MASR — Meta Abstract Semantics Representation

Brian Beckman

10 Apr 2023

Contents

1 Abstract

Abstract Semantics Representation (ASR) is an innovative intermediate representation[1] (IR) for multiple LCompilers.[2] ASR is independent of the particular programming language under compilation. Current compiler front-ends targeting ASR include LFortran[3] and LPython.[4] ASR is also agnostic to the compiler back end. Current back ends targeted *from* ASR include LLVM, x86, C, and WASM[5]

Being agnostic means that it is easy to write new compilers, both at the front end and the back end. For example, LFortran predates LPython. When the need for a Python compiler arose, only a Python front end was necessary. Within a few days, a new end-to-end compiler, LPython, was created.

LCompiler back ends are completely reusable because ASR eliminates all original language syntax from the IR, in sharp contrast to typical practice, which treat semantics as decorations on syntax trees.

In addition to being more flexible, LCompilers are faster than average because optimizers are not hampered by useless syntactic structure.

Current specifications for ASR are written in ASDL,[6] a metalanguage similar in spirit to yacc but less rich, by design.[7] To build an LCompiler like LFortran or LPython, the ASDL grammar for ASR is parsed and a library in C++ is generated. Compiler front ends call this library to transform and emit ASR trees.

ASDL has several deficiencies, and MASR, [8] described in this document, alleviates them.

This document is pedagogical, explaining MASR and teaching how to extend and maintain its Clojure code.

2 Issues with ASDL

2.1 ASDL is Moribund

ASDL has not progressed since originally published in 1987. We know of no other projects adopting ASDL. We should replace ASDL with a modern metalanguage that has a robust, lively ecosystem.

[1] https://en.wikipedia.org/wiki/Intermediate_representation
[2] https://github.com/lcompilers/libasr
[3] https://lfortran.org/
[4] https://lpython.org/
[5] https://webassembly.org/
[6] https://en.wikipedia.org/wiki/Abstract-Type_and_Scheme-Definition_Language
[7] https://en.wikipedia.org/wiki/Yacc
[8] pronounced "maser;" it is a Physics pun

2.2 ASDL is Incomplete

All work with ASR is currently done with opaque binary representations and code written in C++. As usual with such, it's more time-consuming and error-prone than necessary to prototype, verify, validate, visualize, modify, and debug.

2.3 ASDL is Volatile

The ASDL for ASR changes frequently, for good reasons. However, stand-aside tools like asr-tester[9] must chase the ASDL specification. Just keeping up with ASDL consumes almost all development time for asr-tester. We should unify the language for expressing ASR with the tools that verify and test ASR.

2.4 ASDL is Ambiguous

There are many syntactic and semantic ambiguities in the ASDL grammar. For example, the specification `integer*` might mean an ordered collection of `integer` or an unordered collection, with duplicate elements allowed or not.

ASDL cannot express such distinctions. In practice, the C++ implementations implicitly make these distinctions. In one place `integer*` means an ordered collection. In another place, `integer*` means an unordered collection, but only the generated C++ code knows where.

Hiding fine distinctions in specifications in the generated code is not good engineering practice in the face of a known better solution.

3 Clojure Solves ASDL Issues

ASR expressions, being trees, have a natural representation in S-Expressions.[10] Clojure, being a modern Lisp, natively handles S-Expressions. Clojure is modern with a robust, lively ecosystem.

Clojure.spec,[11] is a *force majeure* for precision, completeness, verification, and validation. The collection of MASR specs amounts to a meta-type system for ASR.

Clojure specs are arbitrary predicates, capable of expressing type-system logic beyond the typical capabilities of hard-coded type systems. Clojure specs can be invoked at run time, while a compiler is built, and at compile-time, while the code for the compiler is being generated. That flexibility affords new opportunities, say for experiments in dependent types and concurrency types.[12]. In the short run, clojure.spec will help us make more things explicit and relieve pressure on C++ programmers.

This document may lag the Clojure code. The document mirrors an ASR snapshot in ASDL.[13]

[9] `https://github.com/rebcabin/asr-tester`
[10] `https://en.wikipedia.org/wiki/S-expression`
[11] `https://clojuredocs.org/clojure.spec.alpha`
[12] `https://rholang.io/`
[13] `https://github.com/rebcabin/masr/blob/main/ASR_2023_APR_06_snapshot.asdl`

4 MASR 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

- explicitly specified in ASDL, like `symbol` or `dimension`
- used but not defined in ASDL, like `symbol_table`
- term-like but not defined in ASDL, like `identifier`

The definitions in Table 1 have been abbreviated and edited for presentation.

Heads are things like `Local` and `CaseStmt` that appear on the right-hand sides of terms equations in Table 1. There are of two kinds of heads:

function-like heads — have parentheses and typed parameters, e.g., `CaseStmt(expr*, stmt*)`

enum-like heads — no parentheses, e.g., `Local`

MASR has a Clojure spec and syntactic sugar for each head. There are about 250 heads by a recent count.

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

5 Specs

The following sections

- summarize the Clojure specs for all ASR terms and heads
- pedagogically explain the architecture and approach taken in the Clojure code. The architecture is the remainder from several experiments. For example, `defrecord` and `defprotocol` for polymorphism was tried and discarded in favor of multi-specs.[14]

The tests in `core_test.clj` exhibit many examples that pass and, more importantly, fail the specs. The specs are written in the source file `specs.cl`. The best way to learn the code is to study the tests and to run them in the Clojure REPL or in the CIDER debugger in Emacs.[15]

We present the terms somewhat out of the order of Table 1. First is *intent*, as it is the archetype for several enum-like terms and heads.

5.1 intent

5.1.1 Sets for Contents

An ASR *intent* is one of the symbols

`Local`, `In`, `Out`, `InOut`, `ReturnVar`, `Unspecified`.

The spec for the contents of an intent is simply this set of enum-like heads. Any Clojure *set* (e.g., in `#{ ... }` brackets) doubles as a predicate function that tests for set membership. The result of the test is the candidate member if the candidate is present in the set. In the following two examples, the set appears in the function position of the usual Clojure function-call syntax `(function args*)`:

```
(#{'Local 'In 'Out 'InOut 'ReturnVar 'Unspecified} 'Local)
```

```
Local
```

When the candidate element, say `fubar`, is not in the set, the result is `nil`, which does not print:

```
(#{'Local 'In 'Out 'InOut 'ReturnVar 'Unspecified} 'fubar)
```

Any predicate function can be registered as a Clojure spec.[11] Therefore the spec for *intent contents* is just the set of valid members. The name of the spec is `::intent-enum`.

5.1.2 Specs have Fully Qualified Keyword Names

The double colon in `::intent-enum` is shorthand; it implicitly signifies that `intent-enum` is a keyword in the namespace `masr.specs`. The names of all Clojure specs must be fully qualified in namespaces.

[14] `https://clojure.org/guides/spec`
[15] `https://docs.cider.mx/cider/debugging/debugger.html`

The namespace `masr.specs` covers all names defined in the entire file `specs.clj`, where the syntax `::intent-enum` is valid. In files other than `specs.clj`, one explicitly writes the namespace, as in `:masr.specs/intent-enum`. More on that later.

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

5.1.3 How to Use Specs

To check an expression like `'Local` against the `::intent-enum` spec, write

```
(s/valid? ::intent-enum 'Local)
;; => true
(s/valid? ::intent-enum 'fubar)
;; => false
```

To produce conforming or invalid instances inline to other code, write

```
(s/conform ::intent-enum 'Local)
;; => Local
(s/conform ::intent-enum 'fubar)
;; => :clojure.spec.alpha/invalid
```

To generate a few conforming samples, write

```
(gen/sample (s/gen ::intent-enum) 5)
;; => (Unspecified Unspecified Out Unspecified Local)
```

or, with conformance explanation (trivial in this case):

```
(s/exercise ::intent-enum 5)
;; => ([Out Out]
;;     [ReturnVar ReturnVar]
;;     [In In]
;;     [Local Local]
;;     [ReturnVar ReturnVar])
```

Strip out the conformance information as follows:

```
(map second (s/exercise ::intent-enum 5))
;; => (In ReturnVar Out In ReturnVar)
```

`s/valid?`, `s/conform`, `gen/sample`, and `s/exercise` pertain to any Clojure specs, no matter how complex or rich.

5.1.4 The Spec that Contains the Contents

`::intent-enum` is just the spec for the *contents* of an intent, not for the intent itself. The spec for the intent itself is an implementation of a polymorphic Clojure *multi-spec*[14], `::asr-term`.

5.1.5 Multi-Specs

A multi-spec is like a tagged union in C. The multi-spec, `::asr-term`, as shown below, pertains to all Clojure hash-maps[16] that have a tag named `::term` with a value like `::intent` or `::storage-type`, etc.

A multi-spec has three components:

`defmulti`[17] — a polymorphic interface that declares the *tag-fetcher function*, `::term` in this case, which fetches a tag value from any candidate hash-map. The `defmulti` dispatches to a `defmethod` that matches the fetched tag value, `::intent` in this case. `::term` is a fully qualified keyword of course, but keywords double as tag-fetchers for hash-maps in Clojure.[18]

`defmethod`[19] — individual specs, each implementing the interface; in this case, if the `::term` of a hash-map matches `::intent`, then the corresponding `defmethod` is invoked (see Section 5.1.7 below).

`s/multi-spec` — tying together the `defmulti` and, redundantly, the tag-fetcher.[20]

5.1.6 Specs for All Terms

Start with a spec for `::term`:

```
;; like ::intent, ::symbol, ::expr, ...
(s/def ::term qualified-keyword?)
```

The spec says that any fully qualified keyword, like `::intent`, is a MASR term. This spec leaves room for growth of MASR by adding more fully qualified keywords for more MASR types.

`s/def` stands for `clojure.spec.alpha/def`, the `def` macro in the `clojure.spec.alpha` namespace. The namespace is aliased to `s`.

Next, specify the `defmulti` polymorphic interface `term` (no colons) for all term specs:

```
(defmulti term ::term)
```

This `defmulti` dispatches to a `defmethod` based on the results of applying the keyword-qua-function `::term` to a hash-map:

```
(::term {::term ::intent ...})
```

equals `::intent`.

The spec is named `::term` and the tag-fetcher is named `::term`. They don't need to be the same. They could have different names.

[16] `https://clojuredocs.org/clojure.core/hash-map`
[17] `https://clojuredocs.org/clojure.core/defmulti`
[18] `https://stackoverflow.com/questions/6915531`
[19] `https://clojuredocs.org/clojure.core/defmethod`
[20] Multi-specs allow re-tagging, but we do not need that level of generality.

5.1.7 Spec for intent

If applying `::term` to a Clojure hash-map produces `::intent`, the following spec, which specifies all intents, will be invoked:

```
(defmethod term ::intent [_]
  (s/keys :req [::term ::intent-enum]))
```

This spec states that an *intent* is a Clojure hash-map with at least a `::term` keyword and an `::intent-enum` keyword. The following shows a valid example:

```
(is (s/valid? ::asr-term
              {::term        ::intent,
               ::intent-enum 'Local}))
```

Generate a few samples:

```
(gen/sample (s/gen (s/and
                    ::asr/asr-term
                    #(= ::asr/intent (::asr/term %))))
            5)
;;=> (#::asr{:term ::asr/intent, :intent-enum ReturnVar}
;;     #::asr{:term ::asr/intent, :intent-enum In}
;;     #::asr{:term ::asr/intent, :intent-enum Unspecified}
;;     #::asr{:term ::asr/intent, :intent-enum Unspecified}
;;     #::asr{:term ::asr/intent, :intent-enum InOut})
```

5.1.8 The Multi-Spec, Again: ::asr-term

Where did `::asr-term` come from? It's the `s/multi-spec`:

```
;;      name of the mult-spec    defmulti  tag fn
;;      ----------------------    -----    ------
(s/def ::asr-term (s/multi-spec   term     ::term))
```

This states that `::asr-term` is a multi-spec that ties `defmulti term` to the tag-fetcher `::term`.

5.1.9 Another asr-term

Another asr-term for an enum-like is that for `storage-type`:

```
(s/def ::storage-type-enum
  #{'Default, 'Save, 'Parameter, 'Allocatable})

(defmethod term ::storage-type [_]
  (s/keys :req [::term ::storage-type-enum]))
```

All enum-like specs follow this pattern. All one must do to define a new asr-term is specify the contents and write a `defmethod`.

5.1.10 Syntax Sugar

`{::term ::intent, ::intent-enum 'Local}`, a valid `asr-term` instance, is long and ugly. Write a short function, `intent`, via `s/conform`, explained in Section 5.1.3:

```
(defn intent [sym]
  (let [intent_ (s/conform
                 ::asr-term
                 {::term ::intent, ::intent-enum sym})]
    (if (s/invalid? intent_)
      ::invalid-intent
      intent_)))
```

so that instances have a shorter expressions:

```
(testing "better syntax"
  (is      (s/valid? ::asr-term (intent 'Local)))
  (is      (s/valid? ::asr-term (intent 'Unspecified)))
  (is (not (s/valid? ::asr-term (intent 'foobar))))
  (is (not (s/valid? ::asr-term (intent []))))
  (is (not (s/valid? ::asr-term (intent ()))))
  (is (not (s/valid? ::asr-term (intent {}))))
  (is (not (s/valid? ::asr-term (intent #{}))))
  (is (not (s/valid? ::asr-term (intent "foobar"))))
  (is (not (s/valid? ::asr-term (intent ""))))
  (is (not (s/valid? ::asr-term (intent 42))))
  (is (thrown? clojure.lang.ArityException (intent))))
```

All our specs are like that: a long-form hash-map and a short-form sugar function that does a conformance check.

5.2 unit

5.3 symbol

5.3.1 TODO Variable

5.4 storage_type

5.5 access

5.6 deftype

5.7 presence

5.8 abi

Abi is a rich case. It is enum-like, similar to *intent* (Section 5.1), but with restrictions. Its heads include several *external-abis*:

```
(def external-abis
  #{'LFortranModule, 'GFortranModule,
    'BindC, 'Interactive, 'Intrisic})
```

and one *internal-abi*, specified as a Clojure set to get the membership-test functionality:

```
(def internal-abis #{'Source})
```

The *abi-enum* spec for the contents of an *abi* term is the unions of these two sets:

```
(s/def ::abi-enum
  (set/union external-abis internal-abis))
```

Specify an additional key in a conforming *abi* hash-map with a `::bool` predicate:

```
(s/def ::abi-external ::bool)
```

Add a convenience function for logic:

```
(defn iff [a b]
  (or (and a b)
      (not (or a b))))
```

Specify the `defmethod` for the *abi* itself with a hand-written generator (clojure.spec is not quite strong enough to create the generator automatically):

```
(defmethod term ::abi [_]
  (s/with-gen
    (s/and
     #(iff (= 'Source (::abi-enum %))
           (not (::abi-external %)))
     (s/keys :req [::term ::abi-enum ::abi-external]))
    (fn []
      (tgen/one-of
       [(tgen/hash-map
         ::term         (gen/return ::abi-enum)
         ::abi-enum     (s/gen external-abis)
         ::abi-external (gen/return true))
        (tgen/hash-map
         ::term         (gen/return ::abi-enum)
         ::abi-enum     (s/gen internal-abis)
         ::abi-external (gen/return false))] ))))
```

Generate a few conforming samples:

```
(gen/sample (s/gen (s/and
                    ::asr/asr-term
                    #(= ::asr/abi (::asr/term %))))
            5)
;; => (#::asr{:term ::asr/abi,
;;            :abi-enum Interactive, :abi-external true}
;;     #::asr{:term ::asr/abi,
;;            :abi-enum Source, :abi-external false}
;;     #::asr{:term ::asr/abi,
;;            :abi-enum Source, :abi-external false}
;;     #::asr{:term ::asr/abi,
;;            :abi-enum Source, :abi-external false}
;;     #::asr{:term ::asr/abi,
;;            :abi-enum Interactive, :abi-external true})
```

5.9 stmt

5.10 expr

5.11 ttype

5.12 restriction_arg

5.13 binop

5.14 logicalbinop

5.15 cmpop

5.16 integerboz

5.17 arraybound

5.18 arraystorage

5.19 cast_kind

5.20 dimension

5.21 alloc_arg

5.22 attribute

5.23 attribute_arg

5.24 call_arg

5.25 tbind

5.26 array_index

5.27 do_loop_head

5.28 case_stmt

5.29 type_stmt

5.30 enumtype

5.31 Implicit Terms

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

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

2023-06-Apr :: Start.