MASR — Meta Abstract Semantics Representation

Brian Beckman

10 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 back ends. This is in sharp contrast to standard practice of decorating abstract syntax trees (ASTs) with semantic information and then carrying decorated ASTs into the back ends. Such makes back ends less reusable because most languages have syntactic peculiarities 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]`https://github.com/lcompilers/libasr`
[2]`https://lfortran.org/`
[3]`https://lpython.org/`

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 parse the ASDL grammar for ASR and generate fast 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 ASDL. 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 transformations are the magic of LCompilers. 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, namely *asr-tester*.[5] The current work inherits learnings from that old project.

1.2.3 Volatility

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

We propose to replace the ground-level specification of ASR in ASDL with a ground-level specification in Clojure, directly. We propose to generate the needed C++ processors from Clojure, then rewrite them by hand in C++ if necessary 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.

Clojure for MASR has many advantages over the current approach:

- Metavariables, attributes, and types are explicit via clojure hash-maps[6] and records[7].

[4]`https://en.wikipedia.org/wiki/Abstract-Type_and_Scheme-Definition_Language`

[5]`https://github.com/rebcabin/asr-tester`

[6]`https://clojuredocs.org/clojure.core/hash-map`

[7]`https://clojuredocs.org/clojure.core/defrecord`

- The step from ASDL to Clojure is eliminated. The Clojure will automatically track ASR because it *is* ASR.

1.2.4 Type Systems

Clojure has a powerful specification subsystem, clojure.spec.[8] This should suffice for both ordinary, everyday type-checking as well as for advanced research into dependent types, concurrency types,[9] rewriting, and proofs.

2 MASR

The development in this document may lag the code. The document mirrors the structure of an ASR snapshot in ASDL.[10]

We introduce some terminology, explained in context.

2.1 Terms and Heads

Terms are the "objects" or "productions" of ASR, items to the left-hand side of an equals sign in the ASDL grammar. Table 1 exhibits terms that are fully specified, implicit, or term-like in the current ASDL grammar. The definitions have been abbreviated and edited for presentation and fit.

Heads are the things that appear on the right-hand sides of terms equations. They are of two kinds:

function-like heads — have parentheses and typed parameters.

enum-like heads — no parentheses

2.2 Critique and Objectives

There are many syntactic and semantic ambiguities in the ASDL grammar. These ambiguities are resolved in the actual, hand-written C++ code in LFortran and LPython that implements the grammar.

A primary objective of MASR is to resolve these ambiguities in Clojure, enabling more automation and relieving pressure on C++ programmers.

[8] `https://clojuredocs.org/clojure.spec.alpha`

[9] `https://rholang.io/`

[10] [`https://github.com/rebcabin/masr/blob/main/ASR_2023_APR_06_snapshot.asdl`]

Table 1: Terms (nodes) in the ASDL grammar (things left of equals signs):

	term	**partial expansion**
1	`unit`	`TranslationUnit(symbol_table, node*)`
2	`symbol`	. . . many heads . . .
3	`storage_type`	`Default|Save|Parameter|Allocatable`
4	`access`	`Public|Private`
5	`intent`	`Local|In|Out|InOut|...`
6	`deftype`	`Implementation|Interface`
7	`presence`	`Required|Optional`
8	`abi`	`Source|LFortranModule|...|Intrinsic`
9	`stmt`	. . . many heads . . .
10	`expr`	. . . many heads . . .
11	`ttype`	`Integer(int, dimension*)|...`
12	`restriction_arg`	`RestrictionArg(` ident `, symbol)`
13	`binop`	`Add|Sub|...|BitRShift`
14	`logicalbinop`	`And|Or|Xor|NEqv|Eqv`
15	`cmpop`	`Eq|NotEq|Lt|LtE|Gt|GtE`
16	`integerboz`	`Binary|Hex|Octal`
17	`arraybound`	`LBound|UBound`
18	`arraystorage`	`RowMajor|ColMajor`
19	`cast_kind`	`RealToInteger|IntegerToReal|...`
20	`dimension`	`(expr? start, expr? length)`
21	`alloc_arg`	`(expr a, dimension* dims)`
22	`attribute`	`Attribute(` ident `name,` attr-arg* `args)`
23	`attribute_arg`	`(` ident `arg)`
24	`call_arg`	`(expr? value)`
25	`tbind`	`Bind(string lang, string name)`
26	`array_index`	`(` expr? `left,` expr? `right,` expr? `step)`
27	`do_loop_head`	`(` expr? `v,` expr? `start` expr? `end,` expr? `step)`
28	`case_stmt`	`CaseStmt(expr*, stmt*)|...`
29	`type_stmt`	`TypeStmtName(symbol, stmt*)|...`
30	`enumtype`	`IntegerConsecutiveFromZero|...`
	implicit	
31	`symbol_table`	Clojure maps
32	`symtab_id`	an `int`
	***term-like**	
0	`dimensions`	`dimension*`, via Clojure vectors or lists
0	atoms	`int|float|bool|nat|bignat`
0	`identifier`	by regex
0	`identifiers`	`identifier*`, via Clojure sets

3 Concepts

The following sections explain the architecture and approach taken in the Clojure code. Overall, clojure.spec is *force majeure* for driving out ambiguity. The Clojure tests exhibit many examples that pass and, more importantly, fail the specs in the source.

3.1 Terms and Heads

We present these terms somewhat out of order. `intent` is first as it is the archetype for enum-like terms and heads.

3.1.1 intent

An `intent` is one of `Local`, `In`, `Out`, `InOut`, `ReturnVar`, `Unspecified`. The spec for the contents of an intent is simply this set of enum-like heads. A Clojure *set* in `#{ ... }` brackets doubles as a predicate function that tests set membership. A Clojure *spec* is any predicate function. Therefore the case of `intent` contents is very easy.

The name of the spec is `::intent-enum`. The double colon signifies that `intent-enum` is a keyword in the namespace `masr.specs`. The names of all specs must be so qualified as to namespace. In test files, we must explicitly name these specs.

```
(s/def ::intent-enum
  #{'Local 'In 'Out 'InOut 'ReturnVar 'Unspecified})
```

`::intent-enum` is the spec for the contents of an `intent`. The spec for `intent` itself is a Clojure *multi-spec*.[11] Such multi-specs are articulated via `defmethods` on a `defmulti`, as explained in the guide cited in the footnote.

```
;; ::term is a fn that picks the dispatch value
(defmulti term ::term)
(defmethod term ::intent [_]
  (s/keys :req [::term ::intent-enum]))
```

3.1.2 unit

3.1.3 symbol

1. **TODO** Variable

[11] `https://clojure.org/guides/spec`

3.1.4 storage_type

3.1.5 access

3.1.6 deftype

3.1.7 presence

3.1.8 abi

3.1.9 stmt

3.1.10 expr

3.1.11 ttype

3.1.12 restriction_arg

3.1.13 binop

3.1.14 logicalbinop

3.1.15 cmpop

3.1.16 integerboz

3.1.17 arraybound

3.1.18 arraystorage

3.1.19 cast_kind

3.1.20 dimension

3.1.21 alloc_arg

3.1.22 attribute

3.1.23 attribute_arg

3.1.24 call_arg

3.1.25 tbind

3.1.26 array_index

3.1.27 do_loop_head

3.1.28 case_stmt

3.1.29 type_stmt

3.1.30 enumtype

3.2 Implicit Terms

Terms used, explicitly or implicitly, but not defined in ASDL.

Some items specified in ASDL as *symbol_table* are actually *symtab_id*.

3.2.1 symtab_id

3.2.2 symbol_table

3.3 Term-Like Items

3.3.1 dimensions

3.3.2 atoms

3.3.3 identifier

3.3.4 identifiers

4 Change Log

2023-06-Apr :: Start.