x86 Semantics in Sail

Patrick J. H. Taylor
Gonville and Caius College

UNIVERSITY OF
CAMBRIDGE

A dissertation submitted to the University of Cambridge in partial fulfilment of the requirements for the degree of Computer Science Tripos, Part III

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

June 29, 2020

Declaration

I, Patrick J. H. Taylor of Gonville and Caius College, being a candidate for Computer Science Tripos, Part III, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed: Patrick Taylor

Date: 02.06.2020

Acknowledgements

I would like to thank the following people for their contributions to this project.

My supervisors, Prof. Peter Sewell, Dr Alasdair Armstrong and Dr Robert Norton-Wright, whose suggestions and guidance in the use of Sail made this project possible.

Prof. Peter Sewell, Dr Alasdair Armstrong and my father for reading through multiple revisions of the dissertation and providing such detailed and insightful comments.

Dr Shilpi Goel who kindly spent the time to respond in great detail to my emails enquiring about her work.

Finally, my parents for the wisdom they impart and to my friends, both in college and CUHWC.

Contents

List of Figures

Chapter 1

Introduction

Formal models of instruction set architectures (ISAs) are becoming more prevalent in industry and research. One model, developed by Shilpi Goel [1], uses the interactive theorem prover ACL2 and is written in a subset of Lisp. It models the behaviour and decoding of a large number of instructions from the x86 architecture and provides a base for simulation and formal verification. In this project I developed a program to translate this ACL2 model into Sail, a language designed by Armstrong et al. [2] to model ISA semantics. The contributions are as follows:

- A translation into Sail of all instructions implemented in the original model (excluding floating-point, some system level and some non-deterministic instructions) and the relevant decoding and dispatch routines.
- A program to perform this translation and thus handle future development in ACL2.
- Validation of the translated model against the original model to give confidence in its correctness.
- The discovery of an error in the original model.
- Assessment of the coverage provided by a previously-published test

suite using the translated model and Sail's coverage tools.

- Details of techniques used when performing source-to-source compilation with Lisp as the input language.

The rest of this dissertation is structured as follows. Chapter 2 provides the necessary background in the x86-64 ISA, ISA models, Lisp, ACL2 and Sail. Chapter 3 describes the implementation of the translator, focussing on interesting problems that were encountered. Chapter 4 details the methodology in validating the translated model and assessing the test suite and presents and discusses the results. Chapter 5 describes other work and relates it to this project. Finally, Chapter 6 concludes.

For clarity:

- The original ACL2 model can be found at: `http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/index-seo.php/X86ISA___INTRODUCTION`
- My Python translator code and the Sail code it generates are attached in the submission.

Chapter 2

Background

This chapter covers the relevant background material. It starts with an overview of ISA models; it then describes the purpose of ACL2, the syntax of Lisp and the ACL2 x86 model, and gives a brief description of some important parts of the x86 ISA. Finally it presents an overview of Sail and the reasons for wanting to translate the ACL2 model into Sail.

2.1 ISAs and ISA Models

Instruction Set Architectures form the link between hardware and software. They define how a processor behaves by specifying the number and type of registers, the format of binary instructions, and how executing each instruction affects the state of the processor and memory. They do not specify the exact hardware implementation, so *microarchitectural* features (such as cache sizes, pipeline depth, superscalar degree and other optimisations) can be tuned to meet performance goals such as performance, price and energy consumption. ISAs are thus the standard against which processor implementations are verified and the means by which programmers (or rather, compiler writers) know how their software will behave.

However, current specifications for instruction set architectures exhibit a

number of problems. ISA specifications are large: the specification for *Intel 64*, for example, is over 5000 pages long [3]. They are also written predominantly in natural language, where the effects of each instruction on the processor state and memory are described in prose. This results in difficulties for all parties who use such specifications: development of the spec is hard because the effects of changes are unclear; it is difficult to ensure that hardware implementations are compliant; it is difficult to ensure that machine code programs will behave as expected; and their use relies on *a posteriori* knowledge derived, not from the spec, but from experience and observation of previous implementations.

Thus, formal models have started to make an appearance as they significantly reduce the problems above. One such effort is Arm's Architecture Specification Language (ASL). Although originally an informal pseudocode-like language, an initiative in Arm, with Alastair Reid as a prominent figure, turned it into a specified executable language [4]. Formal models allow design processes and complexity to be managed by running regression tests to gain confidence an implementation matches the specification or proving properties [5] about the specification. Section 5.1 describes related work on translating ASL into Sail and explains why translating ACL2 into Sail is a significantly harder process.

2.2 ACL2 and LISP

'A Computational Logic for Applicative Common Lisp' (ACL2) comprises a logic, in which users can create models of digital systems, and an interactive theorem prover to prove properties about such models [6]. The logic in question is an *applicative* subset of Common Lisp—applicative meaning that basic syntax elements (such as lists, arithmetic and conditional operations) are formalised whilst non-functional features (such as objects and circular structures) are not [7]. Whilst the logical side of ACL2 formalises Lisp to allow theorem proving, the use of Lisp itself allows the execution of models

as well.

ACL2 is used in industry. Hunt et al. describe [8] some notable early projects including the modelling and verifying of a digital signal processor at Motorola and the verification of floating-point division pseudocode at AMD. Other companies such as IBM and Rockwell Collins also started using ACL2. The most relevant industrial use is at Centaur Technology, where ACL2 is used throughout the extensive formal verification efforts of Goel et al. [9]. The x86 ISA model, discussed in the next section, is used as their reference model.

2.3 The ACL2 x86 Model

The ACL2 x86 ISA model [10] makes use of both logical reasoning and execution: one can prove properties about the model symbolically and formally verify programs against a specification; or, given a memory image including a sequence of x86 instructions and a starting state for the registers, one can emulate its behaviour. Examples can be found in Goel's original thesis on the model [1]. The ability to execute instructions is not only useful for simulating programs, but vital for validating the model is correct in the first place. Without this, reliance would be needed on the human ability to translate correctly the specifications into the Lisp model—a situation antithetical to the original purpose of the model. Validation of the Sail translation against the ACL2 model is explored in Chapter 4.

The ACL2 implementation models the instruction semantics for many instructions including 218/256 one-byte opcodes and 113/256 two-byte opcodes.

It also models instruction decoding. This is worth mentioning because x86 decoding is very complex. Another notable x86 model, developed in the K Framework by Dasgupta et al., does not model decoding at all [11, Introduction footnote], instead opting to call semantic functions based on the assembly code rather than the compiled binary. Encoded x86 binary instructions may include, amongst others, the following parts:

- 0-4 bytes of optional *legacy* prefixes which affect the instruction behaviour (for example, repeating the instruction a certain number of times). At most one prefix from each of four groups actually affects the behaviour, although more than one from each group may be present in the instruction—in this case, the behaviour is undefined but, empirically, Goel notes, only the last prefix in each group seems to take effect.

- An optional *REX* prefix byte for certain instructions.

- 1-3 bytes of instruction opcode.

- Other instruction-dependant or optional bytes such as *modR/M*, *SIB* and operands, each of which has its own complex semantics.

As well as modelling instruction semantics and decoding, the model implements both *application* and *system* level *views*, broadly corresponding to x86 protection rings, but with added utilities in the application-level view to provide functionality expected of an operating system.

It is worth presenting a call trace for an example instruction run under the ACL2 model. Figure 2.1 shows the main functions involved in executing the instruction `add $0x1,%eax` (incrementing the value in register `eax`). Execution begins at the top-level step function where some decoding occurs. Once the opcode is found, control is passed, via the one-byte dispatch table, to the *operation specification function* which performs more decoding and finds the operand values, and then to the *instruction specification function* which performs the calculation on the operands, before the results are written back to memory and registers. The boxes on the right show that this is just one possible execution trace: not only do other instructions exist in the one-byte opcode table, but whole other dispatch tables exist.

Furthermore, functionality such as finding the prefixes of an instruction is difficult in itself and, although only mentioned in the diagram, has a whole call trace unto itself. This particular example (finding the prefixes) is difficult for two reasons:

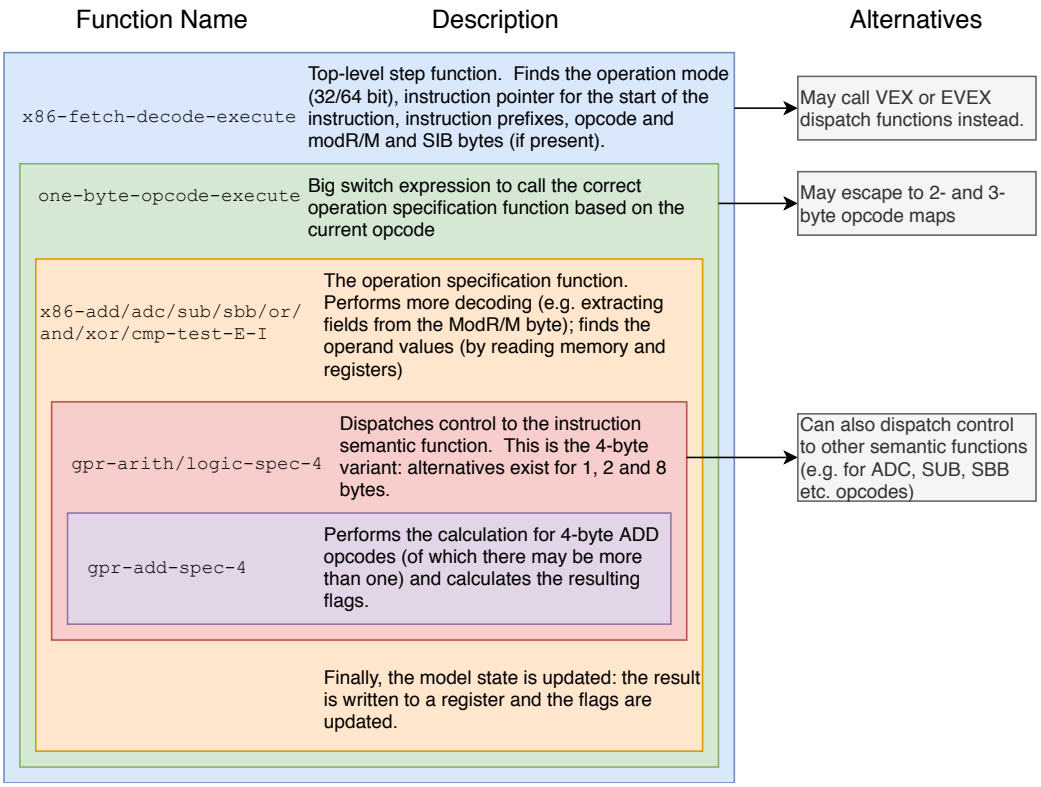

Figure 2.1: Example Call Trace in the ACL2 Model for Instruction `add $0x1, %eax`

1. Finding the prefixes is a non-trivial task: some complications were explained above.

2. Even just reading from memory is a complex task. For example, multiple levels of address calculation must be performed to go from the *effective* address, through *linear* and *logical* address before the final *physical* address is found.

The model itself is spread across 80 files, of which 41 represent instruction semantic functions and the rest are the surrounding framework for handling memory, decoding, dispatch and state representation. This totals over 40 000 lines of code, excluding blank lines and comments [1, Sec. 3.2]

2.4 Sail

Sail is a first-order imperative programming language designed by Armstrong et al. [2] specifically for modelling ISAs. Registers and memory are represented by global state and reads and writes are tracked through the Sail specification with an effects system. It is statically and strongly typed with a light-weight dependent type system, allowing one easily to write functions which can operate on different data widths.

Like the ACL2 and K Framework models, Sail supports both reasoning and execution however, instead of doing so internally, it generates code for well-known existing systems. It targets OCaml and C for generating executable emulators and Isabelle/HOL, HOL4, Coq and SMT for logical reasoning.

Sail is also designed to be simple and similar to pseudocode languages used in existing specifications to provide familiarity to engineers, as opposed to ACL2 which has a high entry barrier and a steep learning curve.

2.5 Conclusion—Background

In this chapter we have: noted that ISA models are becoming increasingly important; seen two languages for ISA specification; introduced the ACL2 x86 ISA model; and made the case for translating it into Sail.

Further specific details of Lisp, ACL2, the ACL2 model and the x86 architecture are introduced as they are needed throughout this dissertation.

Chapter 3

Method

This section first describes the method used to translate Lisp to Sail, including an extended example, before going into detail on particular points of interest. Specifically, we first look at how we can extract type information from Lisp code and fill in the gaps where information is missing. Next we examine the problems encountered when trying to convert certain pieces of Lisp syntax which may map to more than one piece of Sail syntax and how to disambiguate the available options. Finally, a brief description of macro handling is given.

3.1 Overview

In a general compiler pipeline, the first stages are lexing a string of characters into tokens before parsing a list of these tokens into an Abstract Syntax Tree (AST). Fortunately, Lisp's use of prefix (Polish) notation makes it simple to both lex and parse. For the reader unfamiliar with Lisp, the key feature is that code is represented by lists which are always delimited by brackets. For instance, the arithmetic expression

$$1 + (2 \times 3)$$

can be represented as the following Lisp expression:

```
(+ 1 (* 2 3))
```

Note that the functions being applied (addition and multiplication) are the first elements in their respective lists and the arguments to which they are applied come after.

Once a Lisp AST has been constructed, it must be translated into a Sail AST. This is hard, partly due to the complexity of higher-level Lisp forms, and partly because of the desirable properties for the translator. Specifically, it was decided that the translator ought to satisfy the following properties:

- It should be predominantly automatic—it should be able to translate Lisp code to Sail with little manual intervention.
- It should produce idiomatic Sail code.

In reality there is a trade-off between the two. At one extreme a completely manual translation would produce highly readable Sail but any change in the ACL2 must also be tracked manually. At the other end, it would likely be possible to implement a Lisp interpreter within Sail using the lowest level Lisp forms possible (it is possible to construct a Lisp with very few primitives [12]) but this would be far less readable. The more realistic, middle-ground, design decisions are which high level functions to translate manually and which to leave to the translator. For example, the `rotate-left` function, in ACL2, splits its integer input into two by performing appropriate bitwise mask operations, shifts the two parts into their new positions before `OR`ing them together. It would be eminently possible to translate these operations into Sail, but a more idiomatic approach would be as follows:

```
xl = get_slice_int(placesMod, x, 0);
xh = get_slice_int(width, x, placesMod);
unsigned(xl @ xh)
```

The `get_slice_int` functions split the input integer, `x`, into high and low parts, `xh` and `xl` respectively, before simply concatenating the results using

`xl @ xh`. This is far more concise and readable but the implementation required manual inspection. For this function such effort can be justified as it is a low-level utility function which is unlikely to change.

The previous example was contrived: realistic translation contains far more subtlety. Examples of why this is the case are detailed in the following sections; however to gain an intuition of the overarching translation technique, an extended example is presented.

3.2 Extended Example

This section examines the function in Figure 3.1 which is written in ACL2 and which is part of the x86 model. Its features are explained as needed, but at a high level, the function counts the number of set bits in an 8-bit unsigned integer. Further details about many topics introduced in this section, such as macros and typing, are covered in more detail later.

We proceed by examining each token in the code in turn and explaining how it is handled by the translator.

`define`

As mentioned earlier, the first token in a list is the function (or macro) name; the remainder of the list is data it consumes. Thus, `define` tells us how to handle the rest of the example, in this case, defining a function. The question is, how does the translator know what to do with `define`? It is convenient to separate into categories the types of symbol the translator might encounter:

- Handwritten functions, such as `rotate-left` above. When the translator encounters `rotate-left`, the resulting Sail code is simply a call to the handwritten Sail function `rotate_left`.
- Previously translated functions. A function which has already been translated can be translated as a call to that translated function in Sail.

```
(define bitcount8
 ((x :type (unsigned-byte 8)))
 :measure (integer-length x)
 :inline t
 :no-function t
 :verify-guards nil
 :enabled t
 (if (zp x)
    0
  (+ (the (unsigned-byte 1)
      (mbe :logic
          (loghead 1 x)
          :exec (logand 1 x)))
     (the (integer 0 8)
      (bitcount8
       (the (unsigned-byte 8)
        (mbe :logic
            (logtail 1 x)
            :exec (ash x -1)))))))
```

Figure 3.1: Example of an ACL2 Function

For instance here, `bitcount8` is used within the `bitcount8` function (making it recursive).

- Special tokens. It is expedient to handle certain keywords manually. It is into this category that `define` falls. Many such keywords are implemented in Lisp with more primitive constructs (`define`, for instance, is actually a macro) but we choose to handle them manually for convenience. Others are primitives for which we have no choice but to handle them manually (e.g. the token `if`).

Information about handwritten and previously encountered functions, and special tokens is carried around in an *environment* which is threaded through the translation code.

```
bitcount8 ((x :type (unsigned-byte 8)))
```

The special token translation for the `define` keyword specifies that the next

two Lisp *forms* are the function name and formal parameters respectively. The fact that a function called `bitcount8` exists with a bound variable `x` is thus registered with the environment so that they are available for use when the function body is translated. Note that the code which translates a form beginning `define` must handle the list starting with `x` appropriately: the `x` is not a function call but the name of the formal parameter. We also see an example of gaining type information: `:type` introduces the declaration that `x` is an unsigned number at most eight bits long (not eight bytes long, confusingly).

```
:measure (integer-length x)
...
:enabled t
```

These *keyword* parameters are used by `define` and provide instructions to ACL2—in this case they can all be ignored as they do not contribute to the translation.

```
(if ...)
```

The final form in a `define` is known to be the function body. We now encounter, again, a list representing some code and so can call the translator function recursively on it. It finds that, as mentioned above, the handling of `if` is defined as a special token, and that its structure is `(if <predicate> <then-branch> <else-branch>)`.

```
(zp x)
```

The first form within `if` is the conditional predicate. Once again, the translator is called recursively and finds that zp translates to a function which tests for zero.

```
0
```

The second form is the `then` branch. Here the translator must be able to handle number literals. In general, it must also handle other types of literal as well as some macros which translate as literals. Fortunately, these are syntactically disambiguated.

```
(+ (the (unsigned-byte 1)
       (mbe :logic
           (loghead 1 x)
           :exec (logand 1 x)))
     (the (integer 0 8)
      (bitcount8
       (the (unsigned-byte 8)
         (mbe :logic
             (logtail 1 x)
             :exec (ash x -1))))))
```

Again, the translator is called recursively on the subsequent nested forms in the `else` branch. The tokens `+`, `the`, `unsigned-byte`, `loghead`, `integer` and `logtail` are translated as special tokens or handwritten functions as explained above. There are three items which do deserve comment:

1. `mbe` is used in the ACL2 code to allow the use of different forms depending on whether the model is being used for logical reasoning or execution (the two forms must be proved to be equivalent). No such distinction is made in Sail, and so we can pick just one 'branch' to translate—here, the calls to `logand` and `ash` may be ignored because the `:logic` branches are chosen.

2. The call to `bitcount8` is recursive. This requires that, when translating `define`, the presence of the `bitcount8` function is registered with the environment before the translation of its body. It can also complicate typing, although here the type can be resolved by the base case in the other branch of the `if`.

```
val bitcount8 : (int) -> int effect {escape}
function bitcount8 (x) =
  if (x) == (0)
  then 0
  else (the_range(0, 1, loghead(1, x))) +
      (the_range(0, 8, bitcount8(
        the_range(0, 255, logtail(1, x)))))
```

Figure 3.2: Function Shown in Figure 3.1 Translated into Sail

3. The use of `x`, the formal parameter to the function also requires its presence to be registered with the environment in a list of currently bound variables.

The translation of the `bitcount8` into Sail is shown in Figure 3.2.

3.3 Translator Structure

We now describe the structure of the translator from a high level—Figure 3.3 shows the control flow with the most common paths highlighted in blue and recursion indicated by dotted lines. Mostly, after a Lisp file has been lexed and parsed, the translator recursively translates code lists (either as special tokens, handwritten functions or functions automatically translated already), as demonstrated in Section 3.2. Sometimes manual intervention is required to handle very specific cases and, of course, there are base cases such as number literals and strings. Other patterns (such as how to represent consecutive function definitions) do exist but are uninteresting and elided from the diagram.

The translator itself (excluding test harnesses) is implemented across 11 Python files totalling over 7000 lines. The largest file implements the special token translation.

Translations of 69 special tokens have been implemented. Some of these are one-line functions (such as the translation of `(in-package ...)`), but most

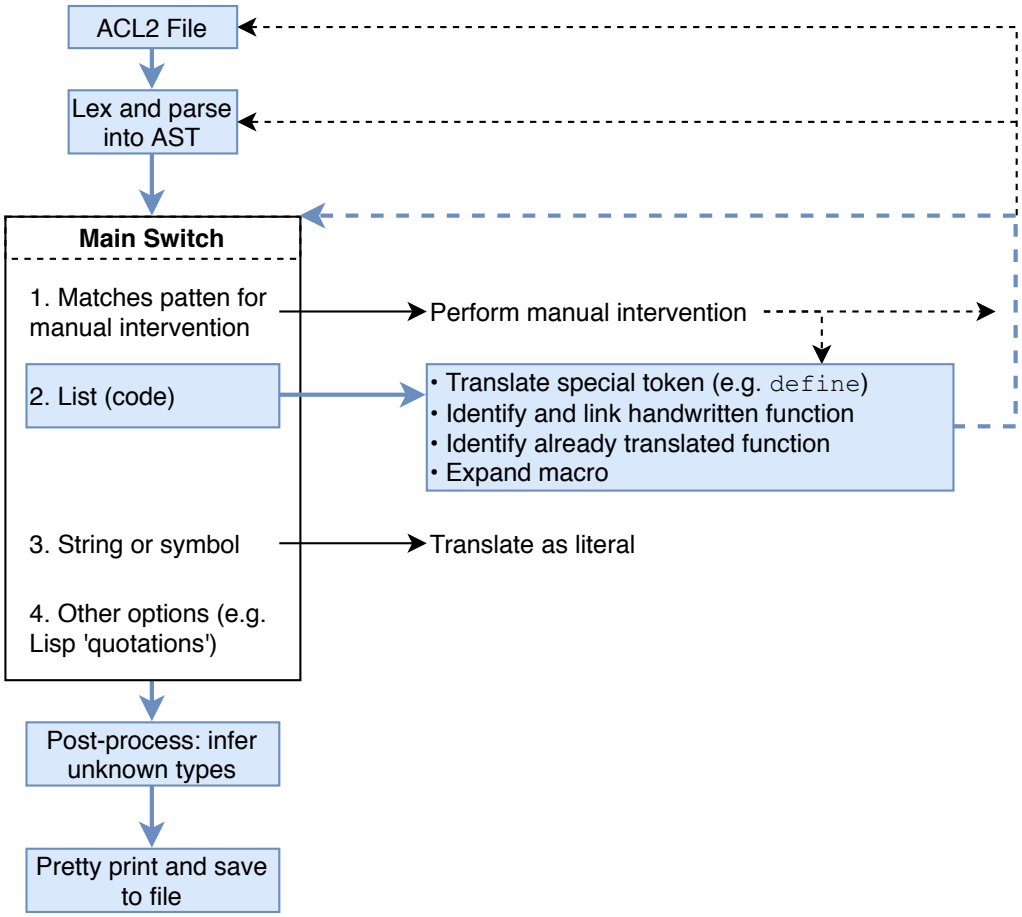

Figure 3.3: Translator Structure

are more lengthy, and some are very complex. Complicated functions include the translations of: function definitions (`(define <args> <body>)`), which require careful handling of bound variables, typing, keywords and recursion; and sophisticated local bindings such as the `b*` form.

Furthermore, there are 74 support functions handwritten directly in Sail to manage register and memory access and basic bitwise operations.

There are also 15 more specific manual interventions which capture edge cases. Creating the patterns on which to select the correct AST elements was sometimes a complex and involved task. It was also necessary to support both pre- and post-translation interventions, adding to the complexity.

These figures give an idea of the scale of the project and show it to be a large undertaking.

Overall, the translator is a recursive function which translates lists based on the first symbol, taking different actions depending on the category into which that symbol falls.

3.4 Typing In General

The extended example in Section 3.2 demonstrated how the translator recursively translates Lisp forms based on manually defined patterns or functions it has already encountered. Although careful implementation is required when translating terms, things become even harder when considering types. This section details some of the problems encountered and their solutions.

There are three issues that make handling types difficult:

1. Lisp is dynamically typed but Sail is statically typed.
2. ACL2 has a rich type system causing unexpected problems when translating Lisp types.
3. Some literal Lisp terms translate to different ACL2 literals depending on their context.

This section covers the first two issues by explaining basic type inference and type approximations before describing a richer type inference algorithm. Section 3.5 covers the third issue.

3.4.1 Basic Type Inference

In Sail there is a syntactic obligation to specify types of functions in the source code. For example, in the simple Sail function below, the specification that `is_nat` is a function from `int -> bool` is mandatory.

```
val is_nat : int -> bool
function is_nat (x) =
  x >= 0
```

Lisp, on the other hand, is dynamically typed and there is no syntactic obligation to provide type annotations. However, in order to translate Lisp code, static type inference must be performed. In particular, it is necessary, for each function, to infer a type for each formal parameter and to infer a return type.

The latter problem is relatively simple: we use unsurprising type inference rules associated with AST elements to infer the return type of each function. When inconsistencies arise typing fails and an error is produced, for instance, when the branches of a conditional expression have different types. Usually typing fails in this way when a heuristic fails to 'guess' the correct type in the presence of ambiguity. More information about such heuristics and why we need them is given in Section 3.5.2.

The remainder of this section (and indeed Sections 3.4.2 and 3.4.3) covers the harder problem of inferring the types of formal parameters to functions.

To begin, we note that, fortunately, there are often hints within the ACL2 code. We saw this in the extended example when the `:type` keyword was used to infer the type of a variable. There are two general ways in which type information can be specified in ACL2:

1. **Guards**. Guards are the mechanism most akin to types. In raw Lisp, a function of n formal parameters can be applied to n actual parameters of any type [13]. Guards restrict the domain of functions so that they only accept arguments which satisfy a given formula [14]. For instance, the following snippet indicates that the argument `result-nbits` is a natural number and `signed-raw-result` is an integer.

   ```
   (define general-of-spec-fn (result-nbits signed-raw-result)
    :guard (and (natp result-nbits)
                (integerp signed-raw-result))
    ...
   ```

   Guards are used both logically—to reason about functions—and in execution—to allow the underlying Lisp implementation to use functions optimised for specific datatypes rather than having to handle any type.

2. **Type Specifiers** [15]. Type specifiers (type-specs) are more ubiquitous, both in their prevalence and in the variety of forms in which they can appear. For instance, we can give extra information to function definitions using the form `(declare <type-spec> v1 ... vn)` [16] or use a similar technique inlined into the code using `(the <type-spec> <expr>)` [17]. Type specifiers are primarily used to improve execution speed but are also added to the *guard obligations* for formal reasoning. As a concrete example, we saw earlier the following form:

   ```
   x :type (unsigned-byte 8)
   ```

   The type-spec `unsigned-byte 8` tells us that `x` is an eight bit number (not an eight byte number).

The above examples already hint at the large number of ways in which types may be specified in ACL2, and it is from this variety that complexity arises. Each way of specifying types must be caught (there are at least five different ways: guards; using `:type`; using `declare`; inline `the` expressions and `the` expressions as part of a `b*` form) [14, 18, 16, 17, 19]. Furthermore, different

forms suggest different translations: `declare` and `the` forms are two contrasting examples. `declare` forms give us information about the type of a function and so influence the top-level `val` declaration in Sail whereas `the` forms are best translated inline. An example is presented in the following code snippet.

```
val the_nat : int -> nat effect {escape}
function the_nat (i) = {
      assert (i >= 0);
      i
}
```

Here we see that Sail incorporates flow-sensitive typing. The `assert` statement provides typing information to the next line: specifically, that `i >= 0` and hence can be given the type `nat`. If, at runtime, `i < 0` then the `assert` throws an exception (this requires the `escape` effect to be declared as described in Section 3.4.4). In this way, translation is made easier but some static type-safety is lost: Sail was designed to allow this trade-off when implementing the ASL to Sail translator [2].

In capturing useful instances of type information we must also be able to translate the guards and type-specs themselves. Type-specs are relatively straightforward in that their syntax contains a small number of oft-used constructs. Guards are more complex in that they can, and do, use arbitrary formulae. They are also less prevalent than type-specs. Overall, we have the same problem of what to translate manually or what to ignore.

Finally, is it worth noting that we cannot extract all the necessary type information directly. For instance, consider this following snippet from the start of a function definition:

```
(define x86-effective-addr
   ((proc-mode
      :type (integer 0 #.*num-proc-modes-1*))
    p4
    (temp-RIP
      :type (signed-byte #.*max-linear-address-size*))
  ...
```

Here, the types of the formal parameters `proc-mode` and `temp-rip` are specified using type-specs, but the parameter `p4` has no type information. The solution to this problem is described in Section 3.4.3.

3.4.2 Type Approximations

In this section we first see an ACL2 type-spec which suggests a certain translation to Sail. We then see why this translation actually causes problems and how to mitigate them.

Consider the following ACL2 type-spec applied to an expression, `x`:

```
(the (integer i j) x)
```

Unsurprisingly, this specifies that `i <= x` and `x <= j`. It is thus very tempting to assign `x` a `range` type in Sail, specifically, `x : range(i, j)`.

Now consider passing `x` into a basic function which returns the first 64 bits of a number as a natural:

```
(n64 (the (integer i j) x))
```

A very tempting type signature for the translation of `n64` in Sail is:

```
val n64 : forall 'n 'm.
       (range('n, 'm)) -> range(0, 2^64 - 1)
```

The assumption is that a range type being passed in, say `range(i, j)`, will have `i` and `j` unified with `'n` and `'m`. This assumption is wrong.

This is because the type signature shown above for `n64` is internally equivalent to:

```
val n64 : forall 'n 'm 'c,
       'n <= 'c <= 'm.
       int('c) -> range(0, 2^64 - 1)
```

Neither `'n` nor `'m` appear in the function argument or return type, and so are unconstrained. Effectively, they could be picked arbitrarily, reducing the type to `val n64 :  int -> range(0, 2^64 - 1)`. In reality the type checker simply fails as it is unable to instantiate the function type.

For this reason, it is sensible to assume that all numeric argument types are integers (the supertype for all numeric types in Sail [20]). In this case, the type of `n64` would become:

```
val n64 : int -> int
```

It is reasonable to ask why the return type is now `int` as well. In general, it is not immediately obvious how to infer the most constrained return type of functions and considering all numeric return types as integers was the expedient solution. There is the possibility for future work of performing a whole AST transformation after translation has occurred to examine how functions are called and attempt to tighten up the top-level types in Sail.

Likewise, it would also be possible to consider all numeric types as the `bits('n)` Sail type. Here we would both be obliged to specify the bit width and would be further from the ACL2 handling of numerics.

Overall, practical type checking is performed at run time using `assert` statements in the translation of `the` expressions as demonstrated in Section 3.4.1. This is more in line with the programming style of the ACL2 model.

3.4.3 Richer Type Inference

We saw in Section 3.4.1 that we can often infer the types of formal parameters to functions from annotations in the ACL2 syntax, but that occasionally the

necessary information is missing or hard to extract. We want to be able to reconstruct this data in order to be able to generate the mandatory type specifications for Sail. This section first shows some examples and then describes the algorithm developed to do this.

The example where the type of `p4` is not specified is repeated as follows:

```
(define x86-effective-addr
   ((proc-mode :type (<type-spec>*))
    p4
    (temp-RIP :type (<type-spec))
      ...
```

Section 3.4.2 showed that any numeric type in ACL2 can be approximated as an integer in Sail, however typing all functions with integers is not a universal solution: bools, strings and option types also exist.

One way of gaining information about `p4` is to see how `x86-effective-addr` is used elsewhere in the codebase—specifically, what type is passed in for the actual parameter of `p4`. For example, in one of the semantic functions for a MOV instruction, this function is used as follows:

```
(x86-effective-addr proc-mode p4? temp-rip
               rex-byte r/m mod sib 0 x86)
```

Earlier in the same function we find that `p4?` was calculated using an equality test:

```
(b*
      ((p4? (equal
       #.*addr-size-override*
       (prefixes->adr prefixes))))
      ...)
```

The function `equal` (and other equality tests) are functions which are translated manually. Specifically, we translate `equal` to the Sail function `==` which has a return type of `bool`. From here we can work our way backwards: `p4?`

`: bool` means that the `p4` formal parameter of `x86-effective-addr` must also have type `bool`.

We could also have gone in the other direction: examining where `p4` is used **within** `x86-effective-addr`. In this case, it is only used as the conditional of `if` expressions. This agrees with the above conclusion that `p4` is a boolean. Do note, however, that we cannot infer a boolean type from use in the conditional of an `if`. This may seem counter-intuitive and it is, in fact, symptomatic of a larger problem described in Section 3.5.

The question now becomes how to generalise this technique. There are two key insights which help us do this.

Firstly, this problem relates solely to instances of bound variables whose type is unknown. Although other type ambiguities do arise, as detailed in 3.5, this problem is decoupled. These bound variables are generated in a single place: the definition of functions. The other piece of Sail syntax which generates variables is the `let` binding, but any variable of unknown type generated here will only inherit an unknown type from a previously bound variable. We can think of all functions in the translation being built up from primitives: literals, handwritten functions (which have well-specified types) and bound variables. Thus, resolving the type of a bound variable will have effects elsewhere in the code as the affect percolates upwards.

The second insight is that, instead of thinking about working up or down the AST as in the examples above, we can resolve the type of unknown bound variables using applications of functions to arguments. For each formal/actual parameter pair we may be able to unify the type of one to the type of the other. So, the application

$$(\lambda \text{<formal>}.\text{<expr>})\text{<actual>}$$

gives four situations:

1. Formal type unknown, actual type unknown—unification is impossible.
2. Formal type known, actual type unknown—actual type becomes formal

type. E.g.

$$(\lambda(x : \text{nat}).e)(x : ???)$$

This case is similar to how type inference algorithms such as Hindley-Milner infer types in the current environment.

3. Formal type unknown, actual type known—formal type becomes actual type. E.g.

$$(\lambda(x : ???).e)(2 : \text{nat})$$

This case is not used in type inference algorithms where functions may be polymorphic, but is acceptable here as we assume monomorphism.

4. Formal type known, actual type known—unification unnecessary.

Note that resolving at applications covers both of the situations in the example above. When we examine what is passed in as the actual of `p4` we are working up the AST: this corresponds to an unknown formal but a known actual. When we look at where `p4` is used within the function, we proceed downwards, further into the AST. We hope to find an instance where `p4` is used as an actual parameter to a function where the corresponding formal has a known type.

This leads us to a fixed point algorithm where we first collect instances of function applications then iteratively resolve instances of bound variables until no changes occur. At each iteration, the number of unresolved bound variables must either decrease or stay the same. Hopefully, by the end, all variables of unknown type will be resolved.

There are a few points to note about the implementation:

- The algorithm required a mechanism to collect instances of function applications. In reality, a more general method was implemented to walk an AST and return instances of AST elements satisfying a certain predicate.
- Taking an object-oriented approach to representing the Sail AST helped with the resolution process. When a function is translated it creates

an object for each of its formal parameters—a bound variable. When a variable is used, the same object is referenced. This removes the need to have a look up table of bound variables and their corresponding types for each function.

- Originally the analysis was performed on a per-file level. Unfortunately, it was necessary to expand this to a global AST analysis to be able to resolve all instances of unknown types. This is, of course, at the expense of having to translate and store the whole AST before any files are written out. An alternative would have been to manually intervene when such cases arose, reducing automation.

- Once the type of a bound variable has been resolved, its type percolates elsewhere. The order in which bound variables are resolved will affect performance, however the best order will depend on the AST in question. For example, a top-level function may have an unknown formal but a known actual, the resolution of which allows the resolution of a function contained therein, and so on for an arbitrary depth. Here, resolving the top most function first and working down the AST is optimal and results in a time linear in the depth. Working upwards, however, we would only hit the first resolution having been through all applications, resulting in an overall quadratic time. On the other hand, working upwards would be optimal if the item that resolves the chain is located at the bottom where an actual is unknown but a formal is known.

The overall process can be described in terms of typing rules. Classically, a typing rule may infer the type of an application from the type of the function and the actual parameter to which it is applied:

$$\frac{f : \tau \to \tau' \qquad x : \tau}{fx : \tau'}(app)$$

In the algorithm presented above, we invert the typing rule in two ways:

$$\frac{fx : \tau' \qquad f : \tau \to \tau'}{x : \tau}(\textit{infer-app1}) \qquad \frac{fx : \tau' \qquad x : \tau}{f : \tau \to \tau'}(\textit{infer-app2})$$

As a final note, the implementation assumes that functions are not polymorphic: the type of each function must be able to be directly represented in terms of concrete types. This is a reasonable assumption to make in the context of ISA specifications as there is little generic code.

3.4.4 Effects and State

Another mandatory piece of Sail syntax is the specification of effects. Effects include reading and writing registers and memory and throwing exceptions. The effects of each handwritten function must be carefully specified. This allows for generated functions to calculate their effects: the union of the effects of functions they call.

It is also worth mentioning the difference in state representation between ACL2 and Sail. ACL2 uses the typically functional style of threading the state through the whole model in the form of an `x86` parameter for most functions. Sail specifies registers as global state and interacts with an external memory simulator. Fortunately, there are a very small number of ACL2 functions which access state—few enough that their types, effects and functionality can all be translated manually.

3.4.5 Conclusion—Typing

In this section we have seen why the translator needs to infer function types from the ACL2 model. It does this mostly from type annotations in the ACL2 code but analyses the translated AST to fill in the gaps and to infer return types. We also saw approximations that were made to help interface with the Sail type system.

The next section goes into detail on some terms which have ambiguous translation and how such cases are handled.

3.5 Translating `nil`

We have seen that the top-level translation function switches on the first item of a list which represents code—the first item is the function being applied—and recursively translates sub-expressions as necessary. For the most part, translation is deterministic: there is exactly one course of action to take for each function symbol.

Unfortunately, this is not universally true. This section details one particular and very prevalent exception, explains the ways in which we handle it and concludes with some brief examples of other anomalies.

3.5.1 The Problem with `nil`

The most common ambiguous Lisp expression is simply `nil`. It is tempting to think we can translate `nil` simply to the boolean value `false`. This is not the case. Unlike in ClojureScript where Sierram and VanderHart note that it is possible to translate Lisp's `nil` to JavaScript's `null` [21, ch. 5, p. 42], there is no single value to which we can map `nil`.

Instead, `nil` is a *falsey* value, and as Staples notes [22], *can* represent `false` but, depending on context, may need translating as the empty list or `None()` of the `option` type. For those unfamiliar, the `option` type indicates the presence or absence of a value and has two constructors: `None()` and `Some(<value>)`.

In the ACL2 x86 model, the main issues arise in distinguishing between `false` and `None()`, instances of which occur throughout the codebase. For instance, in a specification function for a bitwise shift, we have the form `(mv 0 nil 0 0)`. This returns multiple values from a function, one of which is `nil`. In this case, as it happens, `nil` ought to be translated as `false`,

given the other branches the function may take. On the other hand, we also have instances scattered throughout the codebase of the form `(mv <symbol> <exprs>)` where the first value returned is a symbol used for error reporting. If there is no error, we have `(mv nil <exprs>)`—we choose to translate these errors as expressions of the type `option(string)`, so here `nil` ought to be translated as `None()`. Clearly it is impossible to distinguish the two cases given just the `mv` forms (which is all the main translator function sees).

One possible solution would be to translate all instances of `nil` to `None()`, not trying to infer its type, then wrap instances where it is used in a boolean context in the function `is_some`. Indeed, `is_some` is used, as described later but, in general, such a blanket approach would produce undesirable results as all literals would have to be wrapped in an `option` type as well. A more nuanced approach is needed.

3.5.2 Heuristics

Although Staples [22] does provide some infrastructure for extensible type 'guessing', it is not the focus of his paper. This section, however, details techniques that were used in this project to implement more refined type guessing.

The first heuristic is simple. As mentioned, we find that the code adheres to an informal discipline where a `nil` in the pattern `(mv nil <exprs>)` corresponds to an error of type `option(string)` being returned in a different branch. As such, a relatively reliable guess is to translate all instances of `nil` to `None() : option(string)` when they appear as the first returned value in an `mv`. Indeed, empirically we can extend this to translating all instances to `None()` by default and amending them only if we receive information to the contrary based on the alternative schemes below.

The previous heuristic was a blanket 'zeroth-order' approach in which we took no account of context. This next heuristic does consider some basic typing. We saw in Section 3.4 that we do our best to infer the types of

formal parameters to functions. In some cases, this can help us infer the type of `nil`.

Specifically, if a function has a formal parameter whose type has been inferred as boolean, then we can translate a `nil` to which it is applied as `false :bool`. A similar method is used when `nil` is provided as a default value to keyword or *optional* formal parameters [18] which have been typed as boolean.

Other first-order cases include:

- In some contexts we can infer `nil` is a boolean. Examples include instances of the form `(if <cond> t nil)`, the type-spec `(or t nil)` and the guard `booleanp`. In each case it is highly likely the associated `nil` can be translated as a boolean.

- Sometimes data is stored in array literals. For example, an array stores `t` or `nil` for each opcode depending on whether the `modR/M` byte should be present in the encoded instruction. Again, we are fairly safe in translating these values as booleans.

Despite these heuristics, manual intervention is sometimes still necessary. Practically, this is achieved by checking each Lisp form against a list of predicates. If one of the predicates is satisfied, the form is not passed to the main translation function but instead executes some custom code. The predicates can be arbitrarily complex and reference any state in the current environment: it is not uncommon to match against the name of the current function being translated in order to implement custom translations. Some examples can be found in sections 3.5.4 and 3.6.

The advantage of this flexibility comes at the cost of over-specialisation: changes in the ACL2 source code could easily cause some interventions to stop working. There is also the risk of implementing what is effectively the same intervention multiple times without noticing, where a more general rule would work better.

3.5.3 A Further Solution

In order to try to reduce the number of manual interventions, a more nuanced approach is desirable. This problem is different from the one we saw in Section 3.4.3. In that case, uses of bound variables of unknown type were easily tracked and the point at which types were resolved was clearly at function applications. As described in the previous section, we do already test for `nil` being used as a formal parameter in a boolean context.

Instead, we look for a different syntactic structure which will allow us to unify an unknown type for a `nil` with a known type generated elsewhere in the code. We alight on conditional expressions.

It is at first tempting to think that we may instantly assign the boolean type to `nil`s used in the predicate for an `if` expression. This, again, is unfortunately not the case: given the falsey nature of `nil` other types may be used in this context. Consider a variable `flg : option(string)` representing a possible error from a function call. A common pattern is to perform the test `(if flg <expr-involving-flg> <otherwise-expr>)`. We can handle this situation easily by wrapping expressions of type `option` in the predicate `is_some` when they are supposed to be used in boolean contexts, but cannot infer they are a boolean type just because they are used in such contexts. This situation is similar to scripting languages such as Python which can use, for example, lists in boolean contexts to check for empty lists.

What we can do is compare the two branches of an `if` expression (or the multiple branches of `match`). This gives us four cases similar to the typing algorithm: if both branches represent `nil`s or neither is `nil`, no resolution takes place; whereas if exactly one branch is `nil` and the other has a well-specified type, we can use that to type the `nil` branch.

However, whereas in function application, unification inferred the type of exactly one bound variable, the branch of an `if` expression may represent more than one `nil` expression. For example, consider the following contrived example:

```
(if <cond1> (if <cond2> nil nil) ("Some string"))
```

The challenge in this case is to collect the `nil` terms in the inner `if` which need unifying. Cases of this form do appear in the ACL2 model.

It is at this point where the law of diminishing returns comes into effect. It would certainly be possible to generalise this approach such that information about `nil`s can be gained an arbitrary number of levels up the AST, with each syntactic element providing different information based on its structure; however with the heuristics and algorithm already described (which considers a single level of condition expression), most cases are handled correctly as it is.

Once again, we can express the concept behind the algorithm by inverting the typing rule for `if`. Whereas a classical typing rule may read:

$$\frac{b : \mathit{bool} \qquad e_1 : \tau \qquad e_2 : \tau}{\text{if } b \text{ then } e_1 \text{ else } e_2 : \tau}(\mathit{if})$$

one of the symmetric inverted versions becomes:

$$\frac{e_1 : \tau}{e_2 : \tau}(\mathit{infer\text{-}if1})$$

for e_1 and e_2 in the expression

$$\text{if } b \text{ then } e_1 \text{ else } e_2$$

The main difference between the way in which we type `nil`s and the way in which we type bound variables can be thought of as the difference between traversing the AST horizontally, in the case of the former, and vertically in the case of the latter.

3.5.4 Similar Problems

There also exist other Lisp expressions whose translation is ambiguous, although they are far less common:

- The symbol `t` is almost always translated as `true : bool`, but occasionally must be translated as `Some(...)`, where we have to come up with the contents. In reality, these cases are so rare that a manual intervention for each is preferred.
- Error reporting has been approximated by using Sail exceptions. In this case, an inline `throw` statement in Sail takes on a type appropriate to its context. The typing algorithm for `nil` described above is used to infer most instances of `throw`.

3.5.5 Conclusion—`nil`

This section has shown why the ambiguity of `nil` makes its translation awkward. It has also described the heuristics we use to avoid the problem and an algorithm for helping automatic decisions to be made.

3.6 Macros

Lisp has a very rich macro language which can generate arbitrary Lisp code. Without embedding a Lisp interpreter within Sail (or dynamically linking to a running ACL2 session) we must expand macros at translate time (at which point we can translate the expanded code) or handle them manually. Macros are used for code generation extensively throughout the project, so the debate becomes, again, which to expand automatically and which to translate manually. This section briefly describes some examples.

We frequently find snippets which use `make-event` as follows:

```
(make-event (add-af-spec-gen-fn 8))
(make-event (add-af-spec-gen-fn 16))
(make-event (add-af-spec-gen-fn 32))
(make-event (add-af-spec-gen-fn 64))
```

This code uses uses the function `add-af-spec-gen-fn` to generate code specialised for different data widths. The current solution is to define a special token translation for the macro `make-event`: a running instance of ACL2 gives us the AST generated by a call to `add-af-spec-gen-fn`, the result of which we then pass to the function which translates function definitions. Another possibility is to automatically identify instances of the above pattern and, following the example of Armstrong et al. [2, Sec. 3.1], leverage Sail's dependent type system. For example, we could generate a Sail function with the following type signature where the input type depends on the term to which the function is applied:

```
val add_af_spec : forall 'm,
              'm in {8, 16, 32, 64}.
              bits('m) -> bit
```

However, this would be considerably harder as it would necessitate handling Lisp quasiquotations: we are currently better served by using the underlying Lisp for this instead.

The largest examples of code generation are the dispatch functions which execute the correct semantic function based on the opcode of the current instruction. These macros expand to many thousands of lines of Sail code.

At a lower level, it is sensible to handle some macros manually. For example, it turns out that `OR` is a macro with some subtleties. In most cases, a translation to the expected function with type `(bool, bool) -> bool` works perfectly well. However, in reality, the `OR` macro expands to a form which returns its first non-`nil` argument, or `nil` if all arguments evaluate to `nil`. This unusual behaviour is used, but only occasionally, and so a manual intervention in these cases is a good solution.

3.7 Conclusion—Method

This chapter described interesting implementation details and design decisions of the translator and presented two algorithms to enhance type inference. It started by presenting an extended example and an overview of the translator. It then went into detail about typing ACL2 code so we have enough information to generate Sail functions, including how we extract basic types, how to fill in the gaps, how `nil` presents an ambiguity in translation and how to solve the issue. Finally, a brief description of macro handling was given.

Throughout the chapter we saw examples of two important points. Firstly, the number of Lisp forms which required special token translation demonstrates the large scale of the project. Secondly, the number of forms which required a manual intervention to override, generally good, default assumptions also added to the complexity of the project both in debugging where and why the issues arose and in developing correct interventions to solve them.

Chapter 4

Results

The aim of the project was to produce a translator which could automatically convert the x86 ACL2 model to Sail code. The evaluation of the translator, however, for a large part relies on the artefact it produces: the Sail model itself. This chapter first describes how the Sail model is validated against the Lisp model then presents and discusses the results of this validation and an evaluation of the test suite used.

4.1 Methodology

This section describes the technique of *co-simulation* and how coverage analysis is used to evaluate the quality of a chosen test suite.

4.1.1 Co-simulation

It is important that we have a high degree of confidence that the semantics of the translated model match those of the original ACL2 model. One approach, called co-simulation, makes use of the ability to execute the model. Co-simulation involves running the same series of instructions, from the same

starting state, in both models, and comparing the results. The resolution and detail with which states are compared can vary:

- The information logged can include registers and memory reads and writes. In this project we choose to track only register state.
- Logging can occur at the end of a program run, at specific breakpoints or at the end of every cycle. To ensure each tested instruction executes correctly we opt for the final approach here.

Co-simulation is widely used to validate ISA models against results from real processors and was used in Goel's work [1] to validate the ACL2 model. Mismatches in states between the model and a processor indicated modelling errors which were corrected. Here, co-simulation is used to provide confidence that the translation matches the model. Mismatches mostly indicated translation errors: they were investigated by stepping through pieces of custom assembly code to find the source of the problem before being corrected. Details of the results are reported in the next section.

In terms of implementation, the ACL2 model was already equipped with instrumentation functions which could log the output needed. However, some extensive implementation of similar functions at the Sail end and of scripts to compare outputs from the respective models was required.

4.1.2 Coverage

It is clearly impossible to exhaustively test every possible input to the model because the state space is too large (the contents of main memory can itself be considered an input). Thus, we are forced to choose test inputs. The choice of instructions to run affects the resultant level of confidence that the translation is correct. For this reason, it is useful to have some idea of instruction *coverage*—a measurement of how much of the model has been tested.

This project used Sail's inbuilt coverage tool, which, for a particular execu-

tion run, reports the branches which were taken out of all possible branches in each Sail source file.

4.1.3 Test Suites

The test programs used in Goel's original co-simulation of the model were chosen 'more or less randomly... with judiciously chosen inputs that represent certain classes of possible input values' [1] but were not detailed. Furthermore, coverage analysis and test generation from the Lisp source was noted as future work. Currently, the tests used to validate the model in industry at Centaur are not publicly available.

For these reasons, it was necessary to find or create other tests. The chosen test suite came from another notable x86 model developed by Dasgupta et al. [23]: in particular, its *single instruction tests* which test individual instruction mnemonics with carefully chosen inputs. The results from using these tests, and their limitations, are discussed in the next section.

4.2 Results and Discussion

This section evaluates the project and discusses the results of co-simulation, test-suite coverage, and translator features.

4.2.1 Results of Co-simulation

Failures in co-simulations were often traced back to errors in implementing handwritten functions. Examples include:

- Using the incorrect argument order in a function signature.
- An incorrect implementation of a function which was supposed to implement division rounding towards negative infinity.

- Incorrect indexing of a Sail vector.

Sometimes, however, errors were more subtle. In particular, attempting to write large values (greater than $2^{60} - 1$) to registers resulted in an incorrect value being written. One solution was to change the underlying Lisp on which ACL2 was built from Clozure Common Lisp (CCL) to Steel Bank Common Lisp. The root cause is still unknown and is outside the scope of this project, although it is suspected to be related to how CCL represents different sizes of number internally.

The most interesting discovery, however, was a possible error in the original ACL2 model. It should first be noted how this was discovered. There was no discrepancy in co-simulation traces between the ACL2 and Sail runs, but it was noted that the time to simulate this particular test program was unexpectedly long and hit the 10 000 step limit imposed to limit execution time. The instructions used in this test are presented and explained below.

```
1  leaq -8(%rsp), %rsi
2  leaq -16(%rsp), %rdi
3  movq $0x1121314151617181, %rax
4  pushq %rax
5  movq $1, %rcx
6  rep movsq
7  ...
```

Lines 1-2 load two memory addresses into the registers `rsi` (source) and `rdi` (destination) registers respectively.

Lines 3-4 load some arbitrary data to the memory location currently pointed to by `rsi`.

Line 5 loads the number 1 into the register `rcx`.

Line 6 is the problem. It comprises two parts: a `rep` prefix and the `movsq` instruction. The instruction `movsq` copies the data in the memory location pointed to by `rsi` into the memory location pointed to by `rdi`. The `rep` prefix modifies this instruction to repeat a certain number of times: the value in

`rcx` dictates how many times, and the value in that register is decremented on each iteration. In our case, it should only execute once before proceeding to the rest of the program.

In reality, although one iteration was completed as expected, the execution then continued to perform iterations of `movsq` with the value in `rcx` underflowing to $2^{64} - 1$ and decrementing from there.

Analysis of the root cause in the ACL2 code shows that the problem occurs when we test if the counter `rcx` is 0. If `rcx == 0` then we should update the instruction pointer to the next instruction, otherwise it should remain at `rep movsq` ready for the next iteration. However, this update is in the wrong branch.

Reports on the number of functions for which co-simulation produced matching traces are shown in the next section.

4.2.2 Coverage

As discussed in Section 4.1.3 the K Framework single instruction tests were used to evaluate the translated model. This section evaluates the single instruction tests themselves by examining the coverage they provide—the findings are summarised in Figure 4.1.

The K Framework single instruction tests exercise 7% of the branches of the translated model. There are three parameters which the test suite does not vary sufficiently to provide better coverage:

1. Instruction variants for different data widths are often omitted. This is important because the same instruction mnemonic often maps to different opcodes for different data widths: this leaves not only the semantic functions of such opcodes untested but also the decoding paths that lead to them.

2. Instruction variants for different addressing modes are often omitted. For example, without going into too much detail on addressing modes,

Figure 4.1: Coverage Provided by the K Framework Single Instruction Tests when Considering Various Parts of the Sail Model

Branches Considered	Coverage / %
All branches	7
Non-dispatch branches	29
Operation Specification Functions	30
Handwritten Functions	77

although the arithmetic/logic operation specification function for addressing mode 'E-G' is exercised extensively, the 'G-E' mode (and thus the associated decoding and execution paths) are not tested at all.

3. Many instructions are simply not exercised. Notable omissions include jumping, subroutine, stack manipulation and rotate instructions as well as the mnemonics AND, BT, and SBB. Specifically, 19 out of 63 translated one-byte operation specification functions (30%) have been exercised. This is surprising because the K Framework project implements many of the untested instructions itself.

However, there are certain branches which it would be unreasonable to expect the test suite to cover:

- The main examples are the macro-expanded dispatch functions which call the correct semantic function for a given opcode. Before dispatching control, the branch for each opcode also conducts various tests (e.g. whether the model implements the opcode or whether the opcode is legal). These tests are highly repetitive because they are generated automatically, and it is not necessary to exercise them all. When such dispatch tables are removed from coverage calculations, the K Framework single instruction tests provide 29% coverage.
- Utility functions are generated for accessing and updating certain structures. Again, the code is formulaic and if a single instance works, the rest are likely to work as well.
- The source code for the test suite was written in assembly, and so it is unreasonable to expect it to test malformed binary instructions (for

instance, invalid opcodes). Unfortunately, the coverage provided when discounting such branches is far harder to calculate as they are spread throughout the code in inline exceptions.

It should also be noted that 50 out of 119 (42%) instruction variants the test suite implemented were not implemented by the ACL2 model and that 12/119 (10%) were not translated. So, although the model does not exercise all instructions in the model, neither does the model use all the implemented tests.

Finally, co-simulation mismatches occurred mostly due to errors in the hand-written functions. It is thus heartening to see that the K Framework suite tested enough of the model to exercise 77% of handwritten branches, the remainder mostly representing boilerplate code to read and write certain registers.

4.2.3 Other results

Finally it is worth quantifying some of the features of the translation and the translator as well as looking at a performance comparison of the ACL2 and Sail models.

ACL2 Features Translated

The following features of the ACL2 model have been translated:

- The ACL2 model distinguishes between *app-view*, which provides an OS-like abstraction, and *system-view*, which does not. This project translated the functions necessary to implement app-view, which allows us to avoid managing complex structures such as page tables required only by system-level software such as operating systems.
- The ACL2 model focussed on modelling the *long* and *32-bit protected* modes of the *IA-32e* architecture. Testing of the translation focusses on the most relevant mode therein: the *64-bit mode*.

- All opcodes with a one-byte operand which were implemented in the ACL2 model have been translated except for those relating to system calls and random number generation. Some of these functions are re-used in the two-byte opcode map. Floating-point instructions have mostly not been translated.

Specifically, 63 specification semantic functions have been translated (along with 17 of the easier floating-point semantic functions). Between these semantic functions and the framework associated with decoding their binary representation this equates to 12 570 lines of generated Sail code. There are an additional 20 728 translated lines associated with dispatching the aforementioned semantic functions and 742 lines of handwritten Sail implementing basic functionality. Thus, a total of 34 040 lines of Sail code were generated.

Features of the Typing Algorithms

This section briefly assesses the effectiveness of two algorithms presented in Section 3.4.3 and 3.5.3.

The typing algorithm presented in Section 3.4.3 resolved types at 67 function applications, considerably reducing the effort in specifying their types manually or working out how to infer them from the syntax.

As a result of implementing the `nil` resolution algorithm in Section 3.5.3, seven manual interventions were removed and are now handled automatically. It is possible that a further three could be removed with a more generalised algorithm—here, however, the implementation effort would be disproportionate to the automation gain.

Performance Comparison

To demonstrate a larger example than the K Framework single instruction tests, a small bubble sort program, written in C and compiled with GCC was run in both ACL2 and a C emulator generated from the Sail translation. The

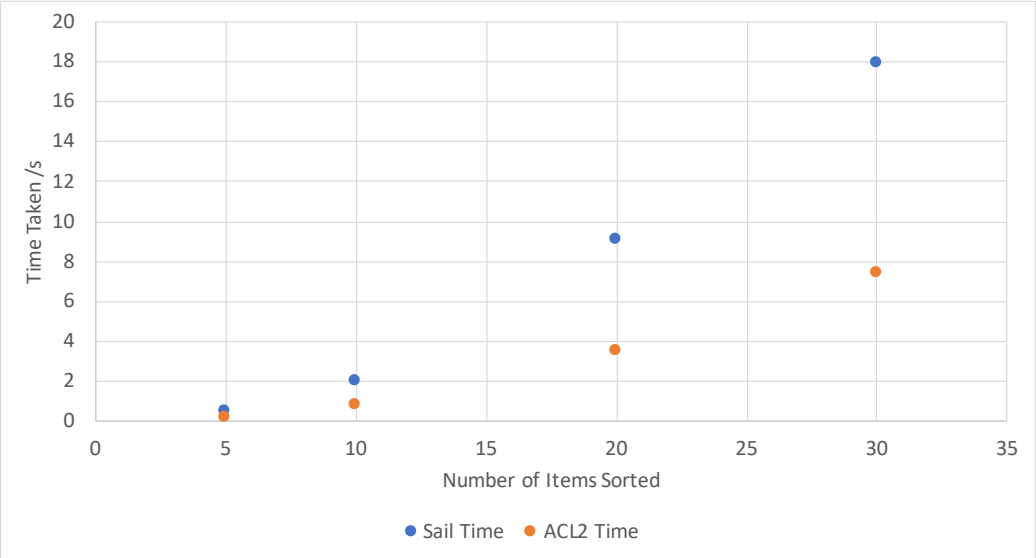

Figure 4.2: Comparison of Sail and ACL2 Execution Times for Bubble Sort of Varying Input Size

bubble sort program was a modified version of one found elsewhere in the K Framework test suite [24]. For each emulator run, co-simulation ensured the two traces matched and the final output of the ACL2 model was checked to ensure the output was correctly sorted.

Figure 4.2 shows the time in seconds for both emulators to complete a bubble sort of varying size. The input to each program comprised a list of numbers in reverse order so the quadratic time is as expected. The ACL2 emulator outperforms the Sail emulator by 2.6 ± 0.2 times.

This result is unsurprising. The underlying Lisp implementation, SBCL, is heavily optimised for arbitrary precision integer arithmetic and Goel emphasises the optimisations she makes to the x86 ISA model itself throughout her paper [1]. Sail, on the other hand, is better suited to fixed precision bitvector manipulation—the first obvious optimisation would therefore be to infer such bitvector widths in the translated model.

Other optimisations include: implementing more functionality manually at a higher level (at the expense of less automation) to reduce call stack depth;

and tightening up static typing as mentioned in Section 3.4.2 to reduce the number of dynamic type checks.

4.3 Conclusion—Results

This chapter has presented the co-simulation technique and shown how it flagged up an implementation error in the original ACL2 model. It has described how the coverage provided by the test suite is not ideal—this situation could be improved by using an instruction generator which creates instructions based on the code or structure of x86-64 instructions. It has also shown some of the main features of the translator and how the implementation details described in Chapter 3 influenced them as well as undertaking an example performance comparison.

Chapter 5

Related Work

In normal compilation, the features of a high level language are progressively encoded in lower level representations until machine code is generated. Source-to-source compilation refers to the translation of one language to another whilst retaining a similar level of abstraction. This section covers examples of work in this area, related both to ISA models and to using Lisp as the source language, and compares them with the scenario this paper addresses.

5.1 Using Sail as the Target Language

As part of the development of Sail, Armstrong et al. [2] developed a tool to automatically translate Arm's ASL into Sail. One difficulty was type translation, although both ASL and Sail are first order imperative languages with static light-weight dependent type systems making them not too far removed. Much of the complexity in converting ACL2 to Sail came from the much larger gap in type systems: the necessity to robustly infer types statically using disparate information spread throughout the ACL2 code. One advantage for the ASL converter was that it was developed alongside Sail: features of the former steering the design of the latter (for example exception

support and arbitrary precision rationals). Although changes were made to the Sail codebase during this project, they were limited to minor modifications such as improving error messages or adding small utility functions, rather than language design. In any case, the similarity of ASL and Sail make this co-design appropriate, whereas in general, the broader scope of ACL2 makes it less applicable, although fortunately many stylistic features of the x86 model itself made it amenable to translation to Sail.

5.2 Using Lisp and ACL2 as Source Languages

In general, Lisp is known as the go-to language for metaprogramming, however this often refers to its ability to define domain-specific languages and convert them into Lisp code, rather than translating Lisp code to other languages. Despite this, there are cases where Lisp is used as the input language in source-to-source compilers.

Perhaps the most well-known is ClojureScript, as described by McGranaghan [25], which is of interest because it compiles, not to a lower-level language, but to JavaScript. The most relevant comparison to our project is its handling of macros. Macros are fully expanded at compile time and the resulting code is then compiled into JavaScript [26]. Overall, however, many details are less similar: JavaScript is dynamically and weakly typed, unlike Sail which is strongly and statically typed, making JavaScript an easier target than Sail; one focus of ClojureScript compilation is size of output code, which is not a concern in this project; conversely, human-readability of the output code is important in Sail, but less so for the ClojureScript compiler.

The most relevant example comes from Mark Staples [22] who designed a framework for creating a shallow embedding of ACL2 in HOL. Many techniques he used are similar to ones used in this project, such as:

- Expanding macros before translation.
- Providing a set of translations for basic Lisp forms (such as literals

and arithmetic expressions) whilst allowing more complex forms to be translated using either pattern matching (shown below) or via arbitrary translations.

- Developing some *type guessing* heuristics to translate untyped lists into typed HOL terms.
- The assumption of monomorphism.
- The acknowledgement that `nil` is tricky to translate.

To demonstrate the framework he presented the translation of ACL2's *Small Machine* theory—a model of a toy computational machine. The example uses simple patterns to translate ACL2 expressions. For instance, `("(EQUAL X Y)" , 'X == Y', [])` specifies that the Lisp list beginning with `EQUAL` should be translated to the infix term `==` with recursively translated sub-expressions `X` and `Y`. Typing is directed from the HOL end: in this example, the use of `==` means that a function with a boolean return type is produced. Human-directed type guessing is used to infer the types for `X` and `Y`: they can be specified as either naturals or integers. It is also used to inform the translation of `nil`.

There are some notable differences between Staples' approach and this project.

- Type information in the ACL2 syntax is not used. Although the type guessing framework is extensible it is not clear that it would be as versatile as the solution in this project.
- Staples' approach is designed for dynamic translation between two running instances of ACL2 and HOL. This project implements a static translation, which Staples noted would likely not be fully automatic.
- Staples notes that it is sometimes sensible to translate an ACL2 definition as a constant in HOL (where there is neither a handwritten nor automatic implementation) and only fully translate theorems about it. Our translation to Sail was required to be executable, so this technique is not applicable.

- Staples focusses on creating the overall translation framework, leaving scope for individual projects to implement their own translations with term and type guessing extensibility. This project effectively implements those details for the ACL2 x86 model.
- The Small Machine example was limited and did not cover many of the complexities encountered in this project.

There is, however, a project from Susanto and Melham [27] which made use of Staples' tool by integrating a formal model of an ARM7 core written in Lisp/ACL2 with formal reasoning techniques in HOL. The function which modelled execution in ACL2, `ARM7execute` was used as a base-constant in HOL, and ACL2 was treated as an *axiom-server* to provide theorems about it. This demonstrates that Staples' translation tool was sophisticated enough to be used in a large project but shows how its dynamic nature allowed less to be translated than is required in this project.

Another paper based on Staples work from Michael Gordon et al. [28] creates a deep embedding where Lisp lists (s-expressions) are represented directly in HOL and only built-in, undefined functions are translated by hand. This increases confidence in translated theories compared with Staples' approach because the translation to the s-expression representation in HOL is relatively simple and, from there, properties about s-expressions (and then the translated definitions) are formally verified in HOL. In other words, a model of Lisp is created within HOL. This approach is suitable for formal reasoning but not for translation into Sail: we want both efficient execution and an idiomatic code style.

5.3 Conclusion—Related Work

This chapter has identified the related work in translating both from Lisp/ACL2 and into Sail.

Chapter 6

Conclusion

This dissertation has presented a translation into Sail of a model of the x86 ISA. The main contributions can be categorised into two themes:

1. **The Translator**. Translating from ACL2 into Sail was a non-trivial task for the following reasons: ACL2 is dynamically typed but Sail is strongly and statically typed; there were ambiguities caused by ACL2 terms such as `nil`; and there were a large number of ACL2 tokens which needed intricate, manual translations to produce idiomatic Sail code. Algorithms and heuristics were developed to resolve ambiguities and unknown types. A trade-off between Sail code style, translator maintainability and translator automation was noted.

2. **The Translation**. Co-simulation was used to validate the Sail translation against the original ACL2 and an error in the original ACL2 model was discovered. Coverage analysis of the chosen external test suite showed where coverage could be improved.

To conclude, Greenspun's satirical *tenth rule* [29] states "*Any sufficiently complicated . . . program contains an . . . implementation of half of Common Lisp*". Although this hints that many Lisp features are desirable in other languages this is not universally the case, and it is hoped that translation into idiomatic Sail is one case where such a 'half implementation' is justified.

Bibliography

[1] S. Goel, "Formal verification of application and system programs based on a validated x86 ISA model," Ph.D. dissertation, 2016.

[2] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. Krishnaswami, and P. Sewell, "ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019. [Online]. Available: https://doi.org/10.1145/3290384

[3] *Combined Volume Set of Intel 64 and IA-32 Architectures Software Developer's Manuals*, Intel, 05 2020.

[4] A. Reid, "Trustworthy specifications of ARM® v8-A and v8-M system level architecture," in *2016 Formal Methods in Computer-Aided Design (FMCAD)*, 2016, pp. 161–168.

[5] "Who guards the guards? Formal Validation of the Arm v8-M architecture Specification," vol. 1, no. OOPSLA. New York, NY, USA: ACM, October 2017, pp. 88:1–88:24.

[6] ACL2 homepage. [Online]. Available: http://www.cs.utexas.edu/users/moore/acl2/

[7] ACL2 documentation. [Online]. Available: http://www.cs.utexas.edu/users/moore/acl2/v8-3/combined-manual/index.html?topic=ACL2____Common_02Lisp

[8] W. Hunt, M. Kaufmann, J. Moore, and A. Slobodova, "Industrial hardware and software verification with ACL2," *Philosophical Transactions of The Royal Society A Mathematical Physical and Engineering Sciences*, vol. 375, p. 20150399, 10 2017.

[9] S. Goel, A. Slobodova, R. Sumners, and S. Swords, "Verifying x86 instruction implementations," in *Proceedings of the 9th ACM*

SIGPLAN International Conference on Certified Programs and Proofs, ser. CPP 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 47–60. [Online]. Available: https://doi.org/10.1145/3372885.3373811

[10] x86ISA documentation. [Online]. Available: http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/index-seo.php/X86ISA____INTRODUCTION

[11] S. Dasgupta, D. Park, T. Kasampalis, V. S. Adve, and G. Roşu, "A Complete Formal Semantics of x86-64 User-Level Instruction Set Architecture," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19)*. ACM, June 2019, pp. 1133–1148.

[12] Stack Overflow: The most minimal lisp? [Online]. Available: https://stackoverflow.com/questions/4589366/the-most-minimal-lisp

[13] ACL2 documentation. [Online]. Available: http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/index-seo.php/ACL2____About_02Types

[14] ACL2 documentation. [Online]. Available: http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/index-seo.php/ACL2____GUARD-INTRODUCTION

[15] ACL2 documentation. [Online]. Available: http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/index-seo.php/ACL2____TYPE-SPEC

[16] ACL2 documentation. [Online]. Available: http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/index-seo.php/COMMON-LISP____DECLARE

[17] ACL2 documentation. [Online]. Available: http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/index-seo.php/COMMON-LISP____THE

[18] ACL2 documentation. [Online]. Available: http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/index-seo.php/STD____EXTENDED-FORMALS

[19] ACL2 documentation. [Online]. Available: http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____B_A2

[20] Sail manual. [Online]. Available: https://github.com/rems-project/sail/blob/sail2/manual.pdf

[21] S. Sierram and L. VanderHart, *ClojureScript: Up and Running*, 2012.

[22] M. Staples, "Linking ACL2 and HOL," University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, phone +44 1223 763500, Tech. Rep. UCAM-CL-TR-476, Nov. 1999.

[23] K Framework single instruction tests. [Online]. Available: https://github.com/kframework/X86-64-semantics/tree/master/tests/single-instruction-tests

[24] K Framework bubble sort test. [Online]. Available: https://github.com/kframework/X86-64-semantics/tree/master/tests/program-tests/bubblesort

[25] M. McGranaghan, "Clojurescript: Functional programming for Javascript platforms," *IEEE Internet Computing*, vol. 15, no. 6, pp. 97–102, 2011.

[26] Clojurescript macros. [Online]. Available: https://code.thheller.com/blog/shadow-cljs/2019/10/12/clojurescript-macros.html

[27] K. W. Susanto and T. Melham, "An AMBA-ARM7 formal verification platform," in *Formal Methods and Software Engineering*, J. S. Dong and J. Woodcock, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 48–67.

[28] M. J. C. Gordon, W. A. Hunt, M. Kaufmann, and J. Reynolds, "An embedding of the ACL2 logic in HOL," in *Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and Its Applications*, ser. ACL2 '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 40–46. [Online]. Available: https://doi.org/10.1145/1217975.1217984

[29] "Greenspuns tenth rule of programming." [Online]. Available: http://wiki.c2.com/?GreenspunsTenthRuleOfProgramming