Extensible proof-producing compilation

Magnus O. Myreen[1], Konrad Slind[2], and Michael J. C. Gordon[1]

[1] Computer Laboratory, University of Cambridge, Cambridge, UK
[2] School of Computing, University of Utah, Salt Lake City, USA

Abstract. This paper presents a compiler which produces machine code from functions defined in the logic of a theorem prover, and at the same time proves that the generated code executes the source functions. Unlike previously published work on proof-producing compilation from a theorem prover, our compiler provides broad support for user-defined extensions, targets multiple carefully modelled commercial machine languages, and does not require termination proofs for input functions. As a case study, the compiler is used to construct verified interpreters for a small LISP-like language. The compiler has been implemented in the HOL4 theorem prover.

1 Introduction

Compilers pose a problem for program verification: if a high-level program is proved correct, then the compiler's transformation must be trusted in order for the proof to carry over to a guarantee about the generated executable code. In practice there is also another problem: most source languages (C, Java, Haskell etc.) do not have a formal semantics, and it is therefore hard to formally state and verify properties of programs written in these languages.

This paper explores an approach to compilation aimed at supporting program verification. We describe a compiler which takes as input functions expressed in the logic of a theorem prover, compiles the functions to machine code (ARM, x86 or PowerPC) and also proves that the generated code executes the supplied functions. For example, given function f as input

$$f(r_1) = \mathsf{if}\ \ r_1 < 10\ \ \mathsf{then}\ \ r_1\ \ \mathsf{else}\ \ \mathsf{let}\ r_1 = r_1 - 10\ \mathsf{in}\ f(r_1)$$

the compiler can generate ARM machine code

```
E351000A      L:  cmp r1,#10
2241100A          subcs r1,r1,#10
2AFFFFFC          bcs L
```

and automatically prove a theorem which certifies that the generated code executes f. The following theorem states, if register one (r1) initially holds value r_1, then the code will leave register one holding value $f(r_1)$. The theorem is expressed as a machine-code Hoare triple [17] where the separating conjunction '$*$' can informally be read as 'and'.

$$\{\mathsf{r1}\ r_1 * \mathsf{pc}\ p * \mathsf{s}\}\ \ p : \texttt{E351000A}, \texttt{2241100A}, \texttt{2AFFFFFC}\ \ \{\mathsf{r1}\ f(r_1) * \mathsf{pc}\ (p{+}12) * \mathsf{s}\}$$

The fact that f is expressed as a function in the native language of a theorem prover means that it has a precise semantics and that one can prove properties about f, e.g. one can prove that $f(x) = x \mod 10$ (here mod is modulus over unsigned machine words). Properties proved for f carry over to guarantees about the generated machine code via the certificate proved by the compiler. For example, one can rewrite the theorem from above to state that the ARM code calculates $r_1 \mod 10$:

$$\{\mathsf{r1}\ r_1 * \mathsf{pc}\ p * \mathsf{s}\}\ \ p : \texttt{E351000A}, \texttt{2241100A}, \texttt{2AFFFFFC}\ \ \{\mathsf{r1}\ (r_1 \bmod 10) * \mathsf{pc}\ (p{+}12) * \mathsf{s}\}$$

Proof-producing compilation from a theorem prover has been explored before by many, as will be discussed in Section 6. The contributions that distinguish the work presented here are that the compiler:

1. targets multiple, carefully modelled, commercial machine languages (namely ARM, PowerPC and x86, as modelled by Fox [7], Leroy [11] and Sarkar [6]);
2. does not require the user to prove termination of the input functions (a restriction posed by the theorem prover in similar work by Li et al. [12–14]);
3. can, without any added complexity to the certification proof, handle a range of optimising transformations (Section 4); and
4. supports significant user-defined extensions to its input language (Section 3.1); extensions which made it possible to compile interpreters for a small LISP-like language as a case study (Section 5).

The compiler[3] uses a functional input, which is meant to either be extended directly by the user, as discussed in Section 3.1, or used as a back-end in compilers with more general input languages, e.g. [8, 13, 14].

This paper builds on the authors' work on *post hoc* verification of realistically modelled machine code [16–18], and certifying compilation [8, 12–14].

2 Core functionality

The compiler presented in this paper accepts tail-recursive functions as input, functions defined as recursive equations '$f(\ldots) = \ldots$' in a format described in Section 2.1. As output the compiler produces machine code together with a correctness certificate, a theorem which states that the generated machine code executes the function given as input.

The overall compilation algorithm can be broken down into three stages:

1. code generation: generates, without proof, machine code from input f;
2. decompilation: derives, via proof, a function f' describing the machine code;
3. certification: proves $f = f'$.

The remaining subsections describe the input language and code generation that make proving $f = f'$ feasible, as well as the mechanism by which f' is derived. Section 3 describes extensions to the core algorithm.

[3] The HOL4 source is at http://hol.sf.net/ under HOL/examples/machine-code.

2.1 Input language

The compiler's input language consists of let-expressions, if-statements and tail-recursion. The language restricts variable names to correspond to names of registers or stack locations.

The following grammar describes the input language. Let r range over register names, r_0, r_1, r_2, etc., and s over stack locations, s_1, s_2, s_3 etc., m over memory modelling functions (mappings from aligned 32-bit machine words to 32-bit machine words), f over function names, g over names of already compiled functions, and i_5, i_7, i_8 and i_{32} over unsigned words of size 5-, 7-, 8- and 32-bits, respectively. Bit-operators &, ??, !!, $\ll$, $\gg$ are and, xor, or, left-shift, right-shift. Operators suffixed with '.' are signed-versions of those without the suffix.

$$
\begin{aligned}
\mathit{input} &::= f(v, v, ..., v) = \mathit{rhs} \\
\\
\mathit{rhs} &::= \mathsf{let}\ r = \mathit{exp}\ \mathsf{in}\ \mathit{rhs} \\
&\mid\ \mathsf{let}\ s = r\ \mathsf{in}\ \mathit{rhs} \\
&\mid\ \mathsf{let}\ m = m[\,\mathit{address} \mapsto r\,]\ \mathsf{in}\ \mathit{rhs} \\
&\mid\ \mathsf{let}\ (v, v, ..., v) = g(v, v, ..., v)\ \mathsf{in}\ \mathit{rhs} \\
&\mid\ \mathsf{if}\ \mathit{guard}\ \mathsf{then}\ \mathit{rhs}\ \mathsf{else}\ \mathit{rhs} \\
&\mid\ f(v, v, ..., v) \\
&\mid\ (v, v, ..., v) \\
\\
\mathit{exp} &::= x \mid \neg\, x \mid s \mid i_{32} \mid x\ \mathit{binop}\ x \mid m\ \mathit{address} \mid x \ll i_5 \mid x \gg i_5 \mid x \gg. i_5 \\
\mathit{binop} &::= + \mid - \mid \times \mid \mathsf{div} \mid \& \mid ?? \mid !! \\
\mathit{cmp} &::= < \mid \leq \mid > \mid \geq \mid <. \mid \leq. \mid >. \mid \geq. \mid = \\
\mathit{guard} &::= \neg\, \mathit{guard} \mid \mathit{guard} \wedge \mathit{guard} \mid \mathit{guard} \vee \mathit{guard} \mid x\ \mathit{cmp}\ x \mid x\ \&\ x = 0 \\
\mathit{address} &::= r \mid r + i_7 \mid r - i_7 \\
\\
x &::= r \mid i_8 \\
v &::= r \mid s \mid m
\end{aligned}
$$

This input language was designed to be machine independent; programs constructed from this grammar can be compiled to any of the target languages: ARM, x86 and PowerPC. However the input language differs for each target in the number of registers available ($r_0 ... r_{12}$ for ARM, $r_0 ... r_6$ for x86 and $r_0 ... r_{31}$ for PowerPC) and some detailed restrictions on the use of $\times$ and div.

2.2 Code generation

The input language was designed to mimic the operations of machine instructions in order to ease code generation. Each let-expression usually produces a single instruction, e.g.

let $r_3 = r_3 + r_2$ in	generates ARM code	`add r3,r3,r2`
let $r_3 = r_3 + r_2$ in	generates x86 code	`add ebx,edx`
let $r_3 = r_3 + r_2$ in	generates PowerPC code	`add 3,3,2`

In some cases one let-expression is split into a few instructions, e.g.

let $r_3 = r_0 - r_2$ in	generates x86 code	`mov ebx,eax` `sub ebx,edx`

let $r_3 = 5000$ in	generates ARM code	`mov r3,#19`
		`mov r3,r3,lsl 8`
		`add r3,r3,#136`

The code generator was programmed to use a few assembly tricks, e.g. on x86 certain instances of addition, which would normally require two instructions (`mov` followed by `add`), can be implemented as a single load-effective-address `lea`:

let $r_3 = r_0 + r_2$ in generates x86 code `lea ebx,[eax+edx]`

A combination of compare and branch are used to implement if-statements, e.g.

if $r_3 = 45$ then ... else ...	generates ARM code	`cmp r3,#45` `bne L1`

Function returns and function calls generate branch instructions.

The compiler generates a list of assembly instructions, which is translated into machine code using off-the-shelf assemblers: Netwide Assembler `nasm` [1] for x86 and the GNU Assembler `gas` [2] for ARM and PowerPC. Note that these tools do not need to be trusted. If incorrect code is generated then the certification phase, which is to prove the correctness certificate, will fail.

2.3 Proving correctness theorem

The theorem certifying the correctness of the generated machine code is proved by first deriving a function f' describing the effect of the generated code, and then proving that f' is equal to the original function to be compiled. Function f' is derived using proof-producing *decompilation* [18]. This section will illustrate how decompilation is used for compilation and then explain decompilation.

Example. Given function f, which traverses r_0 steps down a linked-list in m,

$$
\begin{array}{l}
f(r_0, r_1, m) = \\
\quad \textsf{if } r_0 = 0 \textsf{ then } (r_0, r_1, m) \textsf{ else} \\
\quad\quad \textsf{let } r_1 = m(r_1) \textsf{ in} \\
\quad\quad \textsf{let } r_0 = r_0 - 1 \textsf{ in} \\
\quad\quad\quad f(r_0, r_1, m)
\end{array}
$$

Code generation produces the following x86 code.

```
0:  85C0        L1:  test eax, eax
2:  7405             jz L2
4:  8B09             mov ecx,[ecx]
6:  48               dec eax
7:  EBF7             jmp L1
                L2:
```

Proof-producing decompilation is applied to the generated machine code. The decompiler takes machine code as input and produces a function f' as output,

$$
\begin{array}{l}
f'(eax, ecx, m) = \\
\quad \mathsf{if}\ eax\ \&\ eax = 0\ \mathsf{then}\ (eax, ecx, m)\ \mathsf{else} \\
\quad\quad \mathsf{let}\ ecx = m(ecx)\ \mathsf{in} \\
\quad\quad \mathsf{let}\ eax = eax - 1\ \mathsf{in} \\
\quad\quad\quad f'(eax, ecx, m)
\end{array}
$$

together with a theorem (expressed as a machine-code Hoare triple [17, 18]) which states that f' accurately records the update executed by the machine code. The decompiler derives f' via proof with respect to a detailed processor model written by Sarkar [6]. Here eip asserts the value of the program counter.

$$
\begin{array}{l}
f'_{pre}(eax, ecx, m) \Rightarrow \\
\{\ (\mathsf{eax}, \mathsf{ecx}, \mathsf{m})\ \mathsf{is}\ (eax, ecx, m) * \mathsf{eip}\ p * \mathsf{s}\ \} \\
\ p : \texttt{85C074058B0948EBF7} \\
\{\ (\mathsf{eax}, \mathsf{ecx}, \mathsf{m})\ \mathsf{is}\ f'(eax, ecx, m) * \mathsf{eip}\ (p{+}9) * \mathsf{s}\ \}
\end{array}
$$

The decompiler also automatically defines f'_{pre}, which is a boolean-valued function that keeps track of necessary conditions for the Hoare triple to be valid as well as side-conditions that are needed to avoid raising hardware exceptions. In this case, ecx is required to be part of the memory segment modelled by function m and the underlying model requires ecx to be word-aligned ($ecx \,\&\, 3 = 0$), whenever $eax \,\&\, eax \neq 0$.

$$
\begin{array}{l}
f'_{pre}(eax, ecx, m) = \\
\quad \mathsf{if}\ eax\ \&\ eax = 0\ \mathsf{then}\ true\ \mathsf{else} \\
\quad\quad f'_{pre}(eax{-}1, m(ecx), m)\ \wedge\ ecx \in \mathsf{domain}\ m\ \wedge\ (ecx \,\&\, 3 = 0)
\end{array}
$$

Next the compiler proves $f = f'$. Both f and f' are recursive functions; thus proving $f = f'$ would normally require an induction. The compiler can avoid an induction since both f and f' are defined as instances of $\mathsf{tailrec}$:

$$\mathsf{tailrec}\ x = \mathsf{if}\ (G\ x)\ \mathsf{then}\ \mathsf{tailrec}\ (F\ x)\ \mathsf{else}\ (D\ x)$$

The compiler proves $f = f'$ by showing that the components of the $\mathsf{tailrec}$ instantiation are equal, i.e. for f and f', as given above, the compiler only needs to prove the following. (f'_{pre} is not needed for these proofs.)

$$
\begin{array}{rrcl}
G: & (\lambda(r_0, r_1, m).\ r_0 \neq 0) & = & (\lambda(eax, ecx, m).\ eax\ \&\ eax \neq 0) \\
D: & (\lambda(r_0, r_1, m).\ (r_0, r_1, m)) & = & (\lambda(eax, ecx, m).\ (eax, ecx, m)) \\
F: & (\lambda(r_0, r_1, m).\ (r_0{-}1, m(r_1), m)) & = & (\lambda(eax, ecx, m).\ (eax{-}1, m(ecx), m))
\end{array}
$$

The code generation phase is programmed in such a way that the above component proofs will always be proved by an expansion of let-expressions followed by rewriting with a handful of verified rewrite rules that undo assembly tricks, e.g. $\forall w.\ w\ \&\ w = w$.

The precondition f'_{pre} is not translated, instead f_{pre} is defined to be f'_{pre}. The compiler proves the certificate of correctness by rewriting the output from the decompiler using theorems $f' = f$ and $f'_{pre} = f_{pre}$. The example results in:

$$
\begin{array}{l}
f_{pre}(eax, ecx, m) \Rightarrow \\
\{\, (\mathsf{eax}, \mathsf{ecx}, \mathsf{m})\ \mathsf{is}\ (eax, ecx, m) * \mathsf{eip}\ p * \mathsf{s}\,\} \\
\ p : \texttt{85C074058B0948EBF7} \\
\{\, (\mathsf{eax}, \mathsf{ecx}, \mathsf{m})\ \mathsf{is}\ f(eax, ecx, m) * \mathsf{eip}\ (p{+}9) * \mathsf{s}\,\}
\end{array}
$$

Decompilation. The proof-producing decompilation, which was used above, is explained in detail in [18]. However, a brief outline will be given here.

Decompilation starts by composing together Hoare triples for machine instructions to produce Hoare triples describing one pass through the code. For the above x86 code, successive compositions collapse Hoare triples of the individual instructions into two triples, one for the case when the conditional branch is taken and one for the case when it is not.

$$
\begin{array}{l}
eax \,\&\, eax = 0 \Rightarrow \\
\{\, (\mathsf{eax}, \mathsf{ecx}, \mathsf{m})\ \mathsf{is}\ (eax, ecx, m) * \mathsf{eip}\ p * \mathsf{s}\,\} \\
\ p : \texttt{85C074058B0948EBF7} \\
\{\, (\mathsf{eax}, \mathsf{ecx}, \mathsf{m})\ \mathsf{is}\ (eax, ecx, m) * \mathsf{eip}\ (p{+}9) * \mathsf{s}\,\}
\end{array}
$$

$$
\begin{array}{l}
eax \,\&\, eax \neq 0 \ \wedge\ ecx \in \mathsf{domain}\ m \ \wedge\ (ecx \,\&\, 3 = 0) \Rightarrow \\
\{\, (\mathsf{eax}, \mathsf{ecx}, \mathsf{m})\ \mathsf{is}\ (eax, ecx, m) * \mathsf{eip}\ p * \mathsf{s}\,\} \\
\ p : \texttt{85C074058B0948EBF7} \\
\{\, (\mathsf{eax}, \mathsf{ecx}, \mathsf{m})\ \mathsf{is}\ (eax{-}1, m(ecx), m) * \mathsf{eip}\ p * \mathsf{s}\,\}
\end{array}
$$

Using these one-pass theorems, the decompiler instantiates the following loop rule to produce function f' and the certificate theorem. If F describes a looping pass, and D is a pass that exits the loop, then $\mathsf{tailrec}\ x$ is the result of the loop:

$$
\begin{array}{rl}
\forall \mathsf{res}\ \mathsf{res'}\ c. & (\forall x.\ P\ x \wedge G\ x \Rightarrow \{\mathsf{res}\ x\}\, c\, \{\mathsf{res}\ (F\ x)\}) \wedge \\
& (\forall x.\ P\ x \wedge \neg(G\ x) \Rightarrow \{\mathsf{res}\ x\}\, c\, \{\mathsf{res'}\ (D\ x)\}) \Rightarrow \\
& (\forall x.\ \mathsf{pre}\ x \Rightarrow \{\mathsf{res}\ x\}\, c\, \{\mathsf{res'}\ (\mathsf{tailrec}\ x)\})
\end{array}
$$

Here pre is the recursive function which records the side-conditions that need to be met (e.g. in this case P is used to record that ecx needs to be aligned).

$$\mathsf{pre}\ x = P\ x \wedge (G\ x \Rightarrow \mathsf{pre}\ (F\ x))$$

For the above one-pass Hoare triples to fit the loop rule, the decompiler instantiates G, F, D, P, res and $\mathsf{res'}$ as follows:

$$
\begin{array}{rcl}
G & = & \lambda(eax, ecx, m).\ (eax \,\&\, eax \neq 0) \\
F & = & \lambda(eax, ecx, m).\ (eax{-}1, m(ecx), m) \\
D & = & \lambda(eax, ecx, m).\ (eax, ecx, m) \\
P & = & \lambda(eax, ecx, m).\ (eax \,\&\, eax \neq 0) \Rightarrow ecx \in \mathsf{domain}\ m \wedge (ecx \,\&\, 3 = 0) \\
\mathsf{res} & = & \lambda(eax, ecx, m).\ (\mathsf{eax}, \mathsf{ecx}, \mathsf{m})\ \mathsf{is}\ (eax, ecx, m) * \mathsf{eip}\ p * \mathsf{s} \\
\mathsf{res'} & = & \lambda(eax, ecx, m).\ (\mathsf{eax}, \mathsf{ecx}, \mathsf{m})\ \mathsf{is}\ (eax, ecx, m) * \mathsf{eip}\ (p{+}9) * \mathsf{s}
\end{array}
$$

3 Extensions, stacks and subroutines

The examples above illustrated the algorithm of the compiler based on simple examples involving only registers and direct memory accesses. This section describes how the compiler supports user-defined extensions, stack operations and subroutine calls.

3.1 User-defined extensions

The compiler has a restrictive input language. User-defined extensions to this input language are thus vital in order to be able to make use of the features specific to each target language.

User-defined extensions to the input language are made possible by the proof method which derives a function f' describing the effect of the generated code: function f' is constructed by composing together Hoare triples describing parts of the generated code. By default, automatically derived Hoare triples for each individual machine instruction are used. However, the user can instead supply the proof method with alternative Hoare triples in order to build on previously proved theorems.

An example will illustrate how this observation works in practice. Given the following Hoare triple (proved in Section 1) which shows that ARM code has been shown to implement "r_1 is assigned $r_1 \bmod 10$",

$$\{\mathsf{r1}\ r_1 * \mathsf{pc}\ p * \mathsf{s}\}\ \ p : \texttt{E351000A}, \texttt{2241100A}, \texttt{2AFFFFFC}\ \ \{\mathsf{r1}\ (r_1 \bmod 10) * \mathsf{pc}\ (p{+}12) * \mathsf{s}\}$$

the code generator expands its input language for ARM with the following line:

$$rhs\ ::=\ \mathsf{let}\ r_1 = r_1 \bmod 10\ \mathsf{in}\ rhs$$

Now when a function f is to be compiled which uses this feature,

$$\begin{aligned} f(r_1, r_2, r_3) = {} & \mathsf{let}\ r_1 = r_1 + r_2\ \mathsf{in} \\ & \mathsf{let}\ r_1 = r_1 + r_3\ \mathsf{in} \\ & \mathsf{let}\ r_1 = r_1\ \mathsf{mod}\ 10\ \mathsf{in} \\ & \quad r_1 \end{aligned}$$

the code generator implements "$\mathsf{let}\ r_1 = r_1\ \mathsf{mod}\ 10\ \mathsf{in}$" using the machine code (underlined below) found inside the Hoare triple. The other instructions are `E0811002` for `add r1,r1,r2` and `E0811003` for `add r1,r1,r3`.

`E0811002 E0811003` <u>`E351000A 2241100A 2AFFFFFC`</u>

The compiler would now normally derive f' by composing Hoare triples for the individual machine instructions, but in this case the compiler considers the underlined code as a 'single instruction' whose effect is described by the supplied Hoare triple. It composes the following Hoare triples, in order to derive a Hoare triple for the entire code.

$$\{\mathsf{r1}\ r_1 * \mathsf{r2}\ r_2 * \mathsf{pc}\ p\}\ \ p : \texttt{E0811002}\ \ \{\mathsf{r1}\ (r_1{+}r_2) * \mathsf{r2}\ r_2 * \mathsf{pc}\ (p{+}4)\}$$

$$\{\mathsf{r1}\ r_1 * \mathsf{r3}\ r_3 * \mathsf{pc}\ p\}\ \ p : \texttt{E0811003}\ \ \{\mathsf{r1}\ (r_1{+}r_3) * \mathsf{r3}\ r_3 * \mathsf{pc}\ (p{+}4)\}$$

$$\{\mathsf{r1}\ r_1 * \mathsf{pc}\ p * \mathsf{s}\}\ \ p : \texttt{E351000A}, \texttt{2241100A}, \texttt{2AFFFFFC}\ \ \{\mathsf{r1}\ (r_1 \bmod 10) * \mathsf{pc}\ (p{+}12) * \mathsf{s}\}$$

The resulting f' is trivially equal to f and thus the resulting Hoare triple states that the generated code actually executes f.

$$\begin{array}{l}\{\mathsf{r1}\ r_1 * \mathsf{r2}\ r_2 * \mathsf{r3}\ r_3 * \mathsf{pc}\ p * \mathsf{s}\}\\ p : \texttt{E0811002}, \texttt{E0811003}, \texttt{E351000A}, \texttt{2241100A}, \texttt{2AFFFFFC}\\ \{\mathsf{r1}\ f(r_1, r_2, r_3) * \mathsf{r2}\ r_2 * \mathsf{r3}\ r_3 * \mathsf{pc}\ (p{+}20) * \mathsf{s}\}\end{array}$$

It is important to note that the Hoare triples supplied to the compiler need not concern registers or memory locations, instead more abstract Hoare triples can be supplied. For example, in Section 5, the compiler is given Hoare triples that show how basic operations over LISP s-expressions can be performed. The LISP operation car is implemented by ARM instruction `E5933000`. Here s-expressions are defined as a data-type with type-constructors Dot (pairs), Num (numbers) and Sym (symbols). Details are given in Section 5.

$$\begin{array}{l}(\exists x\ y.\ v_1 = \mathsf{Dot}\ x\ y)\ \Rightarrow\\ \{\ \mathsf{lisp}\ (a, l)\ (v_1, v_2, v_3, v_4, v_5, v_6) * \mathsf{pc}\ p\ \}\\ p : \texttt{E5933000}\\ \{\ \mathsf{lisp}\ (a, l)\ (\mathsf{car}\ v_1, v_2, v_3, v_4, v_5, v_6) * \mathsf{pc}\ (p + 4)\ \}\end{array}$$

The above specification extends the ARM code generator to handle assignments of car v_1 to s-expression variable v_1.

$$rhs\ ::=\ \mathsf{let}\ v_1 = \mathsf{car}\ v_1\ \mathsf{in}\ rhs$$

3.2 Stack usage

The stack can be used by assignments to and from variables s_0, s_1, s_2 etc., e.g. the following let-expressions correspond to machine code which loads register 1 from stack location 3 (three down from top of stack), adds 78 to register 1 and then stores the result in stack location 2.

$$\begin{array}{rl}f(r_1, s_2, s_3) = & \mathsf{let}\ r_1 = s_3\ \mathsf{in}\\ & \mathsf{let}\ r_1 = r_1 + 78\ \mathsf{in}\\ & \mathsf{let}\ s_2 = r_1\ \mathsf{in}\\ & \quad (r_1, s_2, s_3)\end{array}$$

Internally stack accesses are implemented by supplying the decompiler with specifications which specify stack locations using M-assertions (defined formally in [17], informally $\mathsf{M}\,x\ y$ asserts that memory location x holds value y), e.g. the following is the specification used for reading the value of stack location 3 into register 1. Register 13 is the stack pointer.

$$\begin{array}{l}\{\mathsf{r1}\ r_1 * \mathsf{r13}\ sp * \mathsf{M}(sp{+}12)\ s_3 * \mathsf{pc}\ p\}\\ p : \texttt{E59D100C}\\ \{\mathsf{r1}\ s_3 * \mathsf{r13}\ sp * \mathsf{M}(sp{+}12)\ s_3 * \mathsf{pc}\ (p{+}4)\}\end{array}$$

The postcondition for the certification theorem proved for the above function f:

$$\{\ (\mathsf{r1}, \mathsf{M}(sp{+}8), \mathsf{M}(sp{+}12))\ \mathsf{is}\ f(r_1, s_2, s_3)\ *\ \mathsf{r13}\ sp\ *\ \mathsf{pc}\ (p{+}12)\ \}$$

3.3 Subroutines and procedures

Subroutines can be in-lined or called as procedures. Each compilation adds a new let-expression into the input languages of the compiler. The added let-expressions describe the compiled code, i.e. they allow subsequent compilations to use the previously compiled code. For example, when the following function (which uses f from above) is compiled, the code for f will be in-lined as in Section 3.1.

$$\begin{aligned} g(r_1, r_2, s_2, s_3) = \mathsf{let}\ (r_1, s_2, s_3) = f(r_1, s_2, s_3)\ \mathsf{in} \\ \mathsf{let}\ s_2 = r_1\ \mathsf{in} \\ (r_1, r_2, s_2, s_3) \end{aligned}$$

Note that for simplicity, function calls must match the variable names used when compiling the called function was compiled, e.g. a function compiled as '$k(r_1) = ...$' cannot be called as '$\mathsf{let}\ r_2 = k(r_2)\ \mathsf{in}$' since the input is passed to code implementing k in register 1 not in register 2.

If the compiler had been asked to compile f as a procedure, then the numbering of stack variables needs to be shifted for calls to f. Compiling f as a procedure sandwiches the code for f between a push and pop instruction that keep track of the procedure's return address. When f accesses stack locations 2 and 3 (counting in pop-order), these are for caller g locations 1 and 2.

$$\begin{aligned} g(r_1, r_2, s_1, s_2) = \mathsf{let}\ (r_1, s_1, s_2) = f(r_1, s_1, s_2)\ \mathsf{in} \\ \mathsf{let}\ s_2 = r_1\ \mathsf{in} \\ (r_1, r_2, s_1, s_2) \end{aligned}$$

4 Optimising transformations

Given a function f, the compiler generates code, which it decompiles to produce function f' describing the behaviour of the generated code. The code generation phase can perform any optimisations as long as the certification phase can eventually prove $f = f'$. In particular, certain instructions can be reordered or removed, and the code's control flow can use special features of the target language.

4.1 Instruction reordering

Instruction reordering is a standard optimisation applied in order to avoid unnecessary pipeline stalls. The compiler presented here supports instruction reordering as is illustrated by the following example. Given a function f which stores r_1 into stack location s_5, then loads r_2 from stack location s_6, and finally adds r_1 and r_2.

$$\begin{aligned} f(r_1, r_2, s_5, s_6) = \mathsf{let}\ s_5 = r_1\ \mathsf{in} \\ \mathsf{let}\ r_2 = s_6\ \mathsf{in} \\ \mathsf{let}\ r_1 = r_1 + r_2\ \mathsf{in} \\ (r_1, r_2, s_5, s_6) \end{aligned}$$

The code corresponding directly to f might cause a pipeline stall as the result of the load instruction ($\mathsf{let}\ r_2 = s_6\ \mathsf{in}$) may not be available on time for the add instruction ($\mathsf{let}\ r_1 = r_1 + r_2\ \mathsf{in}$). It is therefore beneficial to schedule the load instructions as early as possible; the generated code reduces the risk of a pipeline stall by placing the load instruction before the store instruction:

$$\begin{aligned} f'(r_1, r_2, s_5, s_6) = {} & \mathsf{let}\ r_2 = s_6\ \mathsf{in} \\ & \mathsf{let}\ s_5 = r_1\ \mathsf{in} \\ & \mathsf{let}\ r_1 = r_1 + r_2\ \mathsf{in} \\ & \quad (r_1, r_2, s_5, s_6) \end{aligned}$$

Valid reorderings of instructions are unnoticeable after expansion of let-expressions, thus the proof of $f = f'$ does not need to be smarter to handle this optimisation.

4.2 Removal of dead code

Live-variable analysis can be applied to the code in order to remove unused or *dead code*. In the following definition of f, the first let-expression is unnecessary.

$$\begin{aligned} f(r_1, r_2, s_5, s_6) = {} & \mathsf{let}\ r_1 = s_5\ \mathsf{in} \\ & \mathsf{let}\ r_2 = s_6\ \mathsf{in} \\ & \mathsf{let}\ r_1 = r_2 + 8\ \mathsf{in} \\ & \quad (r_1, r_2, s_5, s_6) \end{aligned}$$

The generated code ignores the first let-expression and produces a function f' which is, after expansion of let-expressions, identical to f.

4.3 Conditional execution

ARM machine code allows conditional execution of nearly all instructions in order to allow short forward jumps to be replaced by conditionally executed instructions (this reduces branch overhead). The compiler produces conditionally-executed instruction blocks where short forward jumps would otherwise have been generated. The functions decompiled from conditionally executed instructions are indistinguishable from those decompiled from code with normal jumps (as can be seen in the examples of Section 1 and 4.4).

x86 supports conditional assignment using the conditional-move instruction `cmov`. For x86, the compiler replaces jumps across register-register moves by conditional-move instructions.

4.4 Shared tails

The compiler's input language supports if-statements that split control, but does not provide direct means for joining control-flow. For example, consider

```
(if r1 = 0 then r2 := 23 else r2 := 56); r1 := 4
```

which can be defined either directly as function f with 'shared tails'

$$f(r_1, r_2) = \mathsf{if}\ r_1 = 0\ \mathsf{then\ let}\ r_2 = 23\ \mathsf{in\ let}\ r_1 = 4\ \mathsf{in}\ (r_1, r_2)\ \mathsf{else\ let}\ r_2 = 56\ \mathsf{in\ let}\ r_1 = 4\ \mathsf{in}\ (r_1, r_2)$$

or as function g with auxiliary function g_2 compiled to be in-lined:

$$g(r_1, r_2) = \mathsf{let}\ (r_1, r_2) = g_2(r_1, r_2)\ \mathsf{in\ let}\ r_1 = 4\ \mathsf{in}\ (r_1, r_2)$$

$$g_2(r_1, r_2) = \mathsf{if}\ r_1 = 0\ \mathsf{then\ let}\ r_2 = 23\ \mathsf{in}\ (r_1, r_2)\ \mathsf{else\ let}\ r_2 = 56\ \mathsf{in}\ (r_1, r_2)$$

Generating code naively for f would result in two instructions for $\mathsf{let}\ r_1 = 4\ \mathsf{in}$, one for each branch. The compiler implements an optimisation which detects 'shared tails' so that the code for f will be identical to that produced for g. The compiler generates the following ARM code for function g (using conditional execution to avoid inserting short jumps).

```
 0:  E3510000          cmp r1,#0
 4:  03A02017          moveq r2,#23
 8:  13A02038          movne r2,#56
12:  E3A01004          mov r1,#4
```

5 Compilation example: verified LISP interpreter

The following example shows how one can utilise extensions to the input language. A verified interpreter for a LISP-like language is constructed using compilation. Details of the following section will be published as a separate paper.

The LISP interpreter constructed here operates over a simple date-type of s-expressions: $\mathsf{Dot}\ x\ y$ is a pair, $\mathsf{Num}\ n$ is a number n, and $\mathsf{Sym}\ s$ is a symbol s, in HOL4, s has type string. Basic operations are defined as follows:

$$\begin{aligned}
\mathsf{car}\ (\mathsf{Dot}\ x\ y) &= x \\
\mathsf{cdr}\ (\mathsf{Dot}\ x\ y) &= y \\
\mathsf{cons}\ x\ y &= \mathsf{Dot}\ x\ y \\
\\
\mathsf{plus}\ (\mathsf{Num}\ m)\ (\mathsf{Num}\ n) &= \mathsf{Num}\ (m + n) \\
\mathsf{minus}\ (\mathsf{Num}\ m)\ (\mathsf{Num}\ n) &= \mathsf{Num}\ (m - n) \\
\\
\mathsf{size}\ (\mathsf{Num}\ w) &= 0 \\
\mathsf{size}\ (\mathsf{Sym}\ s) &= 0 \\
\mathsf{size}\ (\mathsf{Dot}\ x\ y) &= 1 + \mathsf{size}\ x + \mathsf{size}\ y \\
&\ldots
\end{aligned}$$

A new resource assertion lisp is defined which relates LISP objects to concrete memory representations: lisp (a, l) $(v_1, v_2, v_3, v_4, v_5, v_6)$ states that a heap is located at address a, has capacity l, and that s-expressions v_1, v_2, v_3, v_4, v_5, v_6 are stored in this heap. The definition of lisp is omitted in this presentation.

Machine code for basic operations has been proved (in various ways using decompilation and compilation) to implement basic assertions, e.g. ARM code for storing car v_1 into v_1:

$$
\begin{array}{l}
(\exists x\ y.\ v_1 = \mathsf{Dot}\ x\ y)\ \Rightarrow \\
\{\ \mathsf{lisp}\ (a, l)\ (v_1, v_2, v_3, v_4, v_5, v_6) * \mathsf{pc}\ p\ \} \\
\ p : \texttt{E5933000} \\
\{\ \mathsf{lisp}\ (a, l)\ ((\mathsf{car}\ v_1), v_2, v_3, v_4, v_5, v_6) * \mathsf{pc}\ (p + 4)\ \}
\end{array}
$$

A memory allocator with a built-in copying garbage collector (a Cheney garbage collector [4]) is used to implement creation of a new pair Dot v_1 v_2. The precondition of this operation requires the heap to have enough space to accommodate a new cons-cell.

$$
\begin{array}{l}
(\mathsf{size}\ v_1 + \mathsf{size}\ v_2 + \mathsf{size}\ v_3 + \mathsf{size}\ v_4 + \mathsf{size}\ v_5 + \mathsf{size}\ v_6) < l\ \Rightarrow \\
\{\ \mathsf{lisp}\ (a, l)\ (v_1, v_2, v_3, v_4, v_5, v_6) * \mathsf{s} * \mathsf{pc}\ p\ \} \\
\ p : \text{... the allocator code ...} \\
\{\ \mathsf{lisp}\ (a, l)\ ((\mathsf{cons}\ v_1\ v_2), v_2, v_3, v_4, v_5, v_6) * \mathsf{s} * \mathsf{pc}\ (p + 328)\ \}
\end{array}
$$

When the above specifications are supplied to the compiler it knows what machine code to generate for two new commands: one for calculating car of v_1 and one for storing cons v_1 v_2 into v_1:

$$
\mathsf{let}\ v_1 = \mathsf{car}\ v_1\ \mathsf{in} \qquad\qquad \mathsf{let}\ v_1 = \mathsf{cons}\ v_1\ v_2\ \mathsf{in}
$$

Once the compilers language had been extended with sufficiently many such primitive operations, a LISP interpreter was compiled using our proof-producing compiler. The top-level specification function defining a simple LISP interpreter *lisp_eval* is listed in Figure 1. When *lisp_eval* is compiled, code is generated and a theorem is proved which state that this LISP interpreter is implemented by the generated machine code, in this case ARM code.

$$
\begin{array}{l}
\mathit{lisp\_eval\_pre}(v_1, v_2, v_3, v_4, v_5, v_6, l)\ \Rightarrow \\
\{\ \mathsf{lisp}\ (a, l)\ (v_1, v_2, v_3, v_4, v_5, v_6) * \mathsf{s} * \mathsf{pc}\ p\ \} \\
\ p : \text{... the generated code ...} \\
\{\ \mathsf{lisp}\ (a, l)\ (\mathit{lisp\_eval}(v_1, v_2, v_3, v_4, v_5, v_6, l)) * \mathsf{s} * \mathsf{pc}\ (p + 3012)\ \}
\end{array}
$$

Here *lisp_eval_pre* has collected the various side-conditions that need to be true for proper execution of the code.

6 Summary and discussion of related work

This paper has described how an extensible proof-producing compiler can be implemented using decompilation into logic [18]. The implementation required

```
TASK_EVAL = Sym "nil"
TASK_CONT = Sym "t"

lisp_lookup (v1,v2,v3,v4,v5,v6) = ...
lisp_eval0 (v1,v2,v3,v4,v5,v6,l) = ...
lisp_eval1 (v1,v2,v3,v4,v5,v6,l) = ...

lisp_eval (v1,v2,v3,v4,v5,v6,l) =
  if v2 = TASK_EVAL then
    let v2 = TASK_CONT in
      if isSym v1 then (* exp is Sym *)
        let (v1,v2,v3,v4,v5,v6) = lisp_lookup (v1,v2,v3,v4,v5,v6) in
          lisp_eval (v1,v2,v3,v4,v5,v6,l)
      else if isDot v1 then (* exp is Dot *)
        let v2 = CAR v1 in
        let v1 = CDR v1 in
        let (v1,v2,v3,v4,v5,v6,l) = lisp_eval0 (v1,v2,v3,v4,v5,v6,l) in
          lisp_eval (v1,v2,v3,v4,v5,v6,l)
      else (* exp is Num *)
        lisp_eval (v1,v2,v3,v4,v5,v6,l)
  else (* if v2 = TASK_CONT then *)
    if v6 = Sym "nil" then (* evaluation complete *)
      (v1,v2,v3,v4,v5,v6)
    else (* something is still on the to-do list v6 *)
      let (v1,v2,v3,v4,v5,v6,l) = lisp_eval1 (v1,v2,v3,v4,v5,v6,l) in
        lisp_eval (v1,v2,v3,v4,v5,v6,l)
```

Fig. 1. The top-level definition of *lisp_eval* in HOL4.

only a light-weight certification phase (approximately 100 lines of ML code) to be programmed, but still proves functional equivalence between the source and target programs. In contrast to previous work [8, 12–14], correctness proofs are here separated from code generation.

For each run, the compiler generates code and then proves that the code is correct. This is an idea for which Pnueli et al. [20] coined the term *translation validation*. There are two basic approaches to translation validation: (1) code generation is instrumented to generate proofs, and (2) code generation proceeds as usual then the certification phase attempts to guess the proofs. Approach 1 is generally considered more feasible [21]. However, Necula [19] showed that approach 2 is feasible even for aggressively optimising compilers such as GNU gcc [2]. Necula built into his certification phase heuristics that attempt to guess which optimisations were performed. The compiler presented here also implements approach 2, but restricts the (initial) input language and the op-

timisations to such an extent that the certification phase does not need any guesswork.

An alternative to producing a proof for each run is to prove the compiler correct. A recent, particularly impressive, milestone in compiler verification was achieved by Leroy [11], who proved the correctness of an optimising compiler which takes a significant subset of C as input and produces PowerPC assembly code[4] as output. As part of this project Tristan and Leroy [22] verified multiple translation validators. Other recent work is [10, 15, 11, 3, 5]. We chose not to verify our compiler/translation validator, since our compiler constructs all of its proofs in the HOL4 theorem prover. The trusted computing base (TCB) of our compiler is HOL4 and the specifications of the target machine languages. It seems that the user-defined extensions such as those in the LISP example would have been much harder to implement in a verified compiler, since verifying a compiler involves defining a deep embedding of the input language.

The VLISP project [9], which produced verified on-paper proofs for an implementation of a larger subset of LISP, is related to the example above of constructing a verified LISP interpreter. The fact that the proof presented here is mechanised and goes down to detailed models of commercial machine languages distinguishes this work from the VLISP project which stopped at the level of verified algorithms.

Acknowledgements. We thank Anthony Fox, Xavier Leroy and Susmit Sarkar for allowing us to use their processor models for this work. We also thank Thomas Tuerk, Aaron Coble and the anonymous reviewers for comments on earlier drafts. The first author is grateful for funding from EPSRC, UK.

References

1. The Netwide Assembler. http://www.nasm.us/.
2. The GNU Project. GCC, the GNU Compiler Collection. http://gcc.gnu.org/.
3. Nick Benton and Uri Zarfaty. Formalizing and verifying semantic type soundness of a simple compiler. In Michael Leuschel and Andreas Podelski, editors, *Principles and Practice of Declarative Programming (PPDP)*, pages 1–12. ACM, 2007.
4. C. J. Cheney. A non-recursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, 1970.
5. Adam J. Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *Programming Language Design and Implementation (PLDI)*, pages 54–65. ACM, 2007.
6. Karl Crary and Susmit Sarkar. Foundational certified code in a metalogical framework. Technical Report CMU-CS-03-108, Carnegie Mellon University, 2003.
7. Anthony Fox. Formal specification and verification of ARM6. In David Basin and Burkhart Wolff, editors, *Proceedings of Theorem Proving in Higher Order Logics (TPHOLs)*, volume 2758 of *LNCS*. Springer, 2003.

[4] The work presented here builds on Leroy's specification of PowerPC assembly code.

8. Mike Gordon, Juliano Iyoda, Scott Owens, and Konrad Slind. Automatic formal synthesis of hardware from higher order logic. *Electr. Notes Theor. Comput. Sci.*, 145:27–43, 2006.
9. Joshua Guttman, John Ramsdell, and Mitchell Wand. VLISP: A verified implementation of scheme. *Lisp and Symbolic Computation*, 8(1/2):5–32, 1995.
10. Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.*, 28(4):619–695, 2006.
11. Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Principles of Programming Languages (POPL)*, pages 42–54. ACM Press, 2006.
12. Guodong Li, Scott Owens, and Konrad Slind. A proof-producing software compiler for a subset of higher order logic. In *European Symposium on Programming (ESOP)*, LNCS, pages 205–219. Springer-Verlag, 2007.
13. Guodong Li and Konrad Slind. Compilation as rewriting in higher order logic. In Frank Pfenning, editor, *Automated Deduction (CADE)*, volume 4603 of *LNCS*, pages 19–34. Springer, 2007.
14. Guodong Li and Konrad Slind. Trusted source translation of a total function language. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963 of *LNCS*, pages 471–485. Springer, 2008.
15. Thomas Meyer and Burkhart Wolff. Tactic-based optimized compilation of functional programs. In Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, editors, *TYPES*, volume 3839 of *Lecture Notes in Computer Science*, pages 201–214. Springer, 2004.
16. Magnus O. Myreen, Anthony C.J. Fox, and Michael J.C. Gordon. A Hoare logic for ARM machine code. In *IPM International Symposium on Fundamentals of Software Engineering (FSEN)*, LNCS. Springer-Verlag, 2007.
17. Magnus O. Myreen and Michael J.C. Gordon. A Hoare logic for realistically modelled machine code. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS, pages 568–582. Springer-Verlag, 2007.
18. Magnus O. Myreen, Konrad Slind, and Michael J. C. Gordon. Machine-code verification for multiple architectures – An application of decompilation into logic. In *Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2008.
19. George C. Necula. Translation validation for an optimizing compiler. In *Programming Language Design and Implementation (PLDI)*, pages 83–94. ACM, 2000.
20. Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 1384 of *LNCS*, pages 151–166. Springer, 1998.
21. Martin C. Rinard. Credible compilation. In *Compiler Construction (CC)*. Springer, 1999.
22. Jean-Baptiste Tristan and Xavier Leroy. Formal verification of translation validators: a case study on instruction scheduling optimizations. In *Principles of Programming Languages (POPL)*, pages 17–27. ACM, 2008.