MASR — Meta Abstract Semantics Representation

Brian Beckman

6 Apr 2023

Contents

1 Abstract (TL;DR)

1.1 ASR

The Abstract Semantics Representation (ASR) of LCompilers.[1] is a syntax-free, machine-agnostic intermediate language (IL) for multiple compilers. Current compilers targeting ASR include LFortran[2] and LPython.[3] The advantage of removing all vestiges of original language syntax from the IL is flexibility: it's easier to write new front-ends and optimizing back ends. This is in sharp contrast to standard practice of decorating abstract syntax trees (ASTs) with semantic information and then carrying those into the back ends. That makes back ends less reusable because most languages have syntactic peculiarities that must be reflected in their ASTs. ASR has no known downsides: it's fast, compact (in binary) and a full-featured programming language in its own right.

1.2 Issues and Mitigations

1.2.1 ASDL is Moribund

ASR is currently specified in ASDL[4], a moribund meta language published in 1987. To build an LCompiler, we read the ASDL grammar for ASR and generate fast

[1]`https://github.com/lcompilers/libasr`
[2]`https://lfortran.org/`
[3]`https://lpython.org/`
[4]`https://en.wikipedia.org/wiki/Abstract-Type_and_Scheme-Definition_Language`

processors in C++. Compiler front ends call these processors to manipulate ASR trees. Ultimately, ASR is fed to the compiler back ends, e.g., LLVM, x86, C, etc.

So far as we know, our ASDL tools are the only extant ones in the World. There is no ecosystem for it. We need to replace ASDL with something more modern with a robust, lively ecosystem.

1.2.2 ASR Processors are Hard to Prototype and Verify / Validate

ASR-to-ASR transformation is where the magic of LCompilers happens. Optimization, static type-checking, partial evaluation, abstract execution, and rewriting are examples of such transformations.

Though ASR has an output representation in S-expressions, all work with it is currently done in C++ with opaque binary representations. As usual with such, it's difficult (time-consuming and error-prone) to prototype, verify, validate, visualize, modify, and debug.

The mitigation is to perform these development activities with languages friendly to S-expressions and with rich toolkits for visualization and transformations. We've chosen Clojure, and have a prototype for validation and test-generation in Clojure.[5]

1.2.3 Volatility

ASR changes nearly daily, for good reasons. However, existing ASR processors in Clojure must chase the ASDL specification, and that chasing consumes almost all development time in Clojure.

We propose to replace the ground-level specification in ASDL with a ground-level specification in Clojure, directly. Initially, we can generate the needed C++ processors from Clojure, then rewrite them in C++ when ASDL stabilizes. I call this new ASR specification MASR, for Meta-ASR. It's pronounced "Maser," building on Physics puns that abound in our work.

1.2.4 Type Systems

Clojure has a powerful specification subsystem, clojure.spec.[6] This should suffice for both ordinary, everyday type-checking as well as advanced research into dependent types and concurrency types.[7]

2 Change Log

2023-06-Apr :: Start.

[5] `https://github.com/rebcabin/asr-tester`
[6] `https://clojuredocs.org/clojure.spec.alpha`
[7] `https://rholang.io/`