Applied Theorem Proving: Modelling Instruction Sets and Decompiling Machine Code

Anthony Fox
University of Cambridge, Computer Laboratory

Overview

This talk will mainly focus on

1. Specifying instruction set architectures

and briefly look at

2. Formally verifying machine-code programs

Instruction Set Architectures

- There are many architecture families, e.g. Alpha, ARM, MIPS, Motorola 68k, PA-RISC, PowerPC, SPARC, VAX and x86.

- Reference documentation is large, typically thousands of pages in length.

- Pseudo-code is frequently used in reference manuals when describing the semantics of instructions. This can vary from being sketchy (e.g. Intel's x86 pseudo-code is loose and contains English text) to quite rigorous and detailed (e.g. ARM).

- Official *formal* ISA models aren't in the public domain. In-house, simulators are normally written in some HDL (and/or C).

ISA Specification

A couple of uses for formal ISA models are:

- The formalisation can act as a *“golden reference model”*. This could be used in hardware verification and for general validation work.
    - A formal ISA model gives a precise meaning to programs, which can be used in the formal verification of code/compilers.
- At the moment, our group is most interested in the latter use case.

Styles of Formalization

There are number of ways to *formalize* an architecture:

1. As a document containing a “mathematical description”.
   Can be readable, depending on choice of style. However, not especially useful as an artefact.

2. As code in a suitable general purpose programming language, e.g. in a well-defined functional programming language.
   Such languages are not that well known outside of Computer Science. Can result in fast emulation (especially Lisp).

3. As a specification in a theorem prover.
   Prover languages are even less well known, though typically functional in style. ACL2 is a theorem prover and programming language, which provides for fast emulation. Running HOL specifications is very slow in comparison.

4. As a specification in a domain specific language.
   This is the approach that we have now adopted.

L3 — General Features

- ISA specifications were being written directly in HOL4 (approach 3). This had a few downsides.
- A new domain specific language was designed to “make my life easier”.
- Key features are:
  - Looks imperative in style. Provides for a more natural description that is a bit closer to the pseudo-code found in reference manuals.
  - Much easier to write and modify specifications. (In comparison to native HOL4 specifications.)
  - L3 is not a hardware description language (HDL). The language is resolutely focussed on aiding the production of *usable* high-level ISA specifications.
  - Provides export to HOL4.
  - Provides native emulation support. (For initial testing.)
  - Provides export to Standard ML. (For faster emulation.)

L3 — Specific Language Features

- Good support for bit-vector operations.
  This is an essential feature.

- Supports “pattern matching”, including over “bit-patterns”.
  This is very useful when specifying instruction decoding.

- Helps in maintaining an “instruction datatype”.
  Used in specifying instruction decoding, encoding and assembly code parsing.

- Provides support for working with (system) registers that have named sub-fields (bit-ranges).

Under-specification and Non-determinism

- Computer architecture descriptions invariably contain various forms of “looseness”. For example, unpredictable and implementation dependent behaviour, unknown or don’t care values, and non-deterministic choice.

- If there is lots of non-determinism then relational models are needed.

- ISAs are “mostly” deterministic, so we try to get away with writing functional specifications as much as possible.

- Functional models have the advantage of being easy to “run”. The specification corresponds with an “emulator”.

Under-specification and Non-determinism (2)

L3 supports a number of ways to handle under-specification.

- There is a polymorphic value “UNKNOWN”. In HOL4 this corresponds with some fixed but unidentified value. When exporting to Standard ML a fixed choice is made, e.g. always “0”, always “false” and so forth.

- You can use “option types” when you need to detect unknown values. For example, a value can be “Some (expression)” or “None”.

- You can write a specification that incorporates “oracles”. For example, the result is “f(x)” where the function “f” is supplied at runtime.

- One can raise a “model exception”. This corresponds with an exception in Standard ML (emulation will cease and an error will be reported).

L3 models

To date, the following ISAs have been specified in L3:

- ARMv7-AR. Covers all architecture versions from ARMv4.
  No Neon instructions and partial floating-point support. MMU details are not modelled (they are implementation dependent).

- ARM-M0. The “M” architecture variants have a different underlying programmer’s model. The specification includes cycles count information.

- x86-64. Covers a core subset of the instructions and just 64-bit mode.

- MIPS III (with some extensions). Includes a MIPS4k MMU, i.e. a model of a TLB.

Model Validation

- It is important to invest time and effort into validating ISA models.
- In the absence of official “golden reference” models, validation takes the form of running snippets of code in the model and comparing the results against development boards, emulators and PC cores.
- Various people have worked on this:
  - Magnus Myreen for ARM and x86-64.
  - Brian Campbell for ARM-M0.
  - Mike Roe for MIPS.
- Validation provides some assurance but we must still be vigilant when using models.

Machine-code Verification

- Our main area of interest is the verification of low-level code. This includes supporting work on:
  - Compilers and runtime libraries.
    CakeML — University Cambridge and Kent.
  - Operating system micro-kernels.
    Secure Embedded L4 (seL4) — NICTA.
- Other groups at Cambridge are working on related research areas, e.g. adding security features to ISAs and concurrency memory models.

Machine-code Verification (2)

- Magnus Myreen has worked in the area of machine-code verification for a number of years.

- Our main tool is a decompiler. This takes machine-code as input and returns:

  1. The definition of a HOL4 function.

  2. A certificate theorem, which proves that this function's behaviour conforms exactly with running the machine-code.

- This provides a means to "abstract away" the large and complicated ISA model.

Decompiler example

- The following shows a call to the decompiler:

```
val (f1_cert, f1_def) = arm_decompLib.arm_decompile_code "f1"
   `movw r0, #0xFFFF
    lsr  r1, r1, #16
    and  r2, r2, r0, lsl #16
    add  r0, r1, r2`;
```

- The decompiler can accept ARM assembly code as input. This call defines a HOL4 function “f1” as follows:

```
f1 (r1,r2) =
    (let r0 = 65535w in
     let r1 = r1 ⋙ 16 in
     let r2 = r2 && r0 ≪ 16 in
     let r0 = r1 + r2
     in
       (r0,r1,r2))
```

Decompiler example (2)

- Note that the function “f1” makes no reference to the ARM model itself.

- The certificate theorem makes the connection to the ISA model and justifies the definition.

- Similarly, we can decompile “f2” (below right).

f1

```
movw r0, #0xFFFF
lsr  r1, r1, #16
and  r2, r2, r0, lsl #16
add  r0, r1, r2`;
```

f2

```
ror  r2, r2, #16
movw r0, #0xFFFF
and  r2, r2, r0
ror  r2, r2, #16
lsr  r1, r1, #16
eor  r0, r1, r2
```

Decompiler example (3)

- We can now prove that the function “f2” is functionally identical to “f1”. That is, for all r1 and r2

  f1 (r1, r2) = f2 (r1, r2)

- This is a simple proof in HOL4, since we can use the following “tactic”:

  simp [f1_def, f2_def] THEN blastLib.BBLAST_TAC

- This “simplifies” our goal using the function definitions and then applies a “bit-blasting” tactic.

- This shows the two machine-code programs have exactly the same functionality. (They only differ in timing.)

- This example was “simple” because the programs were just sequential blocks of instructions.

- Code with loops/branches can be handled, provided the control flow is not too complex. The verification process can be manually guided.

Some links

- L3 and the ISA models can be downloaded from

  www.cl.cam.ac.uk/~acjf3/l3

- Details about the decompiler can be found at

  www.cl.cam.ac.uk/~mom22/decompilation.html

Questions?