

In practice the output from an optimizing compiler will be considerably more complicated than the simple example presented here. However, extra theorems specifying different code for constructs may easily be included without affecting existing theorems (He *et al.*, 1992), and the pattern matching abilities of Prolog will still be able to unify the correct construct (or constructs). For example, the following theorem could be added:

Theorem 3a {conditional}
If $s+2 \leq j \leq f$ and $m[s] = \mathit{load}(\Psi(b))$ and $m[s+1] = \mathit{cond}(f)$ and $\mathcal{C}\, p\; m[s+2:f]\Psi$ then $\mathcal{C}(\,\texttt{IF}\; b\; \texttt{THEN}\; p) m[s:f]\Psi$

This would allow optimized conditional clauses with no `ELSE` part to be compiled and decompiled.

The efficiency of the compiler and the decompiler presented here is not great and it is envisaged that practical applications of this technique would require some optimization of the Prolog code using program transformation and data refinement techniques. However in both cases the first answer returned will normally be of greatest interest (and the program can be structured with this in mind), so it is not necessary to search through all possibilities (e.g., all the possible combinations of sequential composition). Additionally the theorems can be ordered and applied selectively for efficiency. For example, individual subroutines could be recognized and isolated early on in the decompilation process, so that smaller blocks of code may then be decompiled separately. Assuming that subroutines are all approximately of the same order of magnitude in size, the computational complexity is then linear in the number of subroutines. Indeed, if several processors are available, separate subroutines could be decompiled in parallel, thus reducing the execution time still further – essentially to a (large) constant time given enough processors.

This paper ignores the problems of parsing by using the abstract syntax directly. Fortunately, this can be made fairly readable using Prolog since infix, prefix and postfix operators with specified associativity and precedence are possible. However, in practice real compiler would need a parser (and a static semantics checker). A decompiler would require a concrete syntax to be generated from the abstract syntax produced be the decompiler presented here. Luckily this is far easier that the reverse procedure since only correct abstract syntax is generated and thus error checking is not necessary. All these extra phases are possible in Prolog, although perhaps not as efficiently as other approaches (Paakki, 1991).

10.1 Related work

Of course the above techniques, whilst useful for rapid prototyping, may not be efficient enough for practical and useful implementation in general. Other programming paradigms could go some way to alleviating this. For example, the compiler and decompiler presented here have also been implemented using a functional programming language (Breuer *et al.*, 1992a, Breuer *et al.*, 1992c). Because of the irreversible nature of functions when there are implemented on a machine, the compiler and decompiler must be defined separately. A different set of concerns arise, particularly with regard to the enumeration of sets of possible high-level programs or object codes to ensure that the required programs are enumerated fairly. This work also demonstrates the relationship with attribute grammars (Deransart *et al.*, 1987).

However a practical approach which may gain widespread use must use a more traditional, efficient and widely used language such as ANSI C (Kernighan *et al.*, 1988). Compiler-compiler technology is well established for use in generating compilers. For example, *yacc* (Johnson *et al.*, 1978) is the traditional tool used under UNIX. Implementations of *yacc* have been in existence for many years and it is somewhat restrictive, having been originally designed at a time when memory capacity of computers was much smaller and more critical. Its one-token look-ahead approach, whilst efficient, leads to inelegant parser descriptions which do not match the BNF descriptions provided for most languages directly, and cannot handle some languages that require infinite look-ahead.

More modern tools, such as *precc* (Breuer, 1992b, Breuer *et al.*, 1992b, Breuer *et al.*, 1993), allow languages like occam (INMOS Limited, 1988), which are difficult for *yacc* because of the use of indentation to indicate structure, to be handled with ease (Bowen *et al.*, 1992a). The ideas behind *precc* have been extended using the concept of a decompiler-compiler to allow decompilers to be generated more quickly and efficiently using C (Breuer *et al.*, 1992d).

The decompiler-compiler has been applied at BT Research Laboratories with investigations based around the GNU ANSI C compiler (Breuer, 1992a). Such production compilers rarely have a formal specification associated with them, so the first requirement is to formulate such a description. Obtaining a full specification is a time-consuming exercise. However, provided that optimization is not enabled, the control structure of the object code can be recovered. Rediscovering data structures is significantly more difficult, and even impossible in practice because too much information has been lost, unless the intervention and insight of an engineer is provided.

Overviews of other related work on the **ProCoS**, safemos and REDO projects may be found in (Bjørner, 1992, Bjørner *et al.*, 1992, Bowen, 1993, van Zuylen *et al.*, 1993).

10.2 Possible future directions

The decompiler could be useful in the software maintenance process if, for example, the original code had been lost. However, given the limitations concerning optimized code and complicated data structures, the techniques described here are most likely to prove useful in situations where these are normally avoided, such as in the decompilation of code for safety-critical systems for verification purposes – e.g., as in (Spector *et al.*, 1984, Pavey *et al.*, 1992).

Currently most object program debuggers provide disassembled representations of the object code to the engineer. Decompilation techniques could be used to display a higher-level reconstruction of the code which could aid the understanding of the functioning of the code. Other information could also be extracted which could give an indication of the quality of the code.

It may be possible to specify decompilation of constructs in such a way that the resulting high-level program is more structured that the original program. For instance, the original forward compiler could include `goto` statements, but the decompiler could only include more structured statements, or at least attempt to apply these in preference to less desirable constructs by ordering the Prolog clauses sensibly. Other possible refinements include removal of redundant variable declarations, unused sections of code, etc.

A Constraint Logic Programming system (Cohen, 1990, Colmerauer, 1990) could be used to improve the compiler and decompiler presented here, and maybe even to combine them into a single program. This would be a useful and interesting area of investigation. However it is likely that this will remain a research topic until optimizing compilers are available for such systems which can match the efficiency of current Prolog technology (Paakki, 1991, Quintus, 1990).

The development of techniques of *inductive logic programming* (Muggleton *et al.*, 1990), which aim at the derivation of Prolog programs from examples and background knowledge, could provide the possibility of automatically synthesising the compiling specification program from examples of triples (p, Ψ, m) of an input source program p, symbol table Ψ, and the compiled object code m. This then obviates the need for (perhaps unfounded) assumptions on the part of the decompiler writer about the semantics of the source code.

Acknowledgements

Sincere thanks are due to Prof. Tony Hoare and He Jifeng for providing the specification and inspiration to produce the ideas presented here (Hoare *et al.*, 1990a, Hoare, 1991). The idea for this work was instigated by related work on compiling specification and verification undertaken on the ESPRIT BRA **ProCoS** project and UK IED safemos project (Bowen *et al.*, 1989, Hoare *et al.*, 1990b). Members of the ESPRIT II REDO project provided further help and motivation in the form of a reason why a decompiler could be useful (van Zuylen *et al.*, 1993). The financial support of the UK Information Engineering Directorate safemos project no. IED3/1/1036 is gratefully acknowledged.

The author is indebted to members of the above collaborative projects for their support. Peter Breuer supplied copious comments on an earlier draft and has also undertaken much of the subsequent work using a functional approach. Tony Hoare and Kevin Lano also provided helpful comments. A discussion with Bob Kowalski, Keith Clark, M. Sergot, Mike Spivey, Tony Hoare and others provided some useful input.

Copies of **ProCoS** project documents are available from: Department of Com-

puter Science, Technical University of Denmark, Building 344Ø, DK-2800 Lyngby, Denmark. REDO project documents and PRG Technical Reports may be obtained from the Librarian, Oxford University Computing Laboratory, 11 Keble Road, Oxford OX1 3QD, UK. Referenced draft papers can be obtained directly from the authors.

References

Bjørner, D. (1992). 'Trusted computing systems', in *Proc. 14th International Conference on Software Engineering (ICSE)*, Melbourne, Australia, 11–14 May 1992. North-Holland.

Bjørner, D., Langmaack, H. and Hoare, C.A.R. (1992). *Provably Correct Systems.* **ProCoS** Project Technical Report, Department of Computer Science, DTH, Lyngby, Denmark.

Bowen, J.P. (1992). 'From programs to object code using logic and logic programming', in Giegerich, R. and Graham, S.L. (Eds.), *Code Generation – Concepts, Tools, Techniques*, Proc. International Workshop on Code Generation, Dagstuhl, Germany, 20–24 May 1991. Springer-Verlag, Workshops in Computing, pp. 173–192.

Bowen, J.P. (Ed.) (1993). *Towards System Verification.* Elsevier, Real-Time Safety-Critical Systems series. In preparation.

Bowen, J.P., He Jifeng and Pandya, P.K. (1990). 'An approach to verifiable compiling specification and prototyping', in Deransart, P. and Małuszyński, J. (Eds.), *Programming Language Implementation and Logic Programming, International Workshop PLILP 90*, Springer-Verlag, LNCS **456**, 45–59.

Bowen, J.P. and Breuer, P.T. (1992). 'Occam's Razor: the cutting edge of parser technology', in *Proc. TOULOUSE 92: Fifth International Conference on Software Engineering and its Applications*, Toulouse, France, 7–11 December 1992.

Bowen, J.P. and Breuer, P.T. (1993a). 'Decompilation', Chapter 9 in (van Zuylen *et al.*, 1993).

Bowen, J.P. and Stavridou, V. (1993b). 'Safety-critical systems, formal methods and standards', *Software Engineering Journal*, 1993. To appear. Also issued as a Programming Research Group Technical Report PRG-TR-5-92.

Breuer, P.T. and Lano, K.C. (1991). 'Creating Specifications from Code: Reverse Engineering Techniques', *Journal of Software Maintenance*, **3**, 145–162.

Breuer, P.T. (1992a). 'Report on decompilation', Ref. no. MAINT/MISC/056, Issue 1.1, British Telecommunications plc, BT Research Laboratories, Martlesham Heath, Ipswich, Suffolk, UK.

Breuer, P.T. (1992b). 'A *PRE*ttier *C*ompiler-*C*ompiler: higher order programming in C', in *Proc. TOULOUSE 92: Fifth International Conference on Software Engineering and its Applications*, Toulouse, France, 7–11 December 1992.

Breuer, P.T. and Bowen, J.P. (1992a). *Decompilation: the Enumeration of Types and Grammars.* Oxford University Computing Laboratory, Programming Research Group Technical Report PRG-TR-11-92. Submitted for publication.

Breuer, P.T. and Bowen, J.P. (1992b). *A PREttier Compiler-Compiler: Generating Higher Order Parsers in C.* Oxford University Computing Laboratory, Programming Research Group Technical Report PRG-TR-20-92. Submitted for publication.

Breuer, P.T. and Bowen, J.P. (1992c). 'Decompilation *is* the efficient enumeration of types', in Billaud, M. *et al.* (Eds.), *Journées de Travail WSA'92 Analyse Statique*, BIGRE **81–82**, IRISA-Campus de Beaulieu, F-35042 Rennes cedex, France, pp. 255–273 (1992).

Breuer, P.T. and Bowen, J.P. (1992d). 'Generating Decompilers', Draft, Oxford University Computing Laboratory. Submitted for publication.

Breuer, P.T. and Bowen, J.P. (1993). 'The PRECC compiler-compiler', in Davies, E. and Findlay, A. (Eds.), *Proc. UKUUG/SUKUG Joint New Year 1993 Conference*, St. Cross Centre, Oxford, UK, 6–8 January 1993. UKUUG/SUKUG Secretariat, Owles Hall, Buntingford, Herts SG9 9PL, UK, 167–182.

Bundy, A., Smaill, A. and Wiggins, G. (1990). 'The Synthesis of Logic Programs from Inductive Proofs', in Lloyd, J.W. (Ed.), *Computational Logic*, Springer-Verlag, Basic Research Series, 135–149.

Clement, T.P. and Lau, K.-K. (Eds.) (1992). *Logic Program Synthesis and Transformation*, Proc. LOPSTR 91, International Workshop on Logic Program Synthesis and Transformation, University of Manchester, 4–5 July 1991. Springer-Verlag, Workshops in Computing.

Clocksin, W.F. and Mellish, C.S. (1987). *Programming in Prolog*, 3rd edn. Springer-Verlag.

Clutterbuck, D.L. and Carré, B.A. (1988). 'The verification of low-level code', *Software Engineering Journal*, **3** 3, 97–111.

Cohen, J. (1990). 'Constraint Logic Programming languages', *Communications of the ACM*, **33** 7, 52–68.

Colmerauer, A. (1990). 'An introduction to Prolog III', *Communications of the ACM*, **33** 7, 69–90.

Deransart, P. and Małuszyński, J. (1987). 'Relating logic programs and attribute grammars', *Journal of Logic Programming*, **3** 2, 125–163.

He Jifeng and Bowen, J.P. (1992). 'Specification, verification and prototyping of an optimized compiler', Draft, Oxford University Computing Laboratory. Submitted for publication.

Hoare, C.A.R. (1991). 'Refinement algebra proves correctness of compiling specifications', in Morgan, C.C. and Woodcock, J.C.P. (Eds.), *3rd Refinement Workshop.* Springer-Verlag, Workshops in Computing, 33–48.

Hoare, C.A.R. and He Jifeng (1990). 'Refinement algebra proves correctness of compilation', **ProCoS** Project Document [OU HJF 7].

Hoare, C.A.R., He Jifeng, Bowen, J.P. and Pandya, P.K. (1990). 'An algebraic approach to verifiable compiling specification and prototyping of the ProCoS level 0 programming language', in Directorate-General of the Commission of the European Communities (Eds.), *ESPRIT '90 Conference Proceedings*, Brussels. Kluwer Academic Publishers B.V., 804–818.

Hogger, C.J. (1990). *Essentials of Logic Programming.* Oxford University Press.

Illingworth, V. (Ed.) (1990). *Dictionary of Computing*, 3rd edn. Oxford University Press.

INMOS Limited (1988). *Occam 2 Reference Manual.* Prentice Hall International Series in Computer Science.

Kernighan, B.W. and Ritchie, D.M. (1988). *The C Programming Language*, 2nd edn. Prentice-Hall Software Series.

Johnson, S.C. and Lesk, M.E. (1978). 'Language development tools', *The Bell System Technical Journal*, **57** 6 part 2, 2155–2175.

Lano, K.C. and Breuer, P.T. (1990). 'From Programs to Z Specifications', in Nicholls, J.E. (Ed.), *Z User Workshop, Oxford 1989.* Springer-Verlag, Workshops in Computing, 46–70.

Lloyd, J.W. (1987). *Foundations of Logic Programming*, 2nd edn. Springer-Verlag.

McCarty, L.T. (1988). 'Clausal intuitionistic logic. I. Fixed-point semantics.', *Journal of Logic Programming*, **5** 1, 1–31.

Miller, D. (1989). 'A logical analysis of modules in logic programming', *Journal of Logic Programming*, **6**, 79–108.

Muggleton, S. and Feng, C. (1990). 'Efficient induction of logic programs', *Proc. 1st Conference on Algorithmic Learning*, Tokyo, Japan, October 1990. Ohmsha Publications, Japan.

Paakki, J. (1991). 'Prolog in practical compiler writing', *The Computer Journal*, **34** 1, 64–72.

Pavey, D.J. and Winsborrow, L.A. (1992). 'Demonstrating equivalence of source code and PROM contents', *4th European Workshop on Dependable Computing*, Prague, Czechoslovakia, 8–10 April 1992.

Quintus (1990). *Quintus Prolog – Sun 4 User Manual*, Release 2.5. Quintus Computer Systems, Inc., Mountain View, California, USA.

Samuelson, P. (1990). 'Reverse-engineering someone else's software: is it legal', *IEEE Software*, January 1990, p 90.

Spector, A. and Gifford, D. (1984). 'Case study: the space shuttle primary computer system', *Communications of the ACM*, **27** 9, 872–900.

UK Government (1992). *The Copyright (Computer Programs) Regulations 1992*, Ref. no. SI 1992/3233, HMSO Publications, PO Box 276, London SW8 5DT, UK.

van Emden, M.H. and Kowalski, R.A. (1976). 'The semantics of predicate logic as a programming language', *Journal of the ACM*, **23** 4, 733–742.

van Zuylen, H. (Ed.) (1993). *The REDO Compendium: Reverse Engineering for Software Maintenance*. John Wiley and Sons Ltd.

Ward, M., Calliss, F.W., Munro, M. (1989). 'The maintainer's assistant', in *Proc. Conference on Software Maintenance*, Miami, USA. IEEE Computer Society Press, New York, 307–315.

Warren, D.H.D. (1980). 'Logic programming and compiler writing', *Software—Practice and Experience*, **10**, 97–125.