Executable semantics of Arm's Architecture Specification Language

Hadrien Renaud

Executable semantics of Arm's Architecture Specification Language

Hadrien Renaud[1]

[1]University College London, London, United Kingdom

The behaviour of Arm instructions is specified using a pseudocode language called ASL that does not yet have semantics. We present an interpreter for this language in both sequential and concurrent setting. This allows us to compare the behaviours of AArch64 instructions as implemented by *herd7*, the tool which Arm uses for describing its memory model, against the behaviours of the same instructions as described in ASL. We find discrepancies between those semantics for the Compare-And-Swap instruction.

1 Introduction

The specification of the Arm architecture is a long document called the Arm Architecture Reference Manual (Arm ARM) [Arm23]. In the Arm ARM, one can find a list of all AArch64 instructions with a representative description of its sequential behaviours in a language called Architecture Specification Language (ASL). The Arm ARM also gives the definition of the Arm memory model, or in other words the rules that govern the execution of a concurrent program written in Arm assembly. The Arm memory model is a formal and executable artefact, written in the domain-specific language *cat* [AMT14, ACM16].

ASL The ASL written in the Arm ARM does not have a formal definition, but rather an informal understanding of how it should work, explained at the end of the Arm ARM (Appendix K 16 page 12795 [Arm23]). After some efforts to build verification flows on top of this pseudocode [Rei16], Arm has decided to transition between this informal pseudocode (referred to as ASLv0) to a new language called ASLv1.

ASLv1 is an imperative language, strongly typed, with loops, exceptions, and functions. The main specificity of this language is its encoding of bit-vectors: bit-vector types are dependent on a length and operations on those are checked during type-checking. The language allows some polymorphism on the lengths of bit-vectors: a function can be declared to work on bits of length `N` and all the subsequent operations will work on this symbolically defined bit-vector length. Although there is a (still in review and private) specification for the language, this is not a semantics for the language, i.e. a specification of how to execute ASL code, merely a formal syntax with explanations and some type-checking rules.

The memory model The *cat* file which gives the definition of the Arm memory model can be executed by the *herd7* tool [AMT14]. This tool executes symbolically all the instructions in a test, which create events (or Effects as Arm calls them). For example, a load (resp. store) instruction creates a Memory Read (resp. Write) Effect. It also keeps track of the ordering constraints between the different Effects. Those ordering constraints can come

from explicit instructions such as fences, or from data-flow constraints. The *herd7* tool takes as input a model written in *cat*, and uses it to filter out the executions where those ordering constraints don't comply with this model, e.g. when they form a cycle.

The Arm memory model uses two kinds of intra-instruction ordering constraints: *Intrinsic data* dependencies indicate that the value produced by the first event has been used by the other; and *Intrinsic control* dependencies indicate that a binary decision made in a first event allows or not the other event to happen. There is no precise definition of those ordering constraints, and earlier work [AMT14] and the Arm ARM only mention their existence. The Arm ARM also lists intra-instruction dependencies for some instructions, e.g. loads, stores and Compare-And-Swap (see Section 3).

The semantics of AArch64 instructions in *herd7* are devised in tandem by the Arm staff and the *herd7* authors. When possible, this is done using the listing provided by the Arm ARM; otherwise, this work is a loose interpretation of the Arm ARM. This work is a first step in providing semantics to the ASL language, and thus equip the Arm ARM with a semantics of instructions.

Motivation The *herd7* model thus relies on handwritten semantics of instructions. By providing semantics to the ASL code in the Arm manual, we aim at clarifying the semantics of the instructions and enabling comparison of those semantics and the ones given in *herd7*. With an executable ASL semantics, *herd7*'s ASL interpreter can build instruction semantics directly from the Arm ARM's companion artefacts.

2 Contribution

We present two interpreters for ASL, called *ASLRef*, provided alongside the *herd7* tool [AM23], in the directory `asllib`. We use monadic style OCaml to build interpreters which helps building a purely functional interpreter that implements imperative features. They also make our implementation very flexible: by keeping the interpretation monad abstract, we can give the language different semantics by using different monads. We will use this feature to define both sequential and concurrent semantics. For the latter, the monadic style also helps implementing symbolic execution at a low cost. For concurrent semantics, we instantiate our interpreter with a symbolic execution monad, adding a state monad keeping track of the ordering constraints induced by the ASL code. For example, an Intrinisc data dependency is added between two events when there is a data-flow chain between the two.

With our interpreter for ASLv1, also comes a tool to transliterate ASLv0 into ASLv1. Although not complete, this makes it possible to execute ASLv0 pseudocode with our interpreter. For example, it can parse the 37000-line-long ASLv0 library which comes with the pseudocode in the Arm ARM' We also present a working type-checker for ASLv1. It has not yet been the subject of any theoretical study.

3 Validation

For some instructions, the Arm ARM comes with a listing of intra-instruction dependencies. The *herd7* tool follows the Arm ARm in that respect, building Effects and the Intrinsic Dependencies that order them as per those listings.

Our work provides another way to build Effects and Intrinsic Dependencies for a given instruction: we can use our interpreter on the ASL code given for this instruction. One might expect those two ways of building the semantics of an instruction to coincide. Our work demonstrates that this is not always the case and aims to remedy this state of affairs by reconciling both semantics sources. In the following, we will compare the intra-instruction dependencies for the Compare-And-Swap (`CAS`) instruction, for which the Arm ARM provide a list of intra-instruction dependencies.

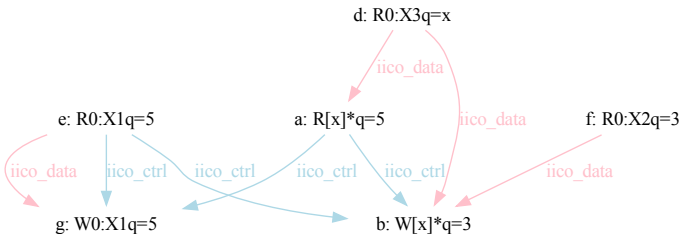

Figure 1. Intrinsic dependencies for `CAS` (ok case)

```
let address = X[n];
let data = Mem[address];
if data == X[s] then
  Mem[address] = X[t];
end
X[s] = data;
```

Figure 2. Simplified ASL code for `CAS`

Example of the `CAS` instruction Informally, the instruction `CAS X1, X2, [X3]` first reads the address indicated by `X3`; then, if the value read is identical as the value in `X1`, it writes to the address indicated by `X3` the value in register `X2`; in the end, it writes the value read from memory into the register `X1`. In the following, we say that `CAS` is in the *ok* case when the comparison is successful and the write to memory is done.

The intrinsic dependencies graph of herd for this instruction, shown in Figure 1, matches the list of intrinsic dependencies of the Arm ARM for this case. On the other hand, Figure 2 shows a simplified version of ASL pseudocode for the `CAS` instruction, taken from the Arm ARM. With our semantics of ASL, we can identify where there should be intrinsic dependencies from this ASL code, and we find some differences with the semantics given by herd. One notable difference is intrinsic data dependencies that arrives on the Write Register effect (line 6 of Figure 2, or arrows $e \to g$ and $a \to g$ on Figure 1). For each of those potential dependencies, we can construct small AArch64 assembly programs called *litmus tests* that can be simulated [AMT14] or executed on hardware [AMSS11]. Three tests [MR23a, MR23b, MR23c] helped Arm architects to decide which intra-instruction dependencies should exist: the ones from *herd7*, the ones from the Arm ARM, or both? The conclusion reached by Arm is that there should be a non-deterministic choice between either dependency, which is novel for Arm instructions. We have implemented this in *herd7* [Ren23], and we have suggested changes to the ASL code for CAS that are currently being discussed within Arm.

4 Conclusion

We have proposed two interpreters for ASL, one sequential and one concurrent. This has allowed the automatic extraction of intra-instruction dependencies from the ASL code associated with an AArch64 instruction. We have exhibited discrepancies between the *herd7* semantics and the ASL semantics of some instructions, such as CAS, that have been fixed in *herd7* and fixes for the corresponding ASL code are in discussion within Arm.

Further work Pre-existing ASLv0 interpreters exist, both internally and externally to Arm [Rei16, Rei20]. We are currently working on regressing our interpreter against those. To do so, we are building a tool called *ASLCarpenter*, provided alongside our interpreters, which generates ASL code for fuzzing those pre-existing interpreters with respect to ours.

We also aim to automate the work of comparing the *herd7* intra-instruction dependencies and the ones extracted from the ASL code from an instruction. For a given AArch64 litmus test, we can replace the *herd7* semantics of AArch64 instructions by executing the ASL code associated with an instruction. Both way of executing tests should behave the same way with respect to the memory model: any discrepancy should reveal a divergence in the way intra-instruction dependencies are computed. This experiment has been devised with Jade Alglave and is now carried on by Luc Maranget.

References

[ACM16] Jade Alglave, Patrick Cousot et Luc Maranget : Syntax and semantics of the weak consistency model specification language cat. *arXiv preprint arXiv:1608.07531*, 2016.

[AM23] Jade Alglave et Luc Maranget : herdtools7, 2023. `https://github.com/herd/herdtools7`.

[AMSS11] Jade Alglave, Luc Maranget, Susmit Sarkar et Peter Sewell : Litmus: Running tests against hardware. *In International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 41–44. Springer, 2011.

[AMT14] Jade Alglave, Luc Maranget et Michael Tautschnig : Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36(2):1–74, 2014.

[Arm23] Arm Ltd. : Arm architecture reference manual for A-profile architecture., 2023. `https://developer.arm.com/documentation/ddi0487/latest`.

[MR23a] Luc Maranget et Hadrien Renaud : *MP+rel+CAS-ok-bothRs-addr.litmus*, 2023. `https://github.com/herd/herdtools7/blob/8a794d05db101563517720885d31fbce2b778a2b/catalogue/aarch64/tests/MP%2Brel%2BCAS-ok-bothRs-addr.litmus`.

[MR23b] Luc Maranget et Hadrien Renaud : *MP+rel+CAS-ok-MRs-addr.litmus*, 2023. `https://github.com/herd/herdtools7/blob/8a794d05db101563517720885d31fbce2b778a2b/catalogue/aarch64/tests/MP%2Brel%2BCAS-ok-MRs-addr.litmus`.

[MR23c] Luc Maranget et Hadrien Renaud : *MP+rel+CAS-ok-RsRs-addr.litmus*, 2023. `https://github.com/herd/herdtools7/blob/8a794d05db101563517720885d31fbce2b778a2b/catalogue/aarch64/tests/MP%2Brel%2BCAS-ok-RsRs-addr.litmus`.

[Rei16] Alastair Reid : Trustworthy specifications of ARM® v8-A and v8-M system level architecture. *In 2016 Formal Methods in Computer-Aided Design (FMCAD)*, pages 161–168. IEEE, 2016.

[Rei20] Alastair Reid : ASL interpreter, 2020. `https://github.com/alastairreid/asl-interpreter`.

[Ren23] Hadrien Renaud : [herd] new semantics for CAS, 2023. `https://github.com/herd/herdtools7/commit/8a794d05db101563517720885d31fbce2b778a2b`.