Formal Aspects of Computing (1994) 6: 643–658

Formal Aspects of Computing

Specification, Verification and Prototyping of an Optimized Compiler

He Jifeng [1] and Jonathan Bowen [2]

Oxford University Computing Laboratory, Programming Research Group, UK

Keywords: Program compilation; Code optimization; Formal verification; Refinement algebra; Logic programming

Abstract. This paper generalizes an algebraic method for the design of a correct compiler to tackle specification and verification of an optimized compiler. The main optimization issues of concern here include the use of existing contents of registers where possible and the identification of common expressions. A register table is introduced in the compiling specification predicates to map each register to an expression whose value is held by it. We define different kinds of predicates to specify compilation of programs, expressions and Boolean tests. A set of theorems relating to these predicates, acting as a correct compiling specification, are presented and an example proof within the refinement algebra of the programming language is given. Based on these theorems, a prototype compiler in Prolog is produced.

1. Introduction

The development of computer-based systems can benefit from a formal approach at all levels of abstraction from requirements through to design, compilation and hardware. Two related collaborative research projects, the **ProCoS** [Bjø92, Bow93b] and safemos [Bow94] projects, have investigated formal techniques to handle these various levels of abstraction, and crucially how they relate to

Correspondence and offprint requests to: J. P. Bowen, Oxford University Computing Laboratory, Programming Research Group, Wolfson Building, Parks Road, Oxford OX1 3QD, UK. Email: Jonathan.Bowen@comlab.ox.ac.uk

[1] Funded by ESPRIT Basic Research **ProCoS** project (nos 3104 & 7071).

[2] Funded by UK Engineering and Physical Sciences Research Council (SAFEMOS project IED3/1/1036 and grant no. GR/J15186).

one another [BFO93]. This paper concentrates on the automatic compilation of a high-level executable source program to a low-level machine code based on the ideas in [BHP90, Hoa91, HoH92, HHB90]. Previously this has been extended to handle a real-time language [HeB92]. Here we investigate how code optimizations can be included in the process.

A compiler takes as input a source program and produces as output an equivalent (or better) sequence of machine instructions wrt. some refinement ordering. Additionally, target program sequences that are frequently executed should be fast and small. Since this process is so complex, it is customary to partition the compilation process into a series of subprocesses called phases. Certain compilers have within them a phase that tries to apply transformations to the source code or the output of the intermediate code generator, in an attempt to produce a faster or smaller machine code. This phase is popularly called the *optimization phase*. Since code optimization is intertwined with code generation, it does not make sense to do a good job of code generation without also doing a good job of code optimization.

As is widely known, one of the richest sources of optimization is in the efficient utilization of the registers and instruction set of a machine [ASU86]. This aspect of optimization is closely connected with code generation, and many issues in this area are highly machine dependent. An additional important source of optimization is the identification of common expressions and the replacement of run-time computations by compile-time computations.

The formalization and verification of code generation optimization does not seem to be well advanced. It has been noted that no proof techniques are available for code generation techniques that are actually used in practice [GiG92]. Realistic optimized compiling schemes have been formally specified but not verified [Bun82]. Where formal development has been undertaken, it has normally been for unoptimized code [Cur93, Ste93, SWC91]. Optimization has often been avoided in safety-critical and other high integrity systems since it can be an extra source of error, although the use of formal methods could help [BoS93]. This paper will take these issues into account in the design of a correct compiler. Other related work in this area has been undertaken in parallel but independently [Lev92].

As advocated by Hoare [Hoa91], a compiler can be specified as a set of theorems, each describing how a construct in the programming language is translated into a sequence of machine instructions. Central to that approach is a predicate $\mathscr{C}\, q\, s\, f\, m\, \Psi\, \Omega$ stating that the machine code stored in the memory m with s as the start address and f as the finish address is a correct translation of the source program q where Ψ is the symbol table mapping each global variable of q to a location in the machine memory where its value is being stored, and Ω is the free storage which can be used to store the value of local variables and the temporary results during the execution of expressions. The compiling specification is given as a set of theorems about the predicate $\mathscr{C}\, q\, s\, f\, m\, \Psi\, \Omega$ stating how each construct can be compiled. To verify the correctness of compiling specification, a mathematical theory of program refinement is developed to establish an improvement relation $\sqsubseteq$ between programs p and q which states that q is better than p in all circumstances.

Following such an approach, this paper defines a new predicate

$$\mathscr{CP}\, q\, s\, f\, m\, \Psi\, \Omega\, \Phi\, \dot{\Phi}$$

with two new parameters Φ and $\dot{\Phi}$ mapping each register to the expressions whose value is held by it before and after execution respectively, to replace the predicate $\mathscr{C}\, q\, s\, f\, m\, \Psi\, \Omega$. Another predicate

$$\mathscr{CE}\, e\, s\, f\, m\, \Psi\, \Omega\, \Phi\, \dot{\Phi}$$

stating that m contains a correct implementation of expression e, is present to support common expression optimization. Finally, we propose a predicate

$$\mathscr{CBE}\, b\, s\, f\, m\, \Psi\, \Omega\, \Phi\, \dot{\Phi}\, \{true \mapsto tl, false \mapsto fl\}$$

to compile a Boolean test b (in both conditional and iteration constructs) into an optimized target code by assigning exit addresses tl and fl in advance.

This paper will present a set of theorems relating predicates $\mathscr{CP}$, $\mathscr{CE}$ and $\mathscr{CBE}$ and provide some examples of verification of these theorems with the help of a refinement algebra developed to specify an algebraic semantics of the programming language. Based on that set of theorems, a prototype compiler is then produced in a very direct manner using Prolog [ClM87].

2. Refinement Algebra

This paper examines a simple programming language which contains assignment, sequential composition, conditional and iteration constructs, and declaration and scoping of variables. In the design of a correct compiler the first and absolute requirement is a perfect comprehension of the meaning of the source and target languages. If the implementation is to be supported by a mathematical proof, these meanings must be expressed by some mathematical definition which forms the basis of the reasoning. A wide variety of formalisms have been proposed for this purpose, and there is difficulty in choosing between them. We suggest the use of a complete set of laws as an algebraic specification of the meaning of the programming language. The sufficiency of such a set of laws can be established by an appropriate kind of normal form theorem. One of the advantages of algebraic laws is that of modularity and generality: each of them is valid in many programming languages; and they often remain valid when the language is extended. The basic laws defining the programming language used in this paper are given in [HoH92]. Some of the more useful laws are repeated here for convenience. We take the simplifying view that all expressions always deliver a value (i.e., no error can occur during the evaluation of an expression).

Sequential composition has SKIP as its unit, and distributes left over conditional.

Law 1.
(1) SKIP ; q = q = q ; SKIP.
(2) $(q \lhd b \rhd r)\, ;\, w = (q\, ;\, w) \lhd b \rhd (r\, ;\, w)$.

We define an *improvement* relation between programs p and q that holds whenever for any purpose the behaviour of q is as good as or better than that of p; more precisely, if q satisfies every specification satisfied by p, and maybe more. This relation is written $p \sqsubseteq q$. $\sqsubseteq$ is a partial order; i.e., it is reflexive, transitive and antisymmetric. The program ABORT represents the completely arbitrary behaviour of a broken machine, and is the least controllable and predictable program; i.e., it is the bottom of $\sqsubseteq$.

Law 2. $\texttt{ABORT} \sqsubseteq q$.
Let b be a Boolean expression. The notation $b_\perp$ represents the conditional

$$\texttt{SKIP} \lhd b \rhd \texttt{ABORT}$$

Law 3. If variable v does not appear in the expression e then
(1) $v := e \,;\, (v = e)_\perp = v := e$.
(2) $(v = e)_\perp \,;\, v := e \sqsubseteq \texttt{SKIP}$.

The command $\texttt{VAR}\, v$ introduces a new variable v, and the command $\texttt{END}\, v$ removes the variable v. Declaration and end of scope commands obey the following laws

Law 4.
(1) $\texttt{END}\, v \,;\, \texttt{VAR}\, v \sqsubseteq \texttt{SKIP} = \texttt{VAR}\, v \,;\, \texttt{END}\, v$.
(2) $v := e \,;\, \texttt{END}\, v = \texttt{END}\, v$.

The iteration $b * q$ is defined as the least fixed point of the equation

$$X = (q \,;\, X) \lhd b \rhd \texttt{SKIP}$$

and satisfies the following law

Law 5. $b * q \,;\, (b \vee c) * q = (b \vee c) * q$.

3. Specification of Machine Instructions

A correct compiler ensures that the execution of the machine code has the same (or better) behaviour than that ascribed to the source code. In order to pursue a rigorous reasoning for the correctness of a compiler, we decide to define the target code in a subset of the source language whose semantics are already known. This allows us to manipulate the machine code and the source program in the same mathematical framework. The definition of the machine language starts with a simple set of the components of the machine state, and each instruction is then identified by a fragment of code describing how the machine state is updated by the execution of the instruction. This paper considers a machine with just six components.

- m : *rom* $\rightarrow$ *word* is the store occupied by the machine code.
- M : *ram* $\rightarrow$ *word* is the store used for variables where the word-length is unspecified.
- P : *rom* is the pointer to the current instruction.
- A, B, C : *word* are the general-purpose registers.

Here *word* is the set of machine word values, *rom* is the set of read-only memory addresses, and *ram* is the (disjoint) set of read-write memory addresses.

We introduce a set of machine instructions below, each of which is defined by a fragment of code operating on the machine state.

$$\begin{aligned}
\textbf{store}(n) &\stackrel{def}{=} M[n], P := A, P + 1 \\
\textbf{load}(n) &\stackrel{def}{=} A, B, C, P := M[n], A, B, P + 1 \\
\textbf{loadc}(n) &\stackrel{def}{=} A, B, C, P := n, A, B, P + 1
\end{aligned}$$

$$\begin{aligned}
\textbf{jump}(k) &\stackrel{def}{=} P := P + k + 1 \\
\textbf{condj}(k) &\stackrel{def}{=} P := P + 1 \lhd A \rhd P := P + k + 1 \\
\textbf{swap}(A, B) &\stackrel{def}{=} A, B, P := B, A, P + 1 \\
\textbf{swap}(A, C) &\stackrel{def}{=} A, C, P := C, A, P + 1 \\
\textbf{add} &\stackrel{def}{=} A, P := A + B, P + 1
\end{aligned}$$

In the following sections we will use "**store**(n)" (for example) to stand for the text of instruction **store**(n).

The behaviour of a machine program stored in $m[s], \ldots, m[f-1]$ can be specified by

$$\begin{aligned}
\mathscr{I}\, s\, f\, m \stackrel{def}{=}\ & \mathtt{VAR}\, A, B, C, P ; P := s ; \\
& (P < f) * \mathit{mstep} ; \\
& (P = f)_{\perp} ; \mathtt{END}\, A, B, C, P
\end{aligned}$$

where *mstep* is an interpreter for a single machine instruction stored in $m[P]$. The program $(P = f)_{\perp}$ ensures that if the execution of the interpreter terminates, it will end at the finish address f.

4. A Provably Correct Compiling Specification

The compiling specification is defined as a predicate $\mathscr{CP}\, q\, s\, f\, m\, \Psi\, \Omega\, \Phi\, \grave{\Phi}$ relating a process q and the machine code stored in $m[s], \ldots, m[f-1]$ where

- The symbol table Ψ maps each global variable of q to its address in the memory M.
- Ω is the set of *free locations* in M which can be used to store the temporary results during the evaluation of expressions; i.e., we assume that $\textbf{range}(\Psi) \cap \Omega = \emptyset$.
- The register tables Φ and $\grave{\Phi}$ are used to map each register to the expression whose value is being held by it before and after the execution of the machine code $m[s], \ldots, m[f-1]$ respectively. For example, $\Phi A[M[\Psi x]/x, \ldots, M[\Psi z]/z]$ is the value of the register A before the execution of the machine program. In order to specify an uninitialized register, we will use $\perp$ to stand for the expression whose value is unspecified. Algebraically, the expression $\perp$ can be formalized by the following law:

  $$\mathtt{VAR}\, x = \mathtt{VAR}\, x ; x := \perp$$

  We define a binary relation $\preceq$ among register tables by

  $$\Phi_1 \preceq \Phi_2 \stackrel{def}{=} \forall R \bullet \Phi_1(R) \neq \perp \Rightarrow (\Phi_1(R) = \Phi_2(R))$$

  Clearly $\preceq$ is a *partial order*. The notation $\Phi_1 \sqcap \Phi_2$ is used to stand for the *greatest lower bound* of register tables Φ_1 and Φ_2.

It is the responsibility of the compiler to ensure that execution of the target code should have the same (or better) behaviour than that ascribed to the source code. This leads to the following definition of the compiling specification predicate $\mathscr{CP}$:

$$
\begin{aligned}
&\mathscr{CP}\, q\, s\, f\, m\, \Psi\, \Omega\, \Phi, \dot{\Phi} \stackrel{def}{=} \\
&\quad [\![\Psi_\Omega]\!](q) \sqsubseteq \\
&\quad \mathtt{VAR}\, P, A, B, C\ ; \\
&\quad P, A, B, C := s, [\![\Psi_\Omega]\!](\Phi A), [\![\Psi_\Omega]\!](\Phi B), [\![\Psi_\Omega]\!](\Phi C)\ ; \\
&\quad (P < f) * mstep\ ; \\
&\quad (P = f \wedge A = [\![\Psi_\Omega]\!](\dot{\Phi} A) \wedge B = [\![\Psi_\Omega]\!](\dot{\Phi} B) \wedge C = [\![\Psi_\Omega]\!](\dot{\Phi} C))_\perp\ ; \\
&\quad \mathtt{END}\, P, A, B, C
\end{aligned}
$$

where the notation $[\![\Psi_\Omega]\!](q)$ was defined in [HoH92] as the *weakest specification* of the correct implementation of q with respect to the symbol table Ψ and the free workspace Ω

$$
\begin{aligned}
[\![\Psi_\Omega]\!](q) &\stackrel{def}{=} \Psi_\Omega\ ;\ q\ ;\ \Psi_\Omega^{-1} \\
\Psi_\Omega &\stackrel{def}{=} \mathtt{VAR}\, x, \ldots, z\ ; \\
&\quad\ x, \ldots, z := M[\Psi x], \ldots, M[\Psi z]; \\
&\quad\ \mathtt{END}\, M\, [\mathbf{range}(\Psi) \uplus \Omega] \\
\Psi_\Omega^{-1} &\stackrel{def}{=} \mathtt{VAR}\, M\, [\mathbf{range}(\Psi) \uplus \Omega]\ ; \\
&\quad\ M[\Psi x], \ldots, M[\Psi z] := x, \ldots, z\ ; \\
&\quad\ \mathtt{END}\, x, \ldots, z
\end{aligned}
$$

where $\{x, \ldots, z\}$ contains all the program variables in the domain of Ψ, and $M[S]$ is an array variable with the index set S. Note that $\uplus$ stands for disjoint union. For any expression e we define

$$[\![\Psi_\Omega]\!](e) \stackrel{def}{=} e[M[\Psi x]/x, \ldots, M[\Psi z]/z]$$

Ψ_Ω and $[\![\Psi_\Omega]\!]$ are fully investigated in [Hoa91, HoH92]. Here we only present those properties of $[\![\Psi_\Omega]\!]$ which will be used in the later proof.

Lemma

(1) $[\![\Psi_\Omega]\!](\mathtt{SKIP}) \sqsubseteq \mathtt{SKIP}$
(2) $[\![\Psi_\Omega]\!](q\ ;\ r)) \sqsubseteq [\![\Psi_\Omega]\!](q)\ ;\ [\![\Psi_\Omega]\!](r)$
(3) $[\![\Psi_\Omega]\!](v := e) \sqsubseteq M[\Psi v] := [\![\Psi_\Omega]\!](e)$
(4) $[\![\Psi_\Omega]\!](q \lhd b \rhd r) \sqsubseteq [\![\Psi_\Omega]\!](q) \lhd [\![\Psi_\Omega]\!](b) \rhd [\![\Psi_\Omega]\!](r)$
(5) $[\![\Psi_\Omega]\!](b * q) \sqsubseteq [\![\Psi_\Omega]\!](b) * [\![\Psi_\Omega]\!](q)$

A predicate $\mathscr{CE}\, e\, s\, f\, m\, \Psi\, \Omega\, \Phi\, \dot{\Phi}$ is provided to relate an expression e to its machine code. $\mathscr{CE}$ is *correct* if the register A will hold the value of e after the execution of the machine code, and the memory used to store the values of program variables will remain unchanged.

$$\mathscr{CE}\, e\, s\, f\, m\, \Psi\, \Omega\, \Phi\, \dot{\Phi} \stackrel{def}{=} (\dot{\Phi} A = e) \wedge \mathscr{CP}\, \mathtt{SKIP}\, s\, f\, m\, \Psi\, \Omega\, \Phi\, \dot{\Phi}$$

For Booleans, we introduce a predicate

$$\mathscr{CBE}\, b\, s\, f\, m\, \Psi\, \Omega\, \Phi\, \dot{\Phi}\, \{true \mapsto tt, false \mapsto ff\}$$

which is correct if the execution of the machine code will terminate at the exit address *tl* when the value of *b* is true, or otherwise at the address *fl* when *b* is false.

$$
\begin{aligned}
&\mathscr{CBE}\, b\, s, f\, m\, \Psi\, \Omega\, \Phi, \dot{\Phi} \stackrel{def}{=} \\
&\quad [\![\Psi_\Omega]\!](\mathtt{SKIP}) \sqsubseteq \\
&\quad \mathtt{VAR}\, P, A, B, C\ ; \\
&\quad P, A, B, C := s, [\![\Psi_\Omega]\!](\Phi A), [\![\Psi_\Omega]\!](\Phi B), [\![\Psi_\Omega]\!](\Phi C)\ ; \\
&\quad (P < f) * mstep\ ; \\
&\quad (P = (tl \lhd [\![\Psi_\Omega]\!](b) \rhd fl) \wedge A = [\![\Psi_\Omega]\!](\dot{\Phi} A) \wedge \\
&\quad\ B = [\![\Psi_\Omega]\!](\dot{\Phi} B) \wedge C = [\![\Psi_\Omega]\!](\dot{\Phi} C))_\perp\ ; \\
&\quad \mathtt{END}\, P, A, B, C
\end{aligned}
$$

4.1. Theorems of Process Compilation

This section presents the theorems of the compiling specification predicates $\mathscr{CP}$, $\mathscr{CE}$ and $\mathscr{CBE}$.

Program Compilation

SKIP compiles to an empty sequence of instructions.

(1) $\mathscr{CP}\, \mathtt{SKIP}\, s\, s\, m\, \Psi\, \Omega\, \Phi\, \Phi$

Sequential composition may be compiled by concatenating the resulting machine code in memory.

(2) $\mathscr{CP}\, (q\ ;\ r)\, s\, f\, m\, \Psi\, \Omega\, \Phi\, \dot{\Phi}$ **if**
$\exists j, \Phi_1 \bullet s \leq j \leq f \wedge \mathscr{CP}\, q\, s\, j\, m\, \Psi\, \Omega\, \Phi\, \Phi_1 \wedge \mathscr{CP}\, r\, j\, f\, m\, \Psi\, \Omega\, \Phi_1\, \dot{\Phi}$

Assignment is compiled by the following four theorems. $\dot{\Phi}$ depends on whether the registers hold values that depend on the assigned variable v. This information is recorded by case analysis below. Vars is a function that returns the set of variables used in an expression and $\oplus$ stands for functional overriding.

(3a) $\mathscr{CP}\, (v := e)\, s\, f\, m\, \Psi\, \Omega\, \Phi\, \dot{\Phi}$ **if**
$\exists \Phi_1 \bullet \mathscr{CE}\, e\, s\, (f-1)\, m\, \Psi\, \Omega\, \Phi\, \Phi_1 \wedge m[f-1] = \mathbf{store}(\Psi v) \wedge$
$v \notin \mathtt{Vars}(\Phi_1 B) \wedge v \notin \mathtt{Vars}(\Phi_1 C) \wedge \dot{\Phi} = \Phi_1 \oplus \{A \mapsto v\}$

(3b) $\mathscr{CP}\, (v := e)\, s\, f\, m\, \Psi\, \Omega\, \Phi\, \dot{\Phi}$ **if**
$\exists \Phi_1 \bullet \mathscr{CE}\, e\, s\, (f-1)\, m\, \Psi\, \Omega\, \Phi\, \Phi_1 \wedge m[f-1] = \mathbf{store}(\Psi v) \wedge$
$v \in \mathtt{Vars}(\Phi_1 B) \wedge v \notin \mathtt{Vars}(\Phi_1 C) \wedge \dot{\Phi} = \Phi_1 \oplus \{A \mapsto v, B \mapsto \perp\}$

(3c) $\mathscr{CP}\, (v := e)\, s\, f\, m\, \Psi\, \Omega\, \Phi\, \dot{\Phi}$ **if**
$\exists \Phi_1 \bullet \mathscr{CE}\, e\, s\, (f-1)\, m\, \Psi\, \Omega\, \Phi\, \Phi_1 \wedge m[f-1] = \mathbf{store}(\Psi v) \wedge$
$v \notin \mathtt{Vars}(\Phi_1 B) \wedge v \in \mathtt{Vars}(\Phi_1 C) \wedge \dot{\Phi} = \Phi_1 \oplus \{A \mapsto v, C \mapsto \perp\}$

(3d) $\mathscr{CP}\, (v := e)\, s\, f\, m\, \Psi\, \Omega\, \Phi\, \dot{\Phi}$ **if**
$\exists \Phi_1 \bullet \mathscr{CE}\, e\, s\, (f-1)\, m\, \Psi\, \Omega\, \Phi\, \Phi_1 \wedge m[f-1] = \mathbf{store}(\Psi v) \wedge$

$$v \in \mathtt{Vars}(\Phi_1 B) \wedge v \in \mathtt{Vars}(\Phi_1 C) \wedge$$
$$\dot{\Phi} = \Phi_1 \oplus \{A \mapsto v, B \mapsto \bot, C \mapsto \bot\}$$

For the conditional construct the value of $\dot{\Phi}$ depends on the greatest lower bound of the values given by the two subprograms q and r since either may be executed.

(4) $\mathscr{CP}(q \lhd b \rhd r)\, s\, f\, m\, \Psi\, \Omega\, \Phi\, \dot{\Phi}$ **if**
$\exists tl, fl, \Phi_1, \Phi_2, \Phi_3 \bullet s \leq tl \leq fl \leq f \wedge$
$\mathscr{CBE}\, b\, s\, tl\, m\, \Psi\, \Omega\, \Phi\, \Phi_1\, \{true \mapsto tl, false \mapsto fl\} \wedge$
$\mathscr{CP}\, q\, tl\, (fl - 1)\, m\, \Psi\, \Omega\, \Phi_1\, \Phi_2 \wedge$
$m[fl - 1] = \mathbf{jump}(f - fl) \wedge$
$\mathscr{CP}\, r\, fl\, f\, m\, \Psi\, \Omega\, \Phi_1\, \Phi_3 \wedge$
$\dot{\Phi} = \Phi_2 \sqcap \Phi_3$

For the iteration construct, the the final value of Φ when b and q are compiled is the same as the starting value since q may or may not be executed depending on the value of b.

(5) $\mathscr{CP}(b * q)\, s\, f\, m\, \Psi\, \Omega\, \Phi\, \dot{\Phi}$ **if**
$\exists tl \bullet s \leq tl \leq f \wedge$
$\mathscr{CBE}\, b\, s\, tl\, m\, \Psi\, \Omega\, \Phi\, \dot{\Phi}\, \{true \mapsto tl, false \mapsto f\} \wedge$
$\mathscr{CP}\, q\, tl\, (f - 1)\, m\, \Psi\, \Omega\, \dot{\Phi}\, \Phi \wedge$
$m[f - 1] = \mathbf{jump}(s - f)$

A weaker value for $\dot{\Phi}$ is always allowed if a stronger one is possible when compiling a program (e.g., when compiling the body q of the iteration construct above).

(6) $\mathscr{CP}\, q\, s\, f\, m\, \Psi\, \Omega\, \Phi\, \dot{\Phi}$ **if**
$\exists \Phi_1 \bullet \dot{\Phi} \preceq \Phi_1 \wedge \mathscr{CP}\, q\, s\, f\, m\, \Psi\, \Omega\, \Phi\, \Phi_1$

Expression Compilation

Below are a few selected theorems for the compilation of integer expressions with an addition operator.

If an expression is already held in the A register, then no object code is necessary.

(7a) $\mathscr{CE}\, e\, s\, s\, m\, \Psi\, \Omega\, \Phi\, \Phi$ **if**
$e = \Phi A$

If an expression is held in the B register, then it is simply necessary to move this to the A register, using the **swap** instruction. The values in the registers, recorded by $\dot{\Phi}$ must be adjusted accordingly.

(7b) $\mathscr{CE}\, e\, s\, f\, m\, \Psi\, \Omega\, \Phi\, \dot{\Phi}$ **if**
$e = \Phi B \wedge$
$m[s] = \mathbf{swap}(A, B) \wedge$
$f = s + 1 \wedge$
$\dot{\Phi} = \Phi \oplus \{A \mapsto \Phi B, B \mapsto \Phi A\}$

If a variable in an expression is not already held in one of the registers, it must be pushed onto the register stack from memory.

(8a) $\mathscr{CE}\, x\, s\, f\, m\, \Psi\, \Omega\, \Phi\, \dot{\Phi}$ **if**
$x \notin \mathbf{range}(\Phi) \wedge$
$m[s] = \mathbf{load}(\Psi x) \wedge$
$f = s + 1 \wedge$
$\dot{\Phi} = \{A \mapsto x, B \mapsto \Phi A, C \mapsto \Phi B\}$

Similarly, a constant integer value that is not in one of the registers must also be pushed onto the register stack.

(8b) $\mathscr{CE}\, n\, s\, f\, m\, \Psi\, \Omega\, \Phi\, \dot{\Phi}$ **if**
$n \notin \mathbf{range}(\Phi) \wedge$
$m[s] = \mathbf{loadc}(n) \wedge$
$f = s + 1 \wedge$
$\dot{\Phi} = \{A \mapsto n, B \mapsto \Phi A, C \mapsto \Phi B\}$

If two expressions to be added are already in registers A and B, only the **add** instruction needs to be generated.

(9a) $\mathscr{CE}\, (e_1 + e_2)\, s\, f\, m\, \Psi\, \Omega\, \Phi\, \dot{\Phi}$ **if**
$e_1 + e_2 \notin \mathbf{range}(\Phi) \wedge \{e_1, e_2\} = \{\Phi A, \Phi B\} \wedge$
$m[s] = \mathbf{add} \wedge$
$f = s + 1 \wedge$
$\dot{\Phi} = \Phi \oplus \{A \mapsto e_1 + e_2\}$

If two expressions to be added are in registers A and C, then the value in the C register must be moved to the B register first, using the **swap** instruction.

(9b) $\mathscr{CE}\, (e_1 + e_2)\, s\, f\, m\, \Psi\, \Omega\, \Phi\, \dot{\Phi}$ **if**
$e_1 + e_2 \notin \mathbf{range}(\Phi) \wedge \{e_1, e_2\} = \{\Phi A, \Phi C\} \wedge$
$m[s] = \mathbf{swap}(B, C) \wedge m[s+1] = \mathbf{add} \wedge$
$f = s + 2 \wedge$
$\dot{\Phi} = \{A \mapsto e_1 + e_2, B \mapsto \Phi C, C \mapsto \Phi B\}$

If one of the expressions in an addition is available in a register, then it may be saved in a temporary location while the other expression is evaluated.

(10a) $\mathscr{CE}\, (e_1 + e_2)\, s\, f\, m\, \Psi\, (\{loc\} \uplus \Omega)\, \Phi\, \dot{\Phi}$ **if**
$e_1 + e_2 \notin \mathbf{range}(\Phi) \wedge e_2 \notin \mathbf{range}(\Phi) \wedge e_1 = \Phi A \wedge$
$\exists j, \Phi_1, \Phi_2 \bullet s < j \leq f \wedge m[s] = \mathbf{store}(loc) \wedge$
$\mathscr{CE}\, e_2\, (s+1)\, j\, m\, \Psi\, \Omega\, \Phi\, \Phi_1 \wedge m[j] = \mathbf{load}(loc) \wedge$
$\mathscr{CE}\, (e_1 + e_2)\, (j+1)\, f\, m\, \Psi\, (\{loc\} \uplus \Omega)\, \Phi_2\, \dot{\Phi} \wedge$
$\Phi_2 = \{A \mapsto e_1, B \mapsto \Phi_1 A, C \mapsto \Phi_1 B\}$

(10b) $\mathscr{CE}\, (e_1 + e_2)\, s\, f\, m\, \Psi\, (\{loc\} \uplus \Omega)\, \Phi\, \dot{\Phi}$ **if**
$e_1 + e_2 \notin \mathbf{range}(\Phi) \wedge e_2 \notin \mathbf{range}(\Phi) \wedge e_1 = \Phi B \wedge$
$\exists j, \Phi_1, \Phi_2, \Phi_3 \bullet (s+2) \leq j \leq f \wedge$
$m[s] = \mathbf{swap}(A, B) \wedge m[s+1] = \mathbf{store}(loc) \wedge$
$\mathscr{CE}\, e_2\, (s+2)\, j\, m\, \Psi\, \Omega\, \Phi_1\, \Phi_2 \wedge$
$\Phi_1 = \Phi \oplus \{A \mapsto \Phi B, B \mapsto \Phi A\} \wedge$
$m[j] = \mathbf{load}(loc) \wedge$
$\mathscr{CE}\, (e_1 + e_2)\, (j+1)\, f\, m\, \Psi\, (\{loc\} \uplus \Omega)\, \Phi_3\, \dot{\Phi} \wedge$
$\Phi_3 = \{A \mapsto e_1, B \mapsto \Phi_2 A, C \mapsto \Phi_2 B\}$

If none of the expressions in an addition are available in any of the registers, then it must be compiled from scratch.

(11) $\mathscr{CE}(e_1 + e_2)\,s\,f\,m\,\Psi\,\Omega\,\Phi\,\dot{\Phi}$ **if**
$e_1 + e_2 \notin \mathbf{range}(\Phi) \wedge e_1 \notin \mathbf{range}(\Phi) \wedge e_2 \notin \mathbf{range}(\Phi) \wedge$
$\exists j, \Phi_1 \bullet s \leq j \leq f \wedge$
$\mathscr{CE}\,e_1\,s\,j\,m\,\Psi\,\Omega\,\Phi\,\Phi_1 \wedge$
$\mathscr{CE}(e_1 + e_2)\,j\,f\,m\,\Psi\,\Omega\,\Phi_1\,\dot{\Phi}$

Boolean Expression Compilation

Compilation of Boolean expressions involves an extra parameter that describes the locations to which a jump should be made in the event of a *true* or *false* evaluation of the expression.

For `true` and `false` constants, the jump is predetermined.

(12) $\mathscr{CBE}$ `true` $s\,f\,m\,\Psi\,\Omega\,\Phi\,\Phi\,\{true \mapsto tl, false \mapsto fl\}$ **if**
$m[s] = \mathbf{jump}(tl - f) \wedge$
$f = s + 1 \wedge tl \neq s \wedge fl \neq s$

(13) $\mathscr{CBE}$ `false` $s\,f\,m\,\Psi\,\Omega\,\Phi\,\Phi\,\{true \mapsto tl, false \mapsto fl\}$ **if**
$m[s] = \mathbf{jump}(fl - f) \wedge$
$f = s + 1 \wedge tl \neq s \wedge fl \neq s$

A variable must be pushed onto the register stack and tested.

(14) $\mathscr{CBE}\,x\,s\,f\,m\,\Psi\,\Omega\,\Phi\,\dot{\Phi}\,\{true \mapsto tl, false \mapsto fl\}$ **if**
$m[s] = \mathbf{load}(\Phi x) \wedge m[s+1] = \mathbf{condj}(fl - (s+2)) \wedge$
$m[s+2] = \mathbf{jump}(tl - (s+3)) \wedge$
$f = s + 3 \wedge tl \notin [s, f) \wedge fl \notin [s, f) \wedge$
$\dot{\Phi} = \{A \mapsto x, B \mapsto \Phi A, C \mapsto \Phi B\}$

The standard Boolean connectives are handled as follows:

(15) $\mathscr{CBE}(b \vee c)\,s\,f\,m\,\Psi\,\Omega\,\Phi\,\dot{\Phi}\,\{true \mapsto tl, false \mapsto fl\}$ **if**
$\exists j, \Phi_1, \Phi_2 \bullet s \leq j \leq f \wedge$
$\mathscr{CBE}\,b\,s\,j\,m\,\Omega\,\Phi\,\Phi_1\,\{true \mapsto tl, false \mapsto j\} \wedge$
$\mathscr{CBE}\,c\,j\,f\,m\,\Omega\,\Phi_1\,\Phi_2\,\{true \mapsto tl, false \mapsto fl\} \wedge$
$\dot{\Phi} = \Phi_1 \sqcap \Phi_2 \wedge$
$tl \notin [s, f) \wedge fl \notin [s, f)$

(16) $\mathscr{CBE}(b \wedge c)\,s\,f\,m\,\Psi\,\Omega\,\Phi\,\dot{\Phi}\,\{true \mapsto tl, false \mapsto fl\}$ **if**
$\exists j, \Phi_1, \Phi_2 \bullet s \leq j \leq f \wedge$
$\mathscr{CBE}\,b\,s\,j\,m\,\Omega\,\Phi\,\Phi_1\,\{true \mapsto j, false \mapsto fl\} \wedge$
$\mathscr{CBE}\,c\,j\,f\,m\,\Omega\,\Phi_1\,\Phi_2\,\{true \mapsto tl, false \mapsto fl\} \wedge$
$\dot{\Phi} = \Phi_1 \sqcap \Phi_2 \wedge$
$tl \notin [s, f) \wedge fl \notin [s, f)$

(17) $\mathscr{CBE}(\neg b)\,s\,f\,m\,\Psi\,\Omega\,\Phi\,\dot{\Phi}\,\{true \mapsto tl, false \mapsto fl\}$ **if**
$\mathscr{CBE}\,b\,s\,f\,m\,\Psi\,\Omega\,\Phi\,\dot{\Phi}\,\{true \mapsto fl, false \mapsto tl\}$

4.2. Verification of Compiling Specification

This section presents a proof of one of the theorems to demonstrate the style of proof used. Algebraic laws are used to gradually transform the program by a

series of refinement steps. In general, we aim for the proofs to be less than a page in length to make them understandable and readable by humans. Long proofs are seldom read and are likely to contain errors, especially if done by hand.

Proof of Theorem 3a

$$
\begin{array}{ll}
 & \mathbf{VAR}\, P, A, B, C \,;\, P, A, B, C := s, [\![\Psi_\Omega]\!](\Phi A), [\![\Psi_\Omega]\!](\Phi B), [\![\Psi_\Omega]\!](\Phi C)\,; \\
 & (P < f) * \mathit{mstep}\,; \\
 & (P = f \wedge A = [\![\Psi_\Omega]\!](\dot{\Phi} A) \wedge B = [\![\Psi_\Omega]\!](\dot{\Phi} B) \wedge C = [\![\Psi_\Omega]\!](\dot{\Phi} C))_\perp\,; \\
 & \mathbf{END}\, P, A, B, C \\
\sqsupseteq & \{\textit{laws } 3(1), 4(1) \textit{ and } 5\} \\
 & \mathbf{VAR}\, P, A, B, C \,;\, P, A, B, C := s, [\![\Psi_\Omega]\!](\Phi A), [\![\Psi_\Omega]\!](\Phi B), [\![\Psi_\Omega]\!](\Phi C)\,; \\
 & (P < (f-1)) * \mathit{mstep}\,; \\
 & (P = (f-1) \wedge A = [\![\Psi_\Omega]\!](\Phi_1 A) \wedge B = [\![\Psi_\Omega]\!](\Phi_1 B) \wedge C = [\![\Psi_\Omega]\!](\Phi_1 C))_\perp\,; \\
 & \mathbf{END}\, P, A, B, C\,; \\
 & \mathbf{VAR}\, P, A, B, C\,; \\
 & P, A, B, C := (f-1), [\![\Psi_\Omega]\!](\Phi_1 A), [\![\Psi_\Omega]\!](\Phi_1 B), [\![\Psi_\Omega]\!](\Phi_1 C)\,; \\
 & (P < f) * \mathit{mstep}\,; \\
 & (P = f \wedge A = [\![\Psi_\Omega]\!](\dot{\Phi} A) \wedge B = [\![\Psi_\Omega]\!](\dot{\Phi} B) \wedge C = [\![\Psi_\Omega]\!](\dot{\Phi} C))_\perp\,; \\
 & \mathbf{END}\, P, A, B, C \\
\sqsupseteq & \{\textit{assumption, def of } \mathscr{C}\mathscr{E}\} \\
 & [\![\Psi_\Omega]\!](\mathbf{SKIP})\,; \\
 & \mathbf{VAR}\, P, A, B, C\,; \\
 & P, A, B, C := (f-1), [\![\Psi_\Omega]\!](e), [\![\Psi_\Omega]\!](\Phi_1 B), [\![\Psi_\Omega]\!](\Phi_1 C)\,; \\
 & (P < f) * \mathit{mstep}\,; \\
 & (P = f \wedge A = [\![\Psi_\Omega]\!](\dot{\Phi} A) \wedge B = [\![\Psi_\Omega]\!](\dot{\Phi} B) \wedge C = [\![\Psi_\Omega]\!](\dot{\Phi} C))_\perp\,; \\
 & \mathbf{END}\, P, A, B, C \\
= & \{\textit{assumption, def of } \mathbf{store} \textit{ and } \dot{\Phi}\} \\
 & [\![\Psi_\Omega]\!](\mathbf{SKIP})\,; \\
 & \mathbf{VAR}\, P, A, B, C\,; \\
 & P, A, B, C, M[\Psi v] := f, [\![\Psi_\Omega]\!](e), [\![\Psi_\Omega]\!](\Phi_1 B), [\![\Psi_\Omega]\!](\Phi_1 C), [\![\Psi_\Omega]\!](e); \\
 & (P = f \wedge A = M[\Psi v] \wedge B = [\![\Psi_\Omega]\!](\Phi_1 B) \wedge C = [\![\Psi_\Omega]\!](\Phi_1 C))_\perp\,; \\
 & \mathbf{END}\, P, A, B, C \\
= & \{\textit{laws } 3(1) \textit{ and } 4(1)\} \\
 & [\![\Psi_\Omega]\!](\mathbf{SKIP})\,;\, M[\Psi v] := [\![\Psi_\Omega]\!](e) \\
\sqsupseteq & \{\textit{lemma } (2)\} \\
 & [\![\Psi_\Omega]\!](\mathbf{SKIP}\,;\, v := e) \\
= & \{\textit{law } 1(1)\} \\
 & [\![\Psi_\Omega]\!](v := e)
\end{array}
$$

5. A Prototype Compiler

All the compiling theorems are in the form of Horn clauses. Thus it is relatively easy to code the specification as a logic program. The practical difficulties that arise are ensuring termination and coding of the constraints. The former may be attacked by taking into account which parameters are used as inputs and which are outputs, and reordering conjoined predicates for efficient execution with this knowledge in mind. Constraints may also be encoded with knowledge of which parameters will be instantiated before use. This allows *negation by failure* to be used. For a fuller discussion of the issues involved, see [Bow92]. In this case, it is assumed that the source program (q), start address (s), symbol table (Ψ), free locations (Ω) and initial register table (Φ) are instantiated, and the finish address (f), object code (m) and final register table ($\dot{\Phi}$) are to be generated.

The example encodings of theorems in this section use "standard" pure Prolog [ClM87]. Prolog's logical basis makes the encoding a relatively mechanical

process. Infix and other operators are defined to aid readability, particularly of source programs. The approach assumes the source language is input in abstract syntax form, and a parser may be required to preprocess the concrete syntax in practice [BoB92]. Constraints are encoded in curly brackets {...}. Since Prolog, unlike most high-level languages, does not evaluate expressions in parameters, it is sometimes necessary to recode such parameters as new variables and add extra constraints on these variables.

Program Compilation

(1) Skip

```
cp(skip,S,S,M,Psi,Omega,Phi,Phi).
```

(2) Sequential composition

```
cp(Q;R,S,F,M,Psi,Omega,Phi,Phi_) :-
  cp(Q,S,J,M,Psi,Omega,Phi,Phi1),
  cp(R,J,F,M,Psi,Omega,Phi1,Phi_),
  {S=<J=<F}.
```

(3a) Assignment

```
cp(V:=E,S,F,M,Psi,Omega,Phi,Phi_) :-
  ce(E,S,L,M,Psi,Omega,Phi,Phi1), {F=L+1},
  {M@L = store(Psi@V)},
  {V notin vars(Phi1@b)}, {V notin vars(Phi1@c)},
  {Phi_ = Phi1 <+> [a->V]}.
```

(4) Conditional

```
cp(Q<B>R,S,F,M,Psi,Omega,Phi,Phi_) :-
  cbe(B,S,Tl,M,Psi,Omega,Phi,Phi1,[true->Tl,false->Fl]),
  cp(Q,Tl,L,M,Psi,Omega,Phi1,Phi2), {Fl=L+1},
  {M@L = jump(F-Fl)},
  cp(R,Fl,F,M,Psi,Omega,Phi1,Phi3),
  {Phi_ = Phi2^Phi3},
  {S<Tl<Fl=<F}.
```

(5) Iteration

```
cp(B*Q,S,F,M,Psi,Omega,Phi,Phi_) :-
  cbe(B,S,Tl,M,Psi,Omega,Phi,Phi_,[true->Tl,false->F]),
  cp(Q,Tl,L,M,Psi,Omega,Phi_,Phi), {F=L+1},
  {M@L = jump(S-F)},
  {S<Tl<F}.
```

Expression Compilation

The expression compilation clauses are straightforward although numerous. For example:

(7a)

```
ce(E,S,S,M,Psi,Omega,Phi,Phi) :-
  {E=Phi@a}.
```

(8a)

```
ce(X,S,F,M,Psi,Omega,Phi,Phi_) :-
  {X notin range(Phi)},
  {M@S = load(Psi@X)},
  {F=S+1},
  {Phi_ = [a->X,b->Phi@a,c->Phi@b]}.
```

(9a)

```
ce(E1+E2,S,F,M,Psi,Omega,Phi,Phi_) :-
  {E1+E2 notin range(Phi)}, {{E1,E2}={Phi@a,Phi@b}},
  {M@S = add}, {F=S+1},
  {Phi_ = Phi <+> [a->E1+E2]}.
```

(10a)

```
ce(E1+E2,S,F,M,Psi,[Loc|Omega],Phi,Phi_) :-
  {E1+E2 notin range(Phi)}, {E2 notin range(Phi)}, {E1=Phi@a},
  {M@S = store(Loc)},
  ce(E2,S+1,J,M,Psi,Omega,Phi,Phi1),
  {M@J = load(Loc)},
  {Phi2 = [a->E1,b->Phi1@a,c->Phi1@b]},
  ce(E1+E2,J+1,F,M,Psi,[Loc|Omega],Phi2,Phi_),
  {S+1=<J<F}.
```

(11)

```
ce(E1+E2,S,F,M,Psi,Omega,Phi,Phi_) :-
  {E1+E2 notin range(Phi)},
  {E1 notin range(Phi)}, {E2 notin range(Phi)},
  ce(E1,S,J,M,Psi,Omega,Phi,Phi1),
  ce(E1+E2,J,F,M,Psi,Omega,Phi1,Phi_),
  {S=<J=<F}.
```

Boolean Expression Compilation

Note that it the case of forward jumps, the location of the destination address is not necessarily known at the time of execution. This can be alleviated by relaxing some of the constraints at the time of execution. In practice this does not matter since the "calling" clauses ensure that the constraints are met anyway.

(12)

```
cbe(true,S,F,M,Psi,Omega,Phi,Phi_,[true->Tl,false->Fl]) :-
  {F=S+1}, {M@S = jump(Tl-F)},
  {Tl notin rng(S,F)}, {Fl notin rng(S,F)}.
```

(14)

```
cbe(X,S,F,M,Psi,Omega,Phi,Phi_,[true->Tl,false->Fl]) :-
  {M@S = load(Psi@X)},
  {S2=S+2}, {M@(S+1) = condj(Fl-S2)},
  {F=S+3}, {M@S2 = jump(Tl-F)},
  {Tl notin rng(S,F)}, {Fl notin rng(S,F)},
  {Phi_ = [a->X,b->Phi@a,c->Phi@b]}.
```

(15)

```
cbe(B or C,S,F,M,Psi,Omega,Phi,Phi_,[true->Tl,false->Fl]) :-
  cbe(B,S,J,M,Psi,Omega,Phi,Phi1,[true->Tl,false->J]),
  cbe(C,J,F,M,Psi,Omega,Phi1,Phi2,[true->Tl,false->Fl]),
  {Phi_ = Phi1^Phi2},
  {Tl notin rng(S,F)}, {Fl notin rng(S,F)}.
```

6. Conclusion

An example of an optimizing compiling specification and matching prototype compiler have been presented together with a technique for proving the compiling specification correct. This has extended previous work by recording the contents

of registers known at compile-time, and using this information to optimize the code generated. It would be possible to extend this technique to cover the contents of program variables as well if desired by supplementing the information recorded in Φ and $\grave{\Phi}$. Additionally, to reduce the number of parameters to the compiling relation, it may be beneficial to merge s with Φ and f with $\grave{\Phi}$ since the former represents information concerned with the precondition and the latter is concerned with the postcondition when the programming constructs are executed. For example, we could make $s = [\![\Psi_\Omega]\!](\Phi P)$ and $f = [\![\Psi_\Omega]\!](\grave{\Phi} P)$.

One issue is to ensure that the theorems are complete in the sense that all valid constructs can be compiled to (at least one) object code. In the case of multiple theorems for different optimizations of the same construct, this can be ensured by checking that the constraining predicates in all the relevant theorems for a particular construct reduce to true when combined (using disjunction). If this is not the case then it is possible for the compiler not to produce object code in certain (valid) cases that have not been covered.

More than one theorem may apply in the compilation of a particular construct and several (possibly an infinite number of) object code sequences may be valid. In this case, the prototype compiler will (attempt to) return all the possibilities. A *real* compiler will of course select one of these sequences. This code selection process is potentially exponential in complexity and an important aspect of an actual compiler is choosing an optimized code sequence efficiently [Gie92]. In the example Prolog prototype compiler presented here, code may be "selected" by ordering the clauses appropriately with the more efficient or preferable clauses placed first.

In standard Prolog, *functors* (in particular, lists) must be used to encode sets, etc., needed by the constraints in the compiling theorems. The extra clauses required to complete the program and implement the constraints (not included in the paper) consist of about two pages of program code. Thus, it would be tractable to formally prove the prototype compiler implements the specification for a given set of inputs, assuming a suitable semantics of (a subset of) Prolog [Llo87], if this is of concern (e.g., see [BSW90]). In addition, optimization using transformation of logic programs [ClL92] would be possible. However this has not (yet) been attempted by the authors, since the prototype has simply been used as a means of quickly animating the specification mechanically.

Proofs of termination and non-violation of the omitted occurs-check in Prolog [KPS93] and the compilation of the Prolog itself [Rus92] are possible. Obviously it would be even more interesting to prove a real (optimizing) compiler correct, but this is still beyond the capability of current proof technology. Attempts have been made to prove a simple compiler correct, but even this is highly intractable [BBF92].

Constraint logic programming [Coh90] is now well established and several implementations are available. Such systems could allow an even more direct encoding from the theorems, avoiding the need for some of the explicit encodings of constraints needed in standard Prolog. This could also allow the prototype to be used in more modes, and perhaps even as a *decompiler* [BoB93, BrB92]. A simple decompiler in Prolog, based on a specification similar to the style presented, here has already been produced [Bow93a].

Compilation into other paradigms, such as via a *normal form* [HHS93] and directly into a *netlist* of hardware components [HPB93], are likely to provide new and interesting optimization challenges for the future.

Acknowledgements

Prof. Tony Hoare originated the style of compiling specification and verification presented here. Tony Hoare, Burghard von Karger and Augusto Sampaio provided helpful comments on an earlier draft.

References

[ASU86] Aho, A. V., Sethi, R. and Ullman, J. D.: *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Series in Computer Science, 1986.

[Bjø92] Bjørner, D.: Trusted Computing Systems: The ProCoS Experience. *Proc. ICSE '14*, North-Holland, Melbourne, Australia, 11–14 May 1992.

[Bow92] Bowen, J. P.: From Programs to Object Code using Logic and Logic Programming. In [GiG92], pp. 173–192.

[Bow93a] Bowen, J. P.: From Programs to Object Code and back again using Logic Programming: Compilation and Decompilation. *Journal of Software Maintenance: Research and Practice*, **5**, 205–234 (December 1993).

[Bow93b] Bowen, J. P. *et al.*: A ProCoS II Project Description: ESPRIT Basic Research Project 7071. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, **50**, 128–137 (June 1993).

[Bow94] Bowen, J. P. (ed.): *Towards Verified Systems.* Elsevier, Real-Time Safety-Critical Systems series, 1994.

[BoB92] Bowen, J. P. and Breuer, P. T.: Occam's Razor: The Cutting Edge of Parser Technology. *Proc. TOULOUSE'92: Fifth International Conference on Software Engineering and its Applications*, Toulouse, France, 7–11 December 1992.

[BoB93] Bowen, J. P. and Breuer, P. T.: Decompilation. In van Zuylen, H. (ed.), *The REDO Compendium: Reverse Engineering for Software Maintenance*, chapter 10, John Wiley & Sons, pp. 131–138, 1993.

[BFO93] Bowen, J. P., Fränzle, M., Olderog, E.-R. and Ravn, A.P.: Developing Correct Systems. *Proc. 5th Euromicro Workshop on Real-Time Systems*, IEEE Computer Society Press, pp. 176–187, 1993.

[BHP90] Bowen, J. P., He Jifeng and Pandya, P. K.: An Approach to Verifiable Compiling Specification and Prototyping. In Deransart, P. and Małuszyński, J. (eds.), *Programming Language Implementation and Logic Programming*, Springer-Verlag, LNCS **456**, pp. 45–59, 1990.

[BoS93] Bowen, J. P. and Stavridou, V.: Safety-Critical Systems, Formal Methods and Standards. *IEE/BCS Software Engineering Journal*, **8**(4), 189–209 (July 1993).

[BrB92] Breuer, P. T. and Bowen, J. P.: 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.

[BSW90] Bundy, A., Smaill, A. and Wiggins G.: The Synthesis of Logic Programs from Inductive Proofs. In Lloyd, J. W. (ed.) *Computational Logic.* Springer-Verlag, Basic research series, pp. 135–149, 1990.

[Bun82] Bunimova, E. O.: *A Method of Language Mappings Describing.* Doctoral dissertation, Moscow University, Russia, 1982. (In Russian.)

[BBF92] Buth, B., Buth, K.-H., Fränzle, M., von Karger, B., Lakhneche, Y., Langmaack, H. and Müller-Olm, M.: Provably Correct Compiler Implementation. In Karstens, U. and Pfahler, P. (eds.), *Compiler Construction*, Springer-Verlag, LNCS **641**, pp. 141–155, 1992.

[ClL92] Clement, T. P. and Lau, K.-K.: *Logic Program Synthesis and Transformation.* Springer-Verlag, Workshops in Computing, 1992.

[ClM87] Clocksin, W. F., and Mellish, C. S.: *Programming in Prolog.* 3rd edition, Springer-Verlag, 1987.

[Coh90] Cohen, J.: Constraint Logic Programming Languages. *Communications of the ACM*, **33**(7), 52–68 (1990).

[Cur93] Curzon, P.: Deriving Correctness Properties of Compiled Code. *Formal Methods in System Design*, **3**, 83–115 (1993).

[Gie92] Giegerich, R.: Considerate Code Selection. In [GiG92], pp. 51–65.

[GiG92] Giegerich, R. and Graham, S. L. (eds.): *Code Generation – Concepts, Tools, Techniques.* Springer-Verlag, Workshops in Computing, 1992.

[HeB92] He Jifeng and Bowen, J. P.: Time Interval Semantics and Implementation of a Real-Time Programming Language. *Proc. Fourth Euromicro Workshop on Real-Time Systems*, IEEE Computer Society Press, pp. 110–115, 1992.

[HPB93] He Jifeng, Page, I. and Bowen, J. P.: Towards a provably correct hardware implementation of Occam. In Milne, G. J. and Pierre, L. (eds.) *Correct Hardware Design and Verification Methods*, Springer-Verlag, LNCS **683**, pp. 214–225, 1993.

[Hoa91] Hoare, C. A. R.: 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, pp. 33–48, 1991.

[HoH92] Hoare, C. A. R. and He Jifeng: Refinement Algebra Proves Correctness of a Compiler. In Broy, M. (ed.), *Programming and Mathematical Method*, Springer-Verlag, NATO ASI Series F: Computer and Systems Sciences, vol. 88, pp. 245–269, 1992.

[HHB90] Hoare, C. A. R., He Jifeng, Bowen, J. P. and Pandya, P. K.: An Algebraic Approach to Verifiable Compiling Specification and Prototyping of the ProCoS Level 0 Programming Language. *ESPRIT '90 Conference Proceedings*, Kluwer Academic Publishers, pp. 804–818, 1990.

[HHS93] Hoare C. A. R., He Jifeng and Sampaio, A.: Normal Form Approach to Compiler Design. *Acta Informatica*, **30**, 701–739 (1993).

[KPS93] Krishna Rao, M. R. K., Pandya, P. K. and Shyamasundar, R. K.: Verification Tools in the Development of Provably Correct Compilers. In Woodcock, J. C. P. and Larsen, P. G. (eds.), *FME '93: Industrial-Stength Formal Methods*, Springer-Verlag, LNCS **670**, pp. 442–461, 1993.

[Lev92] Levin, V.: Algebraically Provable Specification of Optimized Compilations. In Bjørner, D., Broy, M. and Pottosin, I.V. (eds.), *Formal Methods in Programming and Their Applications*, Springer-Verlag, LNCS **735**, 1993.

[Llo87] Lloyd, J. W.: *Foundations of Logic Programming*. 2nd edition, Springer-Verlag, 1987.

[Rus92] Russinoff, D. M.: A Verified Prolog Compiler for the Warren Abstract Machine. *Journal of Logic Programming*, **13**, 367–412 (1992).

[Ste93] Stepney, S.: *High Integrity Compilation: A Case Study*. Prentice Hall, 1993.

[SWC91] Stepney, S., Whitley, D., Cooper, D. and Grant, C.: A Demonstrably Correct Compiler. *Formal Aspects of Computing*, **3**(1), 58–101 (January–March 1991).

Received November 1992
Accepted in revised form November 1993 by C.B. Jones