From Programs to Object Code using Logic and Logic Programming

Jonathan Bowen
Oxford University Computing Laboratory
Programming Research Group
11 Keble Road, Oxford OX1 3QD, England
E-mail: Jonathan.Bowen@comlab.ox.ac.uk

Abstract
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. If the machine that interprets the object code is specified in the source language itself, then the compiler may be verified using algebraic laws about the programming language constructs. By adopting a subset of the programming language OCCAM, we can benefit from the large number of existing laws which have already been proved for this language.

The compiling specification theorems are all Horn clauses in general. Thus it is possible to produce an executable compiler prototype almost directly from this specification in the form of a logic program. The target object code for the transputer has variable-length instruction sequences. Thus in some cases it is necessary to transform the theorems into a form which can be more efficiently executed by avoiding unnecessary backtracking, particularly when there are jumps in the code. However the relational nature of logic programming allows a number of solutions to be returned if desired.

1 Introduction
A compiler may be specified by a set of theorems, each relating a construct in the high-level programming language with the allowed machine instruction sequence that implements that construct. In general the two descriptions act on a different data space (i.e., program variables and memory locations). Thus a symbol table describing this relationship will also be needed.

Hoare has suggested a novel way of verifying such a description [1]. This involves specifying the semantics of the low-level machine using an interpreter written in the high-level programming language (or an extension of it if necessary). This allows algebraic laws about the language to be applied to gradually transform one description into the other. These laws may either be held to be self-evident, or can themselves be proved correct with respect to another semantics [2].

This technique has been applied to a subset of the programming language OCCAM [3] and the transputer [4], both of which are in industrial use. Using OCCAM means that the well-understood semantics and algebraic laws already formulated for this language [5] can be readily adapted and applied to the verification proof. This work has been adequately described elsewhere [6, 7, 8, 9, 10].
8]. In addition the language has been extended to include more complicated constructs such as recursion, to demonstrate that the technique is applicable to non-trivial programming constructs [9]. This work forms the basis for an even more ambitious task that is currently being undertaken, to prove the correctness of a compiler written in the high-level language [10].

The compiling specification theorems are in general formulated as Horn clauses. Thus they are already very close in form to a logic program. In some cases the programs may be transliterated almost directly from the specification. In other cases, the specifications may be transformed using logic into a form that is more efficiently executable. The rest of this paper describes this process, and some of the particular problems encountered in generating code for the transputer. The minimum development to achieve a reasonably efficient executable program is carried out and a rigorous argument [11] is presented to help ensure the correctness of the compiler.

1.1 Logic programming and Prolog

As the name suggests, logic programming has a well established mathematical basis [12, 13, 14]. Prolog [15] is the most widely available logic programming language. However, Prolog includes many non-logical features in an attempt to make it into a usable, practical and efficiently executable 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

\[ P :~ Q_1, \ldots, Q_n. \]

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

\[ \forall x, y, \ldots \cdot ((Q_1 \land \ldots \land Q_n) \Rightarrow P) \]

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 (\( \Leftarrow \) or "if") in predicate logic. This theorem is in turn equivalent to:

\[ \forall x_1, \ldots, x_i \cdot ((\exists y_1, \ldots, y_m \cdot Q_1 \land \ldots \land Q_n) \Rightarrow P) \]

where \( x_1, \ldots, x_i \) 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 \( \text{true} \) and the implication may be omitted since \( \text{true} \Rightarrow P = P \). Such formulae, in which all the variables are set to some specific value are known as facts [15].

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:

\[ ?- G_1, \ldots, G_n. \]

This is equivalent to the following in predicate logic:

\[ \neg \exists y_1, \ldots, y_m \cdot (G_1 \land \ldots \land G_n) \]

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