Machine-code verification for multiple architectures

An application of decompilation into logic

Magnus O. Myreen, Michael J. C. Gordon
University of Cambridge
Computer Laboratory
William Gates Building, 15 JJ Thomson Avenue
Cambridge CB3 0FD, UK
Email: {magnus.myreen, mike.gordon}@cl.cam.ac.uk

Konrad Slind
University of Utah
School of Computing
50 South Central Campus Drive
Salt Lake City UT84112, USA
Email: slind@cs.utah.edu

***Abstract*— Realistic formal specifications of machine languages for commercial processors consist of thousands of lines of definitions. Current methods support trustworthy proofs of the correctness of programs for one such specification. However, these methods provide little or no support for reusing proofs of the same algorithm implemented in different machine languages. We describe an approach, based on proof-producing decompilation, which both makes machine-code verification tractable and supports proof reuse between different languages. We briefly present examples based on detailed models of machine code for ARM, PowerPC and x86. The theories and tools have been implemented in the HOL4 system.**

I. Introduction

This paper concerns verification of programs written in machine code of commercial processors whose semantics is accurately specified. Such processors support a large number of instructions and a multitude of features. Our aim is to be able to verify machine code:

A: without introducing simplifying assumptions, and

B: not requiring expert knowledge of the models, while still

C: allowing reuse of proofs between different architectures.

Current approaches struggle to address challenge C, as they either involve direct reasoning about the next-state function [1] or are based on annotating the code with assertions [2], [3]. Annotating the code with assertions inevitably ties the proof to the specific code and machine model as assertions are mixed with the code and depend on machine-specific resource names.

Our contribution is a method for addressing challenges A, B and C. Our approach adds a thin layer of abstraction to the verification process in order to make verification proofs tractable and reuseable. A fully automatic decompiler is presented, which translates machine code, via automatic deduction, into tail-recursive functions defined in the language of a theorem prover. Given a sequence of machine-code instructions, the decompiler derives a tail-recursive function and proves a theorem stating that the function accurately describes the effect of the given machine code (addresses challenge A). The user can concentrate on proving properties of the generated function, which hides irrelevant details of the underlying machine language specification (challenge B). Properties proved about the generated function are, via an automatically derived theorem, related to the execution of the original machine code. The function describes the executed low-level algorithm and is likely to be very similar (illustrated in Section II-C) to another function describing the same algorithm implemented in a different machine language and thus can facilitate proof reuse (challenge C). The decompiler and all examples, presented in this paper, have been implemented in the HOL4 theorem prover (our theories and tools are available from [4]).

Notation. We write program specifications as *Hoare triples* $\{p\}\, c\, \{q\}$; informal meaning: if p holds for the current state then code c will leave the process in a state satisfying q (formal definition given in Section III-B).

II. Example

This section shows how decompilation aids verification. Subsequent sections describe the decompilation algorithm.

A. Running the automation

Consider the following ARM machine code (and assembly code on the right) which calculates the length of a linked-list. The code sets register 0 to zero; it then compares register 1 (the list pointer) with zero (nil), the last three instructions execute conditionally based on the result of this comparison, if register 1 is not zero, then the last three instructions increment register 0, load register 1 from memory and jumps back to the compare instruction, otherwise they do nothing.

```
 0:  E3A00000        mov r0, #0
 4:  E3510000    L:  cmp r1, #0
 8:  12800001        addne r0, r0, #1
12:  15911000        ldrne r1, [r1]
16:  1AFFFFFB        bne L
```

Given the above list of hexadecimal numbers, our decompiler produces a function f describing the effect of the code.

$$f(r_0, r_1, m) = \mathsf{let}\ r_0 = 0\ \mathsf{in}\ g(r_0, r_1, m)$$

$$g(r_0, r_1, m) = \mathsf{if}\ r_1 = 0\ \mathsf{then}\ (r_0, r_1, m)\ \mathsf{else}\ \mathsf{let}\ r_0 = r_0{+}1\ \mathsf{in}\ \mathsf{let}\ r_1 = m(r_1)\ \mathsf{in}\ g(r_0, r_1, m)$$

The decompiler also automatically proves the following theorem relating the execution of the ARM code with the function f (and an automatically generated precondition f_{pre}, given in Section IV-F). For now informally read the following Hoare-triple specification as (defined in Section III-B): given a state where register 0, register 1 and a part of memory is described by (r_0, r_1, m), the program counter is p and precondition f_{pre} holds, then executing the code will leave the processor in a state where register 0, register 1, a part of memory is described by $f(r_0, r_1, m)$ and the program counter is $p + 20$.

$$\{\ (\mathsf{r0}, \mathsf{r1}, \mathsf{m})\ \mathsf{is}\ (r_0, r_1, m) * \mathsf{pc}\ p * f_{pre}(r_0, r_1, m)\ \}$$
$$p : \texttt{E3A00000},\ p{+}4 : \texttt{E3510000}\ \ldots\ p{+}16 : \texttt{1AFFFFFB}$$
$$\{\ (\mathsf{r0}, \mathsf{r1}, \mathsf{m})\ \mathsf{is}\ f(r_0, r_1, m) * \mathsf{pc}\ (p + 20)\ \}$$

The precondition $f_{pre}(r_0, r_1, m)$ states the side-conditions which must hold for f to execute properly.

B. Verifying the code

In order to verify that the above code computes the length of a linked-list, we need to formalise the statement "the memory holds a linked-list". Let $list(l, a, m)$ be a recursively-defined predicate which states that an abstract list of 32-bit words l, e.g. $l = [4, 5] = 4{::}5{::}\mathsf{nil}$ (we write '::' for list cons), is represented by a linked-list in memory m with its head at address a. Each element of the list is represented by a word for the next pointer and a word for the data. The words are positioned 4 bytes apart, hence "+4" below.

$$\begin{aligned} list(\mathsf{nil}, a, m) &= a = 0 \\ list(x{::}l, a, m) &= \exists a'.\ m(a) = a' \wedge m(a{+}4) = x \wedge a \neq 0 \wedge \\ &\qquad list(l, a', m) \wedge aligned(a) \end{aligned}$$

Let $length(l)$ be the length of an abstract list l, e.g. $length(4{::}5{::}\mathsf{nil}) = 2$. It is now easy (14 lines of HOL4) to prove (by induction on the abstract list l) that function f from above calculates the length of a linked list and also that $list$ implies the precondition f_{pre} (which implies that f and the machine code terminates).

$$\forall x\, l\, a\, m.\ list(l, a, m) \Rightarrow f(x, a, m) = (length(l), 0, m)$$
$$\forall x\, l\, a\, m.\ list(l, a, m) \Rightarrow f_{pre}(x, a, m)$$

These lemmas about the generated function f can be given to a proof tactic, which automatically proves (using the theorem produced by the decompiler) that the original ARM code calculates $(length(l), 0, m)$, i.e. it proves a specification about the original code which does not involve the automatically generated function f or precondition f_{pre}:

$$\{\ (\mathsf{r0}, \mathsf{r1}, \mathsf{m})\ \mathsf{is}\ (r_0, r_1, m) * \mathsf{pc}\ p * list(l, r_1, m)\ \}$$
$$p : \texttt{E3A00000}\ \ldots\ p{+}16 : \texttt{1AFFFFFB}$$
$$\{\ (\mathsf{r0}, \mathsf{r1}, \mathsf{m})\ \mathsf{is}\ (length(l), 0, m) * \mathsf{pc}\ (p + 20)\ \}$$

Hence by proving a property of the abstract function f we have proved a property about the ARM code.

C. Reusing the proof

An interesting aspect of our approach (addressing challenge C from above) is that it facilitates reuse of proofs, even between different architectures. To illustrate this point consider the following x86 code,

```
 0:  31C0            xor eax, eax
 2:  85F6       L1:  test esi, esi
 4:  7405            jz L2
 6:  40              inc eax
 7:  8B36            mov esi, [esi]
 9:  EBF7            jmp L1
               L2:
```

and also the following PowerPC code for calculating the length of a linked-list.

```
 0:  38A00000          addi 5,0,0
 4:  2C140000     L1:  cmpwi 20,0
 8:  40820010          bc 4,2,L2
12:  7E80A02E          lwzx 20,0(20)
16:  38A50001          addi 5,5,1
20:  4BFFFFF0          b L1
                  L2:
```

Since the functional behaviour of all three code examples are essentially the same, the functions describing their behaviour are almost identical. f' is the function extracted for the x86 code and f'' is the same for PowerPC. We write '$\otimes$' for bitwise xor and '$\&$' for bitwise and.

$$f'(eax, esi, m) = \mathsf{let}\ eax = eax \otimes eax\ \mathsf{in}\ g'(eax, esi, m)$$

$$\begin{aligned} g'(eax, esi, m) = {} & \mathsf{if}\ esi\,\&\,esi = 0\ \mathsf{then}\ (eax, esi, m)\ \mathsf{else} \\ & \mathsf{let}\ eax = eax{+}1\ \mathsf{in} \\ & \mathsf{let}\ esi = m(esi)\ \mathsf{in} \\ & g'(eax, esi, m) \end{aligned}$$

$$f''(r_5, r_{20}, m) = \mathsf{let}\ r_5 = 0\ \mathsf{in}\ g''(r_5, r_{20}, m)$$

$$\begin{aligned} g''(r_5, r_{20}, m) = {} & \mathsf{if}\ r_{20} = 0\ \mathsf{then}\ (r_5, r_{20}, m)\ \mathsf{else} \\ & \mathsf{let}\ r_{20} = m(r_{20})\ \mathsf{in} \\ & \mathsf{let}\ r_5 = r_5{+}1\ \mathsf{in} \\ & g''(r_5, r_{20}, m) \end{aligned}$$

Minor differences, such as register names, conditional execution (ARM), variable instruction length (x86) and some instruction reordering (PowerPC example has load before increment), disappear in the functional description of the behaviour of the code. As a result the extracted functions can be proved equal by a short proof, in this case a three line HOL4 proof, using facts $w \otimes w = 0$ and $w\,\&\,w = w$.

$$f = f' = f'' \quad \text{and} \quad f_{pre} = f'_{pre} = f''_{pre}$$

Thus, any result proved for f and f_{pre} also describes the x86 and PowerPC implementations. By applying an automatic proof tactic we immediately obtain the same specification for the x86 code:

$$\{\ (\mathsf{eax}, \mathsf{esi}, \mathsf{m})\ \mathsf{is}\ (eax, esi, m) * \mathsf{eip}\ p * list(l, esi, m)\ \}$$
$$p : \texttt{31C0}\ \ldots\ p{+}9 : \texttt{EBF7}$$
$$\{\ (\mathsf{eax}, \mathsf{esi}, \mathsf{m})\ \mathsf{is}\ (length(l), 0, m) * \mathsf{eip}\ (p + 11)\ \}$$

and similarly for the PowerPC code:

$$\{\ (\mathsf{r5}, \mathsf{r20}, \mathsf{m})\ \mathsf{is}\ (r_5, r_{20}, m) * \mathsf{pc}\ p * list(l, r_{20}, m)\ \}$$
$$p : \texttt{38A00000}\ \ \ldots\ \ p{+}20 : \texttt{4BFFFFF0}$$
$$\{\ (\mathsf{r5}, \mathsf{r20}, \mathsf{m})\ \mathsf{is}\ (length(l), 0, m) * \mathsf{pc}\ (p + 24)\ \}$$

The decompiler automates the machine-specific proofs and delivers completely automatically to the user a recursive function describing the code. The generated functions are sufficiently abstract to be reusable while at the same time have a strong connection with the original machine code enabling properties of the function to carry over to properties proved of the machine code.

D. Larger examples

Our decompilation technique has been applied to a number of verification examples. The most significant are the verification of two copying garbage collectors (variants of the Cheney garbage collector [5]). The largest examples consist of approximately one hundred machine instructions.

Similar Cheney collectors have been verified by Birkedal et al. [6] (on paper) and McCreight et al. [7] (using Coq). The proofs by McCreight et al. (7775 lines) are much longer than our (approximately 2000-line) proofs, which suggests that decompilation aided our verification effort. Our decompiler is currently implemented in 1123 lines of ML with approximately 2300 lines of supporting proof scripts, and works on top of models of x86, ARM and PowerPC whose HOL4 definitions total more than 10000 lines.

III. Preliminaries

Before presenting the algorithm for decompilation into logic (next section), we overview the models used in our examples, the Hoare triples, and our approach to proving loops.

A. Mechanised models of x86, ARM and PowerPC

Our current implementation supports x86, ARM and PowerPC machine code. The underlying operational models of the machine languages are detailed descriptions of a next-state function $next : state \rightarrow state$, which executes one instruction for each next application.

The machine language models were not developed for use with our tool. The ARM model is a specification of ARMv4 written by Fox [8], which he proved correct with respect to a register-transfer-level specification of an ARM processor (ARM6). The PowerPC model is a translation (manual translation of Coq to HOL4) of Leroy's PowerPC specification, which he used in a proof of an optimising C compiler [9]; we attached an instruction decoder to it in order to transform his assembly level model into a machine code model of PowerPC. The x86 specification is a functional HOL4 version of Sarkar's x86 specification [10], developed originally in Twelf for use in a applications of proof-carrying code. All the models we base this work on are available with our HOL4 proof scripts [4].

B. Hoare triple specifications

This section defines, in higher-order logic, the Hoare triples we use to make specifications concise and manageable. These are a stream-lined version of previously presented Hoare triples for machine code [11], [12].

The Hoare triples view states as sets of state elements. In order to accommodate this view, we define translations from the states used by the models to sets of state elements, e.g. for PowerPC we define the type ppc_elem of state elements with the following type constructors:

$$\mathsf{pReg} : \mathsf{ppc\_reg\_name} \times \mathsf{word32} \rightarrow \mathsf{ppc\_elem}$$
$$\mathsf{pMem} : \mathsf{word32} \times \mathsf{word8\ option} \rightarrow \mathsf{ppc\_elem}$$
$$\mathsf{pStatus} : \mathsf{ppc\_bit\_name} \times \mathsf{boolean\ option} \rightarrow \mathsf{ppc\_elem}$$

and define a translation function $ppc2set$, which translates states from the processor model's format to states in the set-based representation. The following is an example of the output from $ppc2set$: (GPR = general purpose register)

$$\{\ \mathsf{pReg}\ (\mathsf{GPR}\ 0)\ 5, \mathsf{pReg}\ (\mathsf{GPR}\ 1)\ 56, \mathsf{pReg}\ (\mathsf{GPR}\ 2)\ 89, \ldots,$$
$$\mathsf{pMem}\ 0\ \mathsf{none}, \mathsf{pMem}\ 1\ (\mathsf{some}\ 67), \mathsf{pMem}\ 2\ (\mathsf{some}\ 255), \ldots,$$
$$\mathsf{pStatus}\ (\mathsf{CPR0}\ 0)\ (\mathsf{some}\ true), \mathsf{pStatus}\ (\mathsf{CPR0}\ 1)\ \mathsf{none}, \ldots\ \}$$

The Hoare triple uses a separating conjunction $*$ inspired by separation logic. Separation logic defines its $*$ over partial functions; we define ours over sets:

$$(p * q)\ s = \exists u\ v.\ p\ u \wedge q\ v \wedge (u \cup v = s) \wedge (u \cap v = \{\})$$

The separating conjunction splits the state into two disjoint parts. Basic assertions access only part of the state, e.g. asserting that GPR 0 has value x is defined as:

$$(\mathsf{r0}\ x)\ s = (s = \{\mathsf{pReg}\ (\mathsf{GPR}\ 0)\ x\})$$

Basic assertions together with the $*$-operator consume a part of the state, e.g. $(\mathsf{r0}\ x{*}res)\ (ppc2set(s))$ is false if res makes an assertion about the value of GPR 0.

Let $run(n, s)$ be a function which applies the $next$-function n-times to s, i.e.

$$run(0, s) = s$$
$$run(n{+}1, s) = run(n, next(s))$$

The Hoare triple is defined to assert: for any state s for which a portion satisfies p and a separate portion satisfies a code assertion (defined, together with other memory assertion, in Section IV-K), there exists some number of $next$ applications which will take the processor to a state where q satisfies a portion of the state separate from the code.

$$\{p\}\ c\ \{q\} =$$
$$\forall s\ r.\ (p * \mathsf{code}\ c * r)\ (ppc2set(s)) \Rightarrow$$
$$\exists k.\ (q * \mathsf{code}\ c * r)\ (ppc2set(run(k, s)))$$

Thus $\{p\}\ c\ \{q\}$ is a total-correctness specification.

The frame rule can be derived from this definition: it allows any assertion for a disjoint portion of the state to be added to the specification:

$$\{p\}\ c\ \{q\} \Rightarrow \forall r.\ \{p * r\}\ c\ \{q * r\}$$

Other important rules are composition, which combines two specifications and takes the set-union of their code elements:

$$\{p\}\ c_1\ \{m\}\ \wedge\ \{m\}\ c_2\ \{q\}\ \Rightarrow\ \{p\}\ c_1 \cup c_2\ \{q\}$$

and a rule for moving pure assertions g (an assertion g is 'pure' if it consumes no resources, i.e. $\forall p\ s.\ (p * g)\ s = p\ s \wedge g$) out of the precondition:

$$\{p * (g)\}\ c\ \{q\}\ =\ (g \Rightarrow \{p\}\ c\ \{q\})$$

Note that these rules are just theorems of higher-order logic proved from the definitions of Hoare triples and separating conjunction given above.

C. Proving loops

When proving specifications for code with loops our decompilation algorithm instantiates a special loop-rule for tail-recursive functions. This rule assumes the existence of a termination proof for the tail-recursive function and uses the induction arising from the termination proof in proving the specification for the code implementing the tail-recursion.

Decompilation only generates tail-recursive functions, i.e. functions $tailrec$ with instantiations of G, F and D, where:

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

$tailrec$ can be defined directly (without a termination proof in HOL4) using a trick by Manolios and Moore [13].

However, the decompiler requires $tailrec$ to terminate for certain inputs. Tail-recursions defined by $tailrec$ terminate for input x if and only if some number of applications of F to x make G false, i.e. $\exists n.\ \neg G(pow(n, F, x))$ with pow as:

$$\begin{aligned} pow(0, F, x)\ &=\ x \\ pow(n{+}1, F, x)\ &=\ pow(n, F, F(x)) \end{aligned}$$

We define $pre(x)$ to state that $tailrec(x)$ terminates and also that side-condition P is true for each call to $tailrec$ (Section IV-E illustrates an instantiation of P).

$$\begin{aligned} pre(x)\ =\ &\exists n.\ \neg\, G\,(pow(n, F, x)) \wedge \\ &\forall k.\ (\forall m.\ m < k \Rightarrow G\,(pow(m, F, x))) \Rightarrow \\ &\quad P\,(pow(k, F, x)) \end{aligned}$$

pre satisfies two desirable properties: pre can be unrolled by a rewrite (particularly useful in proofs by induction):

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

and the following induction can be derived from its definition:

$$\begin{aligned} \forall \varphi.\ &(\forall x.\ pre(x) \wedge G(x) \wedge \varphi(F(x)) \Rightarrow \varphi(x)) \wedge \\ &(\forall x.\ pre(x) \wedge \neg G(x) \Rightarrow \varphi(x)) \Rightarrow \\ &(\forall x.\ pre(x) \Rightarrow \varphi(x)) \end{aligned}$$

This induction rule leads to the following loop rule, which the decompiler instantiates whenever it encounters a loop (an example instantiation is given in Section IV-E).

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

For the proof of this rule, instantiate φ in the induction principle above with $\lambda x.\ \{\mathsf{res}\ x\}\, c\, \{\mathsf{res'}\ tailrec(x)\}$ and use the composition rule and the inductive hypothesis together with $c \cup c = c$ and $G(x) \Rightarrow tailrec(F(x)) = tailrec(x)$.

IV. Algorithm

This section outlines our algorithm for decompiling machine code into logic. There are six steps:

1) calculate the behaviour of each individual instruction;
2) prove a specification for each instruction;
3) discover the control flow by analysing the specifications;
4) split the code according to the control-flow graph;
5) for each code segment:
   a) derive a specification for one pass through the code,
   b) generate a function describing the code;
   c) for loops, instantiate a loop rule.
6) compose the top-level specifications and repeat step 5 until all of the code is described by one specification.

The following subsections explain these steps when applied to the linked-list example from Section II. Later subsections describe support for procedure calls as well as non-nested loops. Restrictions of our approach are outlined in Section V.

A. Behaviour of instructions

As a first step, each instruction's effect on the underlying machine-language model is evaluated. We use standard techniques from symbolic simulation to construct statements about the next-state function. As an example: the x86 instruction `40`, which is the hexadecimal encoding of `inc eax` (increment the EAX register), produces the following theorem. Here eip is the instruction pointer, a.k.a. program counter, and AF, SF, ZF etc. are status bits called "eflags". Here and throughout option-types[1] are used for values that may hold unpredictable or unmodelled values, e.g. the theorem shows that eflag OF gets an unmodelled value, while eflag ZF is assigned the boolean result of the comparison $eax + 1 = 0$.

$$\begin{aligned} &x86\_read\_reg\ \mathsf{EAX}\ s = eax\ \wedge \\ &x86\_read\_eip\ s = eip\ \wedge \\ &x86\_read\_mem\ eip\ s = \mathsf{some\ 0x40}\ \Rightarrow \\ &\quad x86\_next\ s = \\ &\quad\ \mathsf{some}\ (x86\_write\_reg\ \mathsf{EAX}\ (eax + 1) \\ &\qquad (x86\_write\_eip\ (eip + 1) \\ &\qquad (x86\_write\_eflag\ \mathsf{AF\ none} \\ &\qquad (x86\_write\_eflag\ \mathsf{SF}\ (\mathsf{some}\ (sign\_of(eax + 1))) \\ &\qquad (x86\_write\_eflag\ \mathsf{ZF}\ (\mathsf{some}\ (eax + 1 = 0)) \\ &\qquad (x86\_write\_eflag\ \mathsf{PF}\ (\mathsf{some}\ (parity(eax + 1))) \\ &\qquad (x86\_write\_eflag\ \mathsf{OF\ none}\ s))))))) \end{aligned}$$

[1] something of type option is either 'some x', meaning 'has value x', or 'none' meaning 'has an unmodelled or unpredictable value'.

B. Instruction specifications

As a second step we derive Hoare-triple specifications for each instruction in the given program, e.g. the move-instruction from the ARM code for length-of-linked-list has the following specification:

$$\{\, \mathsf{r0}\ r_0 * \mathsf{pc}\ p \,\}\ \ p : \texttt{0000A0E3}\ \ \{\, \mathsf{r0}\ 0 * \mathsf{pc}\ (p{+}4) \,\}$$

Two specifications are produced for instructions that execute conditionally, e.g. the ARM instruction for branch-if-not-equal: (sz asserts the value of status bit 'z')

$$\{\, \mathsf{sz}\ z * \mathsf{pc}\ (p{+}16) * \neg z \,\}\ \ p : \texttt{FBFFFF1A}\ \ \{\, \mathsf{sz}\ z * \mathsf{pc}\ (p{+}4) \,\}$$

$$\{\, \mathsf{sz}\ z * \mathsf{pc}\ (p{+}16) * z \,\}\ \ p : \texttt{FBFFFF1A}\ \ \{\, \mathsf{sz}\ z * \mathsf{pc}\ (p{+}20) \,\}$$

C. Control-flow discovery

A heuristic reads the pc assertions in the postconditions and builds a summary of how control can flow. The linked-list example results in the following description:

$$0 \rightarrow 4,\ 4 \rightarrow 8,\ 8 \rightarrow 12,\ 12 \rightarrow 16,\ 16 \rightarrow 20,\ 16 \rightarrow 4$$

The heuristic searches for loops by analysing this graph. It finds that instructions $4, 8, 12, 16$ constitute a loop.

D. Finding the function

Once loops have been detected in the control-flow graph, we start by proving a specification for the inner-most loop. We compose specifications for individual instructions in order to get a specification for one pass of execution through the code. Composing the specifications for the loop in the ARM code of the linked-list example results in two specifications[2][3]; one for the case $r_1 = 0$:

$$\{\, \mathsf{r0}\ r_0 * \mathsf{r1}\ r_1 * \mathsf{m}\ m * \mathsf{pc}\ (p{+}4) \,\}$$
$$\textit{... the arm code ...}$$
$$\{\, \mathsf{r0}\ r_0 * \mathsf{r1}\ r_1 * \mathsf{m}\ m * \mathsf{pc}\ (p{+}20) \,\}$$

and one for the case $r_1 \neq 0$, if $r_1 \in dom(m) \wedge aligned(r_1)$:

$$\{\, \mathsf{r0}\ r_0 * \mathsf{r1}\ r_1 * \mathsf{m}\ m * \mathsf{pc}\ (p{+}4) \,\}$$
$$\textit{... the arm code ...}$$
$$\{\, \mathsf{r0}\ (r_0{+}1) * \mathsf{r1}\ (m(r_1)) * \mathsf{m}\ m * \mathsf{pc}\ (p{+}4) \,\}$$

Notice that the program counter is returned to $p{+}4$ in case $r_1 \neq 0$, indicating that the function describing the code is to loop when $r_1 \neq 0$. The generated function is constructed to mimic the effect of the code:

$$g(r_0, r_1, m) = \mathsf{if}\ r_1 = 0\ \mathsf{then}\ (r_0, r_1, m)\ \mathsf{else}\ g(r_0{+}1, m(r_1), m)$$

[2]Our implementation inserts let-expressions at this stage but we avoid them in our illustrations in order to reduce the size of expressions.

[3]To be completely accurate, from this point onwards all pre- and postconditions in this paper should end in "... * s}". Here s existentially quantifies the values of the status bits, i.e. the Hoare triples abstract the values of the status bits: they state "... and the status bits have some value before and after execution". One can turn off this abstraction and keep track of the exact value of the status bits just as for any other resource.

E. Proving the specification

The generated function is always a tail-recursion (if recursive at all). This means that the function can be defined as an instance of $tailrec$ from Section III-C. Function g from above is defined as $tailrec$ with F, G and D as:

$$G = \lambda(r_0, r_1, m).\ r_1 \neq 0$$
$$F = \lambda(r_0, r_1, m).\ (r_0{+}1, m(r_1), m)$$
$$D = \lambda(r_0, r_1, m).\ (r_0, r_1, m)$$

and the precondition g_{pre} is defined as pre (also from Section III-C) with the same instantiations, and parameter P as:

$$P = \lambda(r_0, r_1, m).\ r_1 \neq 0 \Rightarrow r_1 \in dom(m) \wedge aligned(r_1)$$

P is defined as such in order for $g_{pre}(r_0, r_1, m)$ to imply the side condition appearing in case $r_1 \neq 0$ of Section IV-D.

Let resource assertions res and res' be:

$$\mathsf{res} = \lambda(r_0, r_1, m).\ (\mathsf{r0}, \mathsf{r1}, \mathsf{m})\ \mathsf{is}\ (r_0, r_1, m) * \mathsf{pc}\ (p{+}4)$$
$$\mathsf{res}' = \lambda(r_0, r_1, m).\ (\mathsf{r0}, \mathsf{r1}, \mathsf{m})\ \mathsf{is}\ (r_0, r_1, m) * \mathsf{pc}\ (p{+}20)$$

With these instantiations the conclusion of the loop-rule from Section III-C is exactly the desired result for the loop:

$$\{\, (\mathsf{r0}, \mathsf{r1}, \mathsf{m})\ \mathsf{is}\ (r_0, r_1, m) * \mathsf{pc}\ (p{+}4) * g_{pre}(r_0, r_1, m) \,\}$$
$$\textit{... the arm code ...}$$
$$\{\, (\mathsf{r0}, \mathsf{r1}, \mathsf{m})\ \mathsf{is}\ g(r_0, r_1, m) * \mathsf{pc}\ (p{+}20) \,\}$$

The premises of the loop-rule are trivial consequences (by simple rewriting in HOL4) of the theorems describing one pass through the code, given in Section IV-D.

F. Merging cases recursively

Sections IV-D and IV-E showed how tail-recursive functions $tailrec$ and specifications of the form

$$\{\mathsf{res}\ x * \mathsf{pc}\ (...) * pre(x)\}\ c\ \{\mathsf{res'}\ tailrec(x) * \mathsf{pc}\ (...)\}$$

can be constructed and proved for code with at most one top-level loop, given specifications for each individual instruction of the code c, obtained in Sections IV-A and IV-B.

Specification for nested loops and code around loops can be proved by recursively repeating the above procedure for code enclosing the inner loops. For the linked-list example the decompilation algorithm will repeat Sections IV-D and IV-E based on the specification of the move-instruction (at the top of Section IV-B) and the above specification proved for g (at the end of Section IV-E), which then defines f and proves the theorem shown in Section II-A. The generated f_{pre} is defined as an instance of pre, but returned as the following theorem.

$$\begin{aligned} f_{pre}(r_0, r_1, m) &= g_{pre}(0, r_1, m) \\ g_{pre}(r_0, r_1, m) &= (r_1 \neq 0) \Rightarrow \\ &\qquad r_1 \in dom(m) \wedge aligned(r_1) \wedge \\ &\qquad g_{pre}(r_0{+}1, m(r_1), m) \end{aligned}$$

G. Non-nested loops

The examples above have considered machine-code programs that start executing at the top of the code and exit at the end of the code, with all intermediate loops properly nested. More general forms of control flow are handled by treating the program counter as any other resource, i.e. the program counter becomes part of the function '(..., pc) is $f(...)$', just as any other register value. In such cases, the position q of the code needs to be passed in to the generated function f. As an example, when the following non-nested loops are processed

```
 0:  E2800001   L: add  r0,r0,#1
 4:  E3100001   M: tst  r0,#1
 8:  1AFFFFFC      bne  L  ;; may goto L
12:  E2500002      subs r0,r0,#2
16:  1AFFFFFB      bne  M  ;; may goto M
```

the generated function compares the value of the program counter p with the position of the code q:

$$
\begin{array}{l}
f(r_0, p, q) = \\
\quad \textsf{if } p = q \textsf{ then} \\
\quad\quad \textsf{let } r_0 = r_0 + 1 \textsf{ in } f(r_0, q{+}4, q) \\
\quad \textsf{else if } r_0 \mathbin{\&} 1 \neq 0 \textsf{ then} \\
\quad\quad f(r_0, q, q) \\
\quad \textsf{else} \\
\quad\quad \textsf{let } r_0 = r_0 - 2 \textsf{ in} \\
\quad\quad\quad \textsf{if } r_0 = 0 \textsf{ then } (r_0, q{+}20) \textsf{ else } f(r_0, q{+}4, q)
\end{array}
$$

The resulting theorem is:

$$
\begin{array}{c}
\{\ (\mathsf{r0}, \mathsf{pc}) \textsf{ is } (r_0, p) * p \in \{q, q{+}4\}\ \} \\
q : \ldots code \ldots \\
\{\ (\mathsf{r0}, \mathsf{pc}) \textsf{ is } f(r_0, p, q)\ \}
\end{array}
$$

The decompiler instantiates q with p before using the above Hoare triple specification in subsequent proofs.

H. Procedure calls

Procedure calls are, in machine code, implemented using branch-and-link instructions. These branch instructions perform a normal branch and at the same time save a return address, e.g. on ARM the branch-and-link instruction stores the return address in register 14:

$$\{\mathsf{r14}\ x * \mathsf{pc}\ p\}\ p : \texttt{EB000009}\ \{\mathsf{r14}\ (p{+}4) * \mathsf{pc}\ (p{+}48)\}$$

Given the following specification for the procedure's code,

$$
\begin{array}{c}
\{\ (\mathsf{pc}, \mathsf{r14}, \mathsf{res}) \textsf{ is } (p, r_{14}, x) * t_{pre}(p, r_{14}, x)\ \} \\
p : procedure\_code \\
\{\ (\mathsf{pc}, \mathsf{r14}, \mathsf{res}) \textsf{ is } t(p, r_{14}, x)\ \}
\end{array}
$$

we can compose the call with the procedure's specification,

$$
\begin{array}{c}
\{\ (\mathsf{pc}, \mathsf{r14}, \mathsf{res}) \textsf{ is } (p, r_{14}, x) * t_{pre}(p{+}48, p{+}4, x)\ \} \\
p : \texttt{EB000009} \cup p{+}48 : procedure\_code \\
\{\ (\mathsf{pc}, \mathsf{r14}, \mathsf{res}) \textsf{ is } t(p{+}48, p{+}4, x)\ \}
\end{array}
$$

and by strengthening the precondition t_{pre} to assume that control returns to the callee, let $\mathsf{fst}(x, y) = x$,

$$t'_{pre}(p, q, x) = t_{pre}(p, q, x) \wedge (\mathsf{fst}(t(p, q, x)) = q)$$

we have a specification for the procedure which has the shape of a normal instruction specification (enters at pc p and exits at pc $(p{+}4)$). Thus specifications for procedure calls can be derived from the specification of the called procedure. The function generated by the decompiler includes a reference to function t generated for the procedures body.

$$\textsf{let } (\_, r_{14}, x) = t(p{+}48, p{+}4, x) \textsf{ in } \ldots$$

Procedural recursion poses a challenge as an induction is required. In principle, it is possible to support procedural induction by regarding the program counter as any other resource, as was done in the previous section. However, the generated function is far less intuitive using that technique.

I. Support for user-defined resource assertions

Notice that the operations of the decompiler do not depend on the particular properties of the basic resource assertions (r0, r1, m etc.). As a result, specifications involving completely different, user-defined, assertions can be fed into the decompiler for use instead of automatically proved instruction specifications.

As an example consider this Hoare triple describing the `alloc` routine of one the garbage collectors we have verified using decompilation (see Section II-D). Here heap is predicate stating that a garbage collected heap is present in memory. The alloc function's precondition states that the number of (#) reachable elements in the abstract heap h from roots v_1, v_2, v_3, v_4 must be less than the heap limit l. The post condition state that the abstract heap modelling function h is updated with a new element *fresh* h, which points at a cons cell containing (v_1, v_2). The address of the new element is stored in the place of root variable v_1.[4]

$$
\begin{array}{l}
\{\ \mathsf{pc}\ p * \#reachable(v_1, v_2, v_3, v_4, h) < l * \\
\quad \mathsf{heap}\ (a, v_1, v_2, v_3, v_4, h, l)\ \} \\
\qquad \ldots collector\ code \ldots \\
\{\ \mathsf{pc}\ (p{+}332) * \\
\quad \mathsf{heap}\ (a, \textit{fresh}\ h, v_2, v_3, v_4, h[\textit{fresh}\ h \mapsto (v_1, v_2)], l)\ \}
\end{array}
$$

When such specifications can be given as input to the decompiler, in our implementation using a special keyword '`insert` ...' in the given code, the decompiler can look-up and use this specification. The resulting generated function contains the roots v_1, v_2, v_3, v_4 and heap h as variables:

$$\textsf{let } (v_1, h) = (\textit{fresh}\ h,\ h[\textit{fresh}\ h \mapsto (v_1, v_2)]) \textsf{ in } \ldots$$

the theorem contains '$(\mathsf{heap}, \mathsf{r0}, \ldots)$ is ...', and the generated precondition will keep track of a sufficient condition under which the heap limit is not exceeded.

J. Code and memory assertions

The code and memory assertions are defined in this section. The code assertion for ARM is the simplest one: it states that

[4]Details of this specification for `alloc` and its proof will be part of the first author's forth coming PhD thesis.

the memory hold a set of (p, w) instructions, where p is the address and w is the 32-bit word (the instruction).

$$\mathsf{code}\ set\ s\ =\ (s = \{\ \mathsf{aMem}\ p\ (\mathsf{some}\ w)\ |\ (p, w) \in set\ \})$$

The memory assertion $\mathsf{m}\ m$ states that memory location a has value $m(a)$ if $a \in dom(m)$ and a is word-aligned (two least significant bits are zero, i.e. $a\,\&\,3 = 0$).

$$\mathsf{m}\ m\ =\ \mathsf{code}\ \{\ (a, m(a))\ |\ a \in dom(m) \wedge a\,\&\,3 = 0\ \}$$

The code assertions for PowerPC and x86 are slightly more complicated due to the fact that their set representation (and the underlying model) considers the memory as byte-addressed, i.e. a 32-bit word consists of four bytes. The code assertion for PowerPC (which is a big-endian architecture):

$$\begin{aligned} word(p, w) = \{\ &\mathsf{pMem}\ (p{+}0)\ (\mathsf{some}\ (w[31{-}24])),\\ &\mathsf{pMem}\ (p{+}1)\ (\mathsf{some}\ (w[23{-}16])),\\ &\mathsf{pMem}\ (p{+}2)\ (\mathsf{some}\ (w[15{-}08])),\\ &\mathsf{pMem}\ (p{+}3)\ (\mathsf{some}\ (w[07{-}00]))\ \} \end{aligned}$$

$$\mathsf{code}\ set\ s\ =\ (s = \bigcup\{\ word(p, w)\ |\ (p, w) \in set\ \})$$

The definition of m for PowerPC uses code just as ARM.

The code assertion for x86 is defined recursively, since x86 instructions are lists of bytes:

$$\begin{aligned} x86\_list(p, []) &= \{\}\\ x86\_list(p, c{::}cs) &= \{\ \mathsf{xMem}\ p\ (\mathsf{some}\ c)\ \} \cup ia\_list(p{+}1, cs) \end{aligned}$$

$$\mathsf{code}\ set\ s\ =\ (s = \bigcup\{\ ia\_list(p, cs)\ |\ (p, cs) \in set\ \})$$

x86's memory assertion takes into account that the architecture is little-endian:

$$\begin{aligned} word(p, w) = \{\ &\mathsf{xMem}\ (p{+}0)\ (\mathsf{some}\ (w[07{-}00])),\\ &\mathsf{xMem}\ (p{+}1)\ (\mathsf{some}\ (w[15{-}08])),\\ &\mathsf{xMem}\ (p{+}2)\ (\mathsf{some}\ (w[23{-}16])),\\ &\mathsf{xMem}\ (p{+}3)\ (\mathsf{some}\ (w[31{-}24]))\ \} \end{aligned}$$

$$\mathsf{m}\ m\ s\ =\ (s = \bigcup\{\ word(a, m(a))\ |\ a \in dom(m) \wedge a\,\&\,3 = 0\ \})$$

These assertions act as a thin layer of additional abstraction.

K. Memory separation

The examples presented so far have only used a single memory assertion $\mathsf{m}\ m$ at a time. However, it is often useful to separate memory into logical segments, e.g. one for the stack s and one for the heap h:

$$\{\mathsf{m}\ s * \mathsf{m}\ h * ...\}\ p : ...\ \{...\}$$

Notice that this specification implicitly assumes that s and h describe disjoint parts of memory, since '$*$' makes assertions 'consume' memory (Section III-B). The functions produced by the decompiler will then use two memory modelling functions s and h, and most importantly an update to the stack s will not affect the heap h (and *vice versa*). This feature is used heavily in some of the garbage collector proofs in order to avoid some proof obligations that arise from possible pointer aliasing between the stack and the heap (in case the stack and the heap happened to overlap, a case we rule out).

Memory separation can easily be implemented by modifying the output from the routines that derive Hoare triple specifications (Section IV-B). A heuristic is fed in to that routine which renames memory modelling functions depending on the registers that access them, e.g. our default heuristic renames memory modelling functions to s, if the stack pointer is used, while all other accesses are to memory called m.

V. Restrictions

Our method is completely automatic and is reasonably light-weight to implement. Restrictions of our approach are discussed below.

A. Deterministic behaviour required

The method is only applicable to programs that have deterministic behaviour, for otherwise the code is not a *function* of its inputs and the decompiler could not produce a function describing the code.

B. Heuristics used for control flow discovery

The decompiler uses heuristics to discover the possible execution paths in the code. The heuristics work well for code where all branches are made to offsets of the current program counter. Branch-and-link instructions are considered to be procedure calls and any instruction moving an address into the program counter from a register or stack location is assumed to perform a procedure return. As a result, our simple heuristic is easily confused by computed branches and calls to code pointers.

VI. Discussion of Related Work

Different techniques for program verification, with respect to accurate models of machine code, are discussed below.

Symbolic simulation is a technique applicable to machine code modelled by an operational semantics. The approach is based on executing the next-state function on states where registers and memory location have been assigned symbolic values (logical variables, e.g. x, y); the result is a new state where resources hold expressions (e.g. $x{+}y$), for which the verifier is to prove properties. This method is emphasised and successfully applied by the ACL2 community [14], [15], e.g. Boyer and Yu used symbolic simulation in pioneering work on verification of the GNU string library compiled by GCC for the Motorola MC68020 [1], and similar techniques were used by Liu and Moore in proofs of Java bytecode programs with respect to an extensive model of the Java Virtual Machine (JVM) [16]. Symbolic simulation has the disadvantages that the expressions produced by simulation, directly on top of the operation semantics, can become fiendishly complex, and loops reqiure user interaction. However, Currie et al. [17] have developed automatic tools, based on symbolic simulation, which prove equivalence between snippets of machine code for embedded devices.

Using a programming logic directly on top of the definition of the semantics of the machine code is an approach which lends itself well to reasoning about loops and control flow that is problematic for symbolic simulation. Shao's group at Yale [18] have used programming logics (inspired by separation logic and rely-guarantee) in verification of (slightly idealised) assembly programs. Foundational proof checking [19] and Typed Assembly Language [20], [21] also belong to this category. However, they aim to check relatively weak safety properties – while this paper's techniques are concerned with proving complete functional correctness.

Using *verification condition generators (VCGs)* one annotates the code with assertions for which the VCGs calculates verification conditions that imply consistency of the assertions with respect to some programming logic. The integrity of the VCG is a concern, as practical VCGs tend to be complex [22], [23]. However, Homeier and Martin [24] showed that VCGs can be verified and Matthews et al. [3] has showed that off-the-shelf theorem provers can be used in a way which gives the benefits of a VCG without actually constructing a full VCG. Hardin et al. [2] have applied the technique described by Matthews et al. to machine code of Rockwell Collins AAMP7G. The main disadvantage of annotating the code with assertions is that the assertions become tied to the specific machine language and/or the particular definition of the semantics and, thus, do not provide the appropriate abstractions required for proof reuse.

Decompilation automatically reverse-engineers an abstraction of machine code. Decompilation is most often used to reverse compilation from a language such as C [25], but can, as we have shown in this paper, be used to produce abstractions in higher-order logic – a language much more amenable to formal reasoning than C. There is generally little work in this area, but work by Filliâtre [26] and Katsumata and Ohori [27] is related to ours. Filliâtre shows how imperative loops can, in type theory, be turned into recursive functions for purposes of verification. Unlike our approach his requires the code to be annotated with invariants and does not apply the method to low-level languages. Katsumata and Ohori have developed a decompiler, from a small subset of idealised Java bytecode to recursive functions, based on ideas from type theory. The decompiler implementing their methodology has not been verified. It seems that the decompiler would need to be trusted, if its output were to be used in verification. In contrast our approach is to produce a proof for each run, hence the decompiler need not be trusted.

Acknowledgments

We would like to thank Thomas Tuerk, Anthony Fox, Boris Feigin, Max Bolingbroke and John Regehr for research discussions. The first author is grateful for funding from Osk. Huttusen säätiö, Finland, and EPSRC, UK.

References

[1] R. S. Boyer and Y. Yu, "Automated proofs of object code for a widely used microprocessor," *J. ACM*, vol. 43, no. 1, pp. 166–192, 1996.

[2] D. S. Hardin, E. W. Smith, and W. D. Young, "A robust machine code proof framework for highly secure applications," in *Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and Its Applications*, P. Manolios and M. Wilding, Eds., 2006.

[3] J. Matthews, J. S. Moore, S. Ray, and D. Vroon, "Verification condition generation via theorem proving," in *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, ser. LNCS, M. Hermann and A. Voronkov, Eds., vol. 4246. Springer, 2006, pp. 362–376.

[4] Project sources files available under 'HOL/examples/mc-logic' in the HOL4 distribution at SourceForge: http://hol.sourceforge.net/. 2008.

[5] C. J. Cheney, "A non-recursive list compacting algorithm," *Commun. ACM*, vol. 13, no. 11, pp. 677–678, 1970.

[6] L. Birkedal, N. Torp-Smith, and J. Reynolds, "Local reasoning about a copying garbage collector," in *Principles of programming languages (POPL)*. ACM Press, 2004, pp. 220–231.

[7] A. McCreight, Z. Shao, C. Lin, and L. Li, "A general framework for certifying garbage collectors and their mutators," in *Programming Language Design and Implementation (PLDI)*, J. Ferrante and K. S. McKinley, Eds. ACM, 2007, pp. 468–479.

[8] A. Fox, "Formal specification and verification of ARM6," in *Theorem Proving in Higher Order Logics (TPHOLs)*, ser. LNCS, D. Basin and B. Wolff, Eds., vol. 2758. Springer, 2003.

[9] X. Leroy, "Formal certification of a compiler back-end, or: programming a compiler with a proof assistant," in *33rd symposium Principles of Programming Languages POPL*. ACM Press, 2006, pp. 42–54.

[10] K. Crary and S. Sarkar, "Foundational certified code in a metalogical framework," Carnegie Mellon University, Tech. Rep. CMU-CS-03-108, 2003.

[11] M. O. Myreen and M. J. Gordon, "A Hoare logic for realistically modelled machine code," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. LNCS. Springer-Verlag, 2007.

[12] M. O. Myreen, A. C. Fox, and M. J. Gordon, "A Hoare logic for ARM machine code," in *International Symposium on Fundamentals of Software Engineering (FSEN)*, ser. LNCS. Springer-Verlag, 2007.

[13] P. Manolios and J. S. Moore, "Partial functions in ACL2," *J. Autom. Reasoning*, vol. 31, no. 2, pp. 107–127, 2003.

[14] R. S. Boyer and J. S. Moore, "Proving theorems about pure LISP fucntions," *JACM*, vol. 22, no. 1, pp. 129–144, 1975.

[15] J. S. Moore, "Symbolic simulation: An ACL2 approach," in *Formal Methods in Computer-Aided Design (FMCAD)*, 1998, pp. 334–350.

[16] H. Liu and J. S. Moore, "Java program verification via a JVM deep embedding in ACL2," in *Theorem Proving in Higher Order Logics (TPHOLs)*, ser. Lecture Notes in Computer Science, K. Slind, A. Bunker, and G. Gopalakrishnan, Eds., vol. 3223. Springer, 2004, pp. 184–200.

[17] D. Currie, X. Feng, M. Fujita, M. Kwan, S. Rajan, A. J. Hu, and A. J. Hu, "Embedded software verification using symbolic execution and uninterpreted functions," *International Journal of Parallel Programming*, vol. 34, 2006.

[18] The FLINT Group. Yale University. http://flint.cs.yale.edu/.

[19] A. Chlipala, "Modular development of certified program verifiers with a proof assistant," in *International Conference on Functional Programming (ICFP)*. New York, NY, USA: ACM, 2006, pp. 160–171.

[20] J. G. Morrisett, D. Walker, K. Crary, and N. Glew, "From system F to typed assembly language," in *Principles of Programming Languages (POPL)*, 1998, pp. 85–97.

[21] J. Chen, D. Wu, A. W. Appel, and H. Fang, "A provably sound TAL for back-end optimization," in *Programming Language Design and Implementation (PLDI)*. New York, NY, USA: ACM, 2003, pp. 208–219.

[22] C. Flanagan and J. B. Saxe, "Avoiding exponential explosion: generating compact verification conditions," in *Principles of Programming Languages (POPL)*, 2001, pp. 193–205.

[23] K. R. M. Leino, "Efficient weakest preconditions," *Inf. Process. Lett.*, vol. 93, no. 6, pp. 281–288, 2005.

[24] P. V. Homeier and D. F. Martin, "A mechanically verified verification condition generator," *Comput. J.*, vol. 38, no. 2, pp. 131–141, 1995.

[25] Proceedings of the Working Conference on Reverse Engineering. IEEE. 1995–.

[26] J.-C. Filliâtre, "Verification of non-functional programs using interpretations in type theory," *J. Funct. Program.*, vol. 13, no. 4, pp. 709–745, 2003.

[27] S. Katsumata and A. Ohori, "Proof-directed de-compilation of low-level code," in *European Symposium on Programming (ESOP)*. Springer-Verlag, 2001, pp. 352–366.