Decompilation into Logic — Improved

Magnus O. Myreen, Michael J. C. Gordon
Computer Laboratory, University of Cambridge, UK

Konrad Slind
Rockwell Collins, USA

***Abstract*—This paper presents improvements to a technique which aids verification of machine-code programs. This technique, called decompilation into logic, allows the verifier to only deal with tractable extracted models of the machine code rather than the concrete code itself. Our improvements make decompilation simpler, faster and more generally applicable. In particular, the new technique allows the verifier to avoid tedious reasoning directly in the underlying machine-code Hoare logic or the model of the instruction set architecture. The method described in this paper has been implemented in the HOL4 theorem prover.**

I. Introduction

Verification of machine-code programs is hopelessly tedious without good tool support, particularly if verification is to be done against realistic models of commercial machine languages, e.g. x86, ARM, PowerPC, MIPS, whose formal models are several thousand lines long. Done naively, verification efforts fail to scale, get tied to a specific architecture model and may require reading or even annotating machine-code programs.

In previous work [13], we have proposed a technique that can significantly ease verification of machine-code programs, namely: decompilation into logic. Given some concrete machine code and a model of an instruction set architecture (ISA), this decompilation extracts functions (defined in logic) which capture the functional behaviour of the machine code. We have demonstrated that this decompilation technique can be used for post hoc verification, as described below, and also for implementation of proof-producing synthesis tools, as described in [14]. We have shown that these techniques scale to significant examples, including verification of functional correctness of garbage collectors and Lisp implementations in ARM, x86 and PowerPC [10], [11], and decompilation of the seL4 microkernel [12] (12,000 lines of ARM).

This paper's contribution is a presentation of significant technical improvements to our decompilation method [13]. The improvements make the new approach faster, simpler and more widely applicable. In particular, the new technique allows the verifier to avoid reasoning directly in the underlying machine-code Hoare logic, even in the presence of code pointers. The new approach retains all of the features of the previous approach and adds a few new ones (Section II-A). The technique described in this paper has been implemented (www.cl.cam.ac.uk/~mom22/decompilation) in the HOL4 theorem prover [16] and applied to ARM machine code.

This document is an overview of MOD sponsored research and is released to inform projects that include high integrity software or complex electronics. The information contained in this document should not be interpreted as representing the views of the MOD, nor should it be assumed that it reflects any current or future MOD policy. The information cannot supersede any statutory or contractual requirements or liabilities and is offered without prejudice or commitment.

A. Example: Sum of An Array

We start with an example which illustrates what we mean by decompilation and verification via decompilation. Consider the following C code which calculates the sum of an array. (It ignores indexed 0 to make the ARM assembly neater.)

```
do { k += a[i] } while (--i != 0);
```

The C code above can be compiled to ARM assembly:

```
L0: ldr r1,[r2,r3]   ; load mem[r2+r3] into r1
L1: add r0,r1        ; add r1 to r0
L2: subs r3,#4       ; decrement r3 by 4
L3: bne L0           ; goto L0 if r3 ≠ 0
L4:
```

which can be assembled into ARM machine code:

```
E7921003 E0800001 E2533004 1AFFFFFB
```

Decompilation takes concrete machine code as input. From this machine code it extracts a function which describes the behaviour of the code. In this case, sum below which records how registers r0–r3 and memory are affected and also what side condition must hold for correct execution. The side conditions are collected by the *cond* component.

$$
\begin{array}{l}
\mathsf{sum}(cond, r_0, r_1, r_2, r_3, m) = \\
\quad \mathsf{let}\ cond = cond \wedge \mathsf{valid\_address}\ (r_2 + r_3)\ m\ \mathsf{in} \\
\quad \mathsf{let}\ r_1 = m(r_2 + r_3)\ \mathsf{in} \\
\quad \mathsf{let}\ r_0 = r_0 + r_1\ \mathsf{in} \\
\quad \mathsf{let}\ r_3 = r_3 - 4\ \mathsf{in} \\
\qquad \mathsf{if}\ r_3 = 0\ \mathsf{then}\ (cond, r_0, r_1, r_2, r_3, m) \\
\qquad\qquad\quad \mathsf{else}\ \mathsf{sum}(cond, r_0, r_1, r_2, r_3, m)
\end{array}
$$

Decompilation also automatically proves a theorem, which we call a *certificate theorem*, relating the new function to the machine code from which it was extracted. The certificate theorem is derived w.r.t. a model of the ISA of the machine code; in this case, a model of ARM developed by Fox [4]. We state these certificate theorems using a machine-code Hoare triple which is parametrised by the ISA model. The Hoare triples will be explained in later sections, for now read the following certificate theorem for sum informally: for any input $(c, r_0, r_1, r_2, r_3, m)$ which sum relates to output $(c', r_0', r_1', r_2', r_3', m')$, the execution of the ARM machine code can perform the state update corresponding to sum:

$$
\begin{array}{l}
(\mathsf{sum}(c, r_0, r_1, r_2, r_3, m) = (\mathsf{true}, r_0', r_1', r_2', r_3', m')) \Longrightarrow \\
\{\ \text{ARM state holds } (r_0, r_1, r_2, r_3, m)\ \} \\
\texttt{E7921003 E0800001 E2533004 1AFFFFFB} \\
\{\ \text{ARM state holds } (r_0', r_1', r_2', r_3', m')\ \}
\end{array}
$$

The benefit of using decompilation in verification is that once the machine code has been decompiled, subsequent verification can concentrate on only proving properties of the extracted function, since any property proved of the extracted function applies directly to the machine code via the certificate theorem. For example, with an appropriate definition of how arrays are stored in memory, it is easy to prove that sum correctly sums the content of an array and using the certificate theorem relate this property to the machine code.

II. Improvements

As mentioned above, this paper's contribution is to present improvements to the decompilation approach. In particular, we show how decompilation into logic can be made simpler, faster and more generally applicable.

Simpler: The original approach to decompilation was geared towards automating proofs in a Hoare logic that was intended for manual proofs — a complicated Hoare logic that was never optimised for mechanisation performance. In this paper, we show that a much simpler Hoare logic can be used for decompilation. The new Hoare logic (Section III-A) is only a thin layer over the model of the ISA.

Faster: In the new approach, we carefully state the intermediate theorems so that composition of intermediate results can be done in a handful of fast operations. In the previous approach, composition was the main performance bottleneck: often involving simplification through rewriting and calculation of a separation logic 'frame'. The new approach to composition is described in Section III-B. The speed-up we gained can seen in benchmarks listed in Figure 1.

More widely applicable: The main practical drawback of the previous approach was its inability to deal with code involving exotic control flow (e.g. code using more than just goto-like jumps). This lead to an unsatisfactory compromise where certain complicated code had to be verified manually using the machine-code Hoare triples. The new approach is engineered so that it successfully extracts a function even in the presence of code pointers. With the new approach, one can practically always avoid reasoning directly in the underlying Hoare logic. The new approach extracts a single function from the given machine code as before if possible; otherwise, it extracts a function which describes each chunk of well-behaved code. The example below will illustrate what we mean.

A. Example: Calling Every Code Pointer of An Array

To illustrate how the new decompilation approach deals with complicated control-flow, consider the following example program which calls each code pointer stored in an array.

```
do { (a[i])() } while (--i != 0);
```

The C code above can be compiled to ARM assembly:

```
L0: ldr r4,[r5,r6]    ; load mem[r5+r6] into r4
L1: blx r4            ; call code-pointer r4
L2: subs r6,#4        ; decrement r6 by 4
L3: bne L0            ; goto L0 if r6 ≠ 0
L4:
```

Our previous approach to decompilation is not able to process the resulting ARM code, because it is unclear what function describes the effect of the call to the code pointer. In the new approach, we avoid this issue by essentially leaving 'holes' in the extracted function.

The ARM code above decompiles into a function which explicitly mentions the value of the program counter pc. The extracted function has three parts; the first part describes the effect of starting execution from the top of the code ($pc = \mathtt{L0}$): in this case, a load is performed and a call is made to a code pointer, i.e. the pc is updated (with a word-aligned address) and a return address is stored in r_{14}. Note that this function does not make any assumption that the call to the code pointer returns (it might not). The second part of the function describes what happens if execution returns ($pc = \mathtt{L2}$): in this case r_6 is decremented and control moves either to the top or bottom of the code. The third case just states that all other cases are ignored, i.e. no progress is made.

```
calls$(cond, pc, r_4, r_5, r_6, r_{14}, m)$ =
  if $pc = \mathtt{L0}$ then
    let $cond = cond \wedge$ valid_address $(r_5 + r_6)\ m$ in
    let $r_4 = m(r_5 + r_6)$ in
    let $cond = cond \wedge$ word_aligned_address $r_4$ in
    let $(pc, r_{14}) = (r_4, \mathtt{L2})$ in
      $(cond, pc, r_4, r_5, r_6, r_{14}, m)$
  else if $pc = \mathtt{L2}$ then
    let $r_6 = r_6 - 4$ in
      if $r_6 = 0$ then $(cond, \mathtt{L4}, r_4, r_5, r_6, r_{14}, m)$
                 else $(cond, \mathtt{L0}, r_4, r_5, r_6, r_{14}, m)$
  else $(cond, pc, r_4, r_5, r_6, r_{14}, m)$
```

The automatically derived certificate theorem makes use of a feature of our machine-code Hoare triple that allows the pre- and postconditions to mention the value of the program counter (PC) as state component, i.e. control does not need to enter/exit at specific points of the code in the Hoare triple.

$(\mathsf{calls}(c, pc, r_4, r_5, r_6, r_{14}, m) = (\mathsf{true}, pc', r_4', \ldots)) \Longrightarrow$
$\{$ ARM state holds $(r_4, r_5, r_6, r_{14}, m)$ and PC is pc $\}$
`E7954006 E12FFF34 E2566004 1AFFFFFB`
$\{$ ARM state holds $(r_4', r_5', r_6', r_{14}', m')$ and PC is pc' $\}$

With this result from decompilation one can verify properties of this code without tedious proofs in the Hoare logic.

III. Improved Decompilation Algorithm

The new decompilation algorithm has three phases. The key technical differences over our previous approach [13] are highlighted with **bold text**.

Phase 1: Evaluate the underlying ISA model for each machine instruction; derive a theorem, **stated in terms of a simple machine-code Hoare triple**, describing each instruction; and in order to make *phase 3* faster, also **make the code segment of each Hoare triple identical** (explained in Section III-A).

Phase 2: Compute the control-flow graph (CFG) of the given code using information gathered from the Hoare triples.

ARM machine code	instr.	time/cost of old	time/cost of new	reduction	model eval.
sum of array (Sec. I-A)	4	2.5 s (73,039 i)	0.3 s (4,019 i)	86 % (94 %)	7.8 s (1.5 Mi)
copying garbage collector [10]	89	50 s (1,526,281 i)	6.0 s (53,301 i)	88 % (97 %)	173 s (40 Mi)
1024-bit multiword addition	224	70 s (1,029,685 i)	1.2 s (10,802 i)	98 % (99 %)	37 s (8.9 Mi)
256-bit Skein hash function	1,352	5,366 s (21,432,926 i)	56 s (1,842,642 i)	99 % (91 %)	500 s (105 Mi)

Fig. 1. Benchmarks comparing the new approach (new) with our previous approach (old). The cost is given in seconds (s) and number of primitive higher-order logic inferences (i) in HOL4 [16]. The cost of evaluating the ISA model is separated as this cost is independent of decompilation approach.

Split the CFG into separate *decompilation rounds*, i.e. separate inner loops from enclosing outer loops where possible. For complicated CFGs, **insert an extra final decompilation round which ties up the disjoint pieces if necessary** (as illustrated by the example in Section II-A).

Phase 3: For each decompilation round: compose the Hoare triples following the CFG **in a way which directly constructs the extracted function in the postcondition of the theorem** (Section III-B). **This function in the postcondition also collects accumulated side conditions as if they were updates to a state component** (*cond* in Section III-A). If the code has a loop, a loop rule is applied which wraps the result up using a tailrec-combinator and **combines the resulting side condition on termination with the other side conditions**.

A. Simple Machine-Code Hoare Logic

The machine-code Hoare triples, $\{\,pre\,\}\ code\ \{\,post\,\}$, that were used above will be explained next. More formally, these are parametrised by two functions: $next$, a the next-state function for the ISA model of interest; and $assert$, a state assertion which inspects the state (explained below).

$$(assert, next)\ \vdash\ \{\,pre\,\}\,code\,\{\,post\,\} \tag{1}$$

This machine-code Hoare triple is defined to be true: if for all states that satisfy pre and including $code$, then another state can be reached (by some n applications of $next$) such that $post$ is true for this state and $code$ is included in it. The total-correctness Hoare triple (1) is formally defined to mean,

$$\begin{aligned}&\forall state\ c.\ assert\ (code \cup c, pre)\ state \implies\\ &\qquad \exists n.\ assert\ (code \cup c, post)\ (next^n(state))\end{aligned}$$

where the set union $\cup$ with arbitrary code extension c is present to facilitate extension of the code (explain in the next section).

We instantiate $next$ and $assert$ for each supported architecture, e.g. ARM, x86 or PowerPC. We instantiate $assert$ to check that each state component is consistent with $code$ and $pre/post$. Here $code$ is represented as a set of (address,instruction) pairs, and pre and $post$ are, for efficiency reasons, simply a large tuple listing the value of state components, e.g. $(pc, r_0, r_1, \ldots)$ asserts that the value of PC is pc and register 0 is r_0 etc. By representing $pre/post$ as tuples, composition and matching becomes fast and simple. We always include a special $cond$ element in $assert$. This $cond$ is a condition for the entire assertion to make sense, e.g. for ARM we instantiate $assert$ with:

$$\begin{aligned}&\mathsf{arm\_assert}\ (code, pc, r_0, r_1, \ldots, cond)\ state =\\ &\quad (cond \implies code \text{ is in memory of } state \text{ and}\\ &\qquad\quad \text{the PC of } state \text{ is } pc \text{ and } \ldots)\end{aligned}$$

{ARM registers `r1-r3` are (r_1, r_2, r_3) and
 m is a model of part of memory and PC is `L0`}
`E7921003 E0800001 E2533004 1AFFFFFB`
{ARM registers `r1-r3` are $(m(r_2 + r_3), r_2, r_3)$ and
 m is a model of part of memory and PC is `L1` and
 $\mathsf{valid\_address}(r_2 + r_3)\ m$ is added to $cond$}

{ ARM assert $(c, r_0, r_1, r_2, r_3, m)$ and PC is `L1` }
`E7921003 E0800001 E2533004 1AFFFFFB`
{ $\mathsf{let}\ (pc', c', r_0', r_1', r_2', r_3', m') =$
  $(\mathsf{let}\ r_0 = r_0 + r_1\ \mathsf{in}$
   $\mathsf{let}\ r_3 = r_3 - 4\ \mathsf{in}$
    $\mathsf{if}\ r_3 = 0\ \mathsf{then}\ (\mathtt{L4}, c, r_0, r_1, r_2, r_3, m)$
      $\mathsf{else}\ (\mathtt{L0}, c, r_0, r_1, r_2, r_3, m))\ \mathsf{in}$
 ARM assert $(c', r_0', r_1', r_2', r_3', m')$ and PC is pc' }

Fig. 2. Two machine-code Hoare triples for: (a) the load instruction from Section I-A, and (b) the last three ARM instructions from Section I-A. Both contain other code too, explained in Section III-B.

B. Composing Hoare triples

Our machine-code Hoare triple supports composition:

$$\{pre\}\,code\,\{m\} \wedge \{m\}\,code\,\{post\} \implies \{pre\}\,code\,\{post\}$$

For this rule to be applicable, the Hoare triples must have identical code sets $code$. Note that each code set is a set of (address,instruction) pairs which is a *sufficient* assumption for getting execution from pre to $post$. To make the code sets identical, we apply the following theorem which can be used to extend the code sets. This theorem is applied as a pre-processing step in *Phase 1* to speed up composition in *Phase 3*. Here $\subseteq$ is the ordinary subset relation.

$$\{pre\}\,code_1\,\{post\} \wedge code_1 \subseteq code_2 \implies \{pre\}\,code_2\,\{post\}$$

In *Phase 3*, composition of Hoare triples is performed bottom-up following the CFG (or the part of it which is relevant for this decompilation round). Each compositions returns a theorem where the relevant part of the extracted function, including the side conditions, appears in the postcondition. Each composition returns a theorem of the form:

$$\begin{aligned}&\{pre[v_0 \ldots v_n]\}\\ &code\\ &\{\mathsf{let}\ (v_0' \ldots v_n') = f(v_0 \ldots v_n)\ \mathsf{in}\ post[v_0' \ldots v_n']\}\end{aligned} \tag{2}$$

Figure 2 show the concrete inputs to the final composition for the sum-of-an-array example (Section I-A). The second input carries the extracted function in the form of (2).

C. Extracting Recursive Functions

As illustrated by our first example in Section I-A, loops in the machine code turn into loops in the extracted function (if the control-flow is simple enough). We define these tail-recursive functions by instantiating f, g and d in the following function template:

$$\textsf{tailrec}\ g\ f\ d\ x = \textsf{if}\ g\ x\ \textsf{then}\ \textsf{tailrec}\ g\ f\ d\ (f\ x)\ \textsf{else}\ d\ x$$

Our definition of tailrec is based on while $g\ f\ x$ which repeatedly applies f to x until g becomes false. Crucially, while can be defined in (higher-order) logic without a termination proof [7], which is important because most of the functions the decompiler extracts do not terminate for all inputs. However, note that our Hoare triples are total-correctness Hoare triples, i.e. we need to know that our use of while terminates for certain inputs: it terminated for input x if $\neg g\ (f^n\ x)$. For this reason, we insert the termination requirement into the definition of tailrec. We make this requirement part of the $cond$ side condition that our extracted functions produce:

$$\begin{array}{l}\textsf{tailrec}\ g\ f\ d\ x = \\ \quad \textsf{let}\ (cond, v_1 \ldots v_n) = d\ (\textsf{while}\ g\ f\ x)\ \textsf{in} \\ \quad\quad (cond \wedge (\exists n.\ \neg g\ (f^n\ x)), v_1 \ldots v_n)\end{array}$$

Any verification that uses such extracted function must prove that the returned $cond$ is equal to true (for relevant input values); otherwise, the postcondition of the certificate theorem has no meaning (due to $\Longrightarrow$ at the bottom of Section III-A).

To introduce a tail-recursive function, we apply the following theorem with appropriate instantiations of pre, $post$ etc.

$$\begin{array}{l}(\forall x.\ \{pre\ x\}\ code\ \{\textsf{if}\ g\ x\ \textsf{then}\ pre\ (f\ x)\ \textsf{else}\ post\ (d\ x)\} \\ \Longrightarrow \\ (\forall x.\ \{pre\ x\}\ code\ \{post\ (\textsf{tailrec}\ g\ f\ d\ x)\}\}\end{array}$$

Often pre and $post$ are instantiated with functions that simply just set the program counter value. For the example mentioned above, pre is instantiated as follows to set the PC to `L0`.

$$\begin{array}{l}\lambda(c, r_0, r_1, r_2, r_3, m). \\ \quad \text{ARM state holds } (r_0, r_1, r_2, r_3, m, c) \text{ and PC is } \texttt{L0}\end{array}$$

In our implementation, we avoid defining separate component functions f, g and d by defining a single function which returns three different outcomes. We omit the details of this space optimisation as it is not crucial for understanding the main novelties of the new technique: the new Hoare triple; collection of side conditions as if they were state updates; and our new approach to handling complicated control flow.

IV. Summary and Related Work

This paper has presented significant improvements to decompilation into logic — a technique which aids verification of machine-code programs. We have simplified the technique, optimised it for mechanisation speed and made it applicable even to code with arbitrary use of code pointers.

Formal verification of machine code using theorem provers was pioneered in impressive work by Boyer and Yu [2]. Boyer and Yu verified functional correctness of string functions compiled for the Motorola MC68020. Their proofs were carried out in the Boyer-Moore theorem prover Nqthm [6] and required significant manual effort. Since then most work in this area has focused on making proofs easier: Matthews et al. [8] have showed how verification condition generation for machine code can be accomplished, Hardin et al. [5] show how ACL2 can be used and our work [13] has shown how functions can be extracted from machine code and how that aids verification. Various program logics for assembly and machine code have also been developed [10], [15], [1], [3]. Chlipala's approach [3], using Coq, has a distinct emphasis on proof automation for functional correctness. There has also been work targeting mostly automatic proofs of basic safety properties for low-level code [18], [9], [17].

References

[1] Nick Benton. Abstracting allocation: The new new thing. In *Computer Science Logic (CSL)*, Computer Science Logic. Springer, 2006.

[2] Robert S. Boyer and Yuan Yu. Automated proofs of object code for a widely used microprocessor. *J. ACM*, 43(1):166–192, 1996.

[3] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Programming Language Design and Implementation (PLDI)*. ACM, 2011.

[4] Anthony C. J. Fox and Magnus O. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving (ITP)*, LNCS. Springer, 2010.

[5] David S. Hardin, Eric W. Smith, and William 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*, ACL2 '06, pages 11–20, New York, NY, USA, 2006. ACM.

[6] M. Kaufmann, R. S. Boyer, and J Moore. The Boyer-Moore theorem prover and its interactive enhancement. *Computers and Mathematics with Applications*, 29(2):27–62, 1995.

[7] Panagiotis Manolios and J. Strother Moore. Partial functions in ACL2. *J. Autom. Reasoning*, 31(2):107–127, 2003.

[8] John Matthews, J. Strother Moore, Sandip Ray, and Daron Vroon. Verification condition generation via theorem proving. In *Logic Programming and Automated Reasoning (LPAR)*. Springer, 2006.

[9] J. Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *Principles of Programming Languages (POPL)*. ACM, 1998.

[10] Magnus O. Myreen. *Formal verification of machine-code programs*. PhD thesis, University of Cambridge, 2009.

[11] Magnus O. Myreen and Jared Davis. A verified runtime for a verified theorem prover. In Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Interactive Theorem Proving (ITP)*, LNCS. Springer, 2011.

[12] Magnus O. Myreen, Thomas Sewell, Michael Norrish, and Gerwin Klein. Using the Cambridge ARM model to verify the concrete machine code of seL4. Talk at HCSS'11 http://cps-vo.org/node/1127, 2011.

[13] 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.

[14] Magnus O. Myreen, Konrad Slind, and Michael J.C. Gordon. Extensible proof-producing compilation. In de Moor and Schwartzbach, editors, *Compiler Construction (CC)*, LNCS. Springer, 2009.

[15] Zhaozhong Ni and Zhong Shao. Certified assembly programming with embedded code pointers. *ACM SIGPLAN Notices*, 41(1):320–333, January 2006.

[16] Konrad Slind and Michael Norrish. A brief overview of HOL4. In *Theorem Proving in Higher Order Logics (TPHOLs)*. Springer, 2008.

[17] Gang Tan and Andrew W. Appel. A compositional logic for control flow. In E. Allen Emerson and Kedar S. Namjoshi, editors, *Verification, Model Checking and Abstract Interpretation (VMCAI)*. Springer, 2006.

[18] Lu Zhao. *A program logic and its applications to fully verified software fault isolation*. PhD thesis, University of Utah, 2012.