MASR Summary

Brian Beckman

23 Apr 2023

Contents

1 Abstract

Abstract Semantics Representation (ASR) is a novel intermediate representation (IR)[1] for a new collection of LCompilers [*sic*].[2] ASR is agnostic to the particular programming language under compilation. Current compiler front-ends targeting ASR include LFortran[3] and LPython.[4] ASR is also agnostic to the back end. ASR currently targets LLVM, x86, C, and WASM[5]

Typical IRs encode semantics as decorations on the Abstract Syntax Tree, (AST)[6] ASR lifts *semantics* to the top level and expunges the syntax of the surface language as early as possible. Free of syntactical baggage, ASR optimizers are cleaner and faster than average, and ASR back ends are completely reusable. If syntax information is ever necessary, as with semantical-feedback parsing, such information will be encoded as decorations on the ASR, rather than the other way around.

Current specifications for ASR are written in ASDL,[7] a metalanguage similar in spirit to yacc but less rich, by design.[8] To build an LCompiler like LFortran or LPython, the ASDL grammar[9] for ASR is parsed, and a library in C++, libasr,[10] is generated. Compiler front ends call functions in this library to manipulate ASR and to emit code from the back ends.

ASDL has several deficiencies, and MASR [11] alleviates them. Chief among the deficiencies is the lack of type-checking. MASR adds a type system to ASR via Clojure *specs*.[12] MASR is a complete programming language in its own regard. It is, in fact, a Domain-Specific Language (DSL),[13] embedded in Clojure.[14]

We aim to replace ASDL with MASR and to integrate MASR with the LCompiler code base. When so integrated in the future, MASR will be called LASR.

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

This document may lag the Clojure code. It may also lag libasr, at least until MASR replaces ASDL. The document mirrors an ASDL snapshot.[9]

[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_syntax_tree
[7] https://en.wikipedia.org/wiki/Abstract-Type_and_Scheme-Definition_Language
[8] https://en.wikipedia.org/wiki/Yacc
[9] https://github.com/rebcabin/masr/blob/main/ASR_2023_APR_06_snapshot.asdl
[10] https://github.com/lfortran/lfortran/tree/c648a8d824242b676512a038bf2257f3b28dad3b/src/libasr
[11] pronounced "maser;" it is a Physics pun
[12] https://clojure.org/guides/spec
[13] https://en.wikipedia.org/wiki/Domain-specific_language
[14] https://en.wikipedia.org/wiki/Clojure

2 Summary and Cheat Sheet

2.1 Specs

Clojure specs are simple, one-parameter predicate functions. They double as *types*, constituting ASR's type system.[12, 15] The function `s/valid?`[16] recursively checks instances of a form against a spec. The function `s/conform`[17] produces a conforming instance from a candidate instance, or a value that passes `s/invalid?`.

2.2 Namespace-Qualified Keywords

Specs are named, defined, and registered via *namespace-qualified keywords* like `:masr.specs/nat`, in which the namespace is `masr.specs` and the keyword is `:nat`. All MASR spec keywords are registered in namespace `masr.specs`. The file `specs.clj` defines the namespace `masr.specs`. In the file `specs.clj`, a double-colon shorthand is available. For example, `::nat` in the file `specs.clj` is short for `:masr.specs/nat`.

2.3 Three Kinds of Specs

MASR has three kinds of registered specs:

simple specs — registered via `s/def`,[18] as in `(s/def ::bool boolean?)`

entity specs — defined but not registered via `s/keys`;[19] have required and optional attributes; examples below

multi-specs — defined via `s/multi-spec`,[20] registered via `s/def`; multi-specs have a distinguished *tag* attribute like tagged unions in C; examples below

MASR multi-specs are tagged collections of entity specs.

2.4 Full-Form

Full-form instances that are checked against specs are Clojure *hash-maps*:[21] collections of key-value pairs like Python dictionaries. For example,

```
;; key          value
{::term         ::intent,
 ::intent-enum 'Local}
```

In MASR, all keys in all hash-maps are namespace-qualified keywords. Such keys may have specs registered for them, or not. When a spec is registered for a key, automatic recursive type-checking is invoked.

[15] https://clojuredocs.org/clojure.spec.alpha
[16] https://clojuredocs.org/clojure.spec.alpha/valid_q
[17] https://clojuredocs.org/clojure.spec.alpha/conform
[18] https://clojuredocs.org/clojure.spec.alpha/def
[19] https://clojuredocs.org/clojure.spec.alpha
[20] https://clojuredocs.org/clojure.spec.alpha/multi-spec
[21] https://clojuredocs.org/clojure.core/hash-map

2.5 Sugar

In addition to the full form for every type, there are several shorter sugared forms. Sugar is defined by functions like `Integer` and `Integer-` that return instances in full-form. Sugar comes in three flavors, *light*, *heavy*, and *legacy*. See Section 2.9.

2.6 Terms and Heads

MASR defines *terms* and *heads* that describe the semantics of programs. Terms are top-most in the legacy ASDL grammar[9] — to the left of equals signs — and heads are at the bottom level — *vbar*-separated alternatives to the right of equals signs. There are only two levels.

The following tables summarize MASR via conforming examples, written in heavy sugar or legacy form.

Equally important are non-conforming examples. See (1) the body of the reference document, (2) `tests` in `specs.clj`, and (3) `deftest` in `core_tests` for many non-conforming examples.

Table 1: Atomic and Naked Specs: No Sugar

Spec	Predicate	Example
`::bool`	`boolean?`	`true`
`::float`	`float?`	`3.142`
`::int`	`int?`	`-1789`

Table 2: Top-Level *term-like* Specs, not in ASDL

Spec	Example
`::nat`	`(nat 42)`
`::identifier`	`(identifier 'boofar)`
`::identifier-set`	`(identifier-set ['a 'a])`
`::identifier-list`	`(identifier-list ['a 'a])`
`::identifier-suit`	`(identifier-suit ['a 'b])`
`::dimensions`	`(dimensions [[6 60] [1 42]])`

Table 3: Higher-Order Specs, Defined in Terms of Other Specs

Spec	Predicate	Example
`::symtab-id`	`::nat`	`(symtab-id 42)`
`::value-attr`	`::bool`	`(value-attr false)`
`::dependencies`	`::identifier-set`	`(dependencies ['a 'b 'c])`
`::symbolic-value`	TODO `empty?`	`(symbolic-value ())`
`::value`	TODO `empty?`	`(value ())`
`::type-declaration`	nilable `symtab-id`	`(type-declaration nil)`
`::varnym`	`::identifier`	`(varnym 'x)`

Table 4: *Term* Multi-Specs; Like Tagged Unions

Multi-Spec	**Term**	**Example**
`::asr-term`	`::dimension`	`(dimension [6 60])`
`::asr-term`	`::intent`	`(intent 'Local)`
`::asr-term`	`::storage-type`	`(storage-type 'Default)`
`::asr-term`	`::abi`	`(abi 'Source)`
`::asr-term`	`::access`	`(access 'Public)`
`::asr-term`	`::presence`	`(presence 'Required)`

Table 5: `::asr-term` Specs with Nested *Head* Multi-Specs

Term	**Head**	**Example**
`::ttype`	`::Integer`	`(Integer 4 [[6 60] [1 42]])`
`::ttype`	`::Real`	`(Real 8 [[6 60] [1 42]])`
`::ttype`	`::Complex`	`(Complex 4 [[6 60] [1 42]])`
`::ttype`	`::Logical`	`(Logical 1 [[6 60] [1 42]])`
`::symbol`	`::Variable`	`(Variable 42 x (Integer 4)...)`
`::expr`	`::LogicalConstant`	`(LogicalConstant true (Logical 4)`

2.7 Multi-Specs

Instance hash-maps that conform to multi-spec `::asr-term` are polymorphic. They have a tag attribute, fetched via `::term`, that must match a term `defmethod`.[22] The keyword, `::term`, doubles as an attribute key in the instance and as a function that fetches the value of the `::term` attribute from any instance hash-map.

For instance, the following example is a valid `::asr-term` in full-form; its `::term` attribute is `::intent`:

```
(s/valid? ::asr-term
          {::term        ::intent, ;; matches a defmethod
           ::intent-enum 'Local})  ;; specifies contents
```

where `::intent-enum` is a simple spec defined and registered via `s/def`:

```
(s/def ::intent-enum ;; #{...} is a Clojure set.
  #{'Local 'In 'Out 'InOut 'ReturnVar 'Unspecified})
```

Its `::term`, `::intent`, matches a term-`defmethod` below.

Here is another `::asr-term` in full-form, matching a term `defmethod` for `::abi`:

```
(s/valid? ::asr-term
          {::term          ::abi,  ;; matches a defmethod
           ::abi-enum      'Source ;; specifies contents
           ::abi-external false}) ;; specifies contents
```

where

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

Other `::asr-term` specs follow the obvious pattern. The `::term` attributes, `::intent`, `::abi`, etc., each match a `term` `defmethod`:

```
(defmulti term ::term) ;; ::term fetches the tag-value
(defmethod term ::intent    [_]  ;; tag-value match
  (s/keys :req [::term ::intent-enum])) ;; entity spec
(defmethod term ::dimension [_] ,,,) ;; tag-value match
(defmethod term ::abi       [_] ,,,) ;; tag-value match
(defmethod term ::ttype     [_] ,,,) ;; tag-value match
(defmethod term ::symbol    [_] ,,,) ;; tag-value match
;; etc.
```

Finally, the multi-spec itself is named `::asr-term`:

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

[22] https://clojuredocs.org/clojure.core/defmulti

2.8 Nested Multi-Specs

Contents of multi-specs can, themselves, be multi-specs. Such cases obtain when an `::asr-term` has multiple function-like heads. Examples include `::ttype`, `::symbol`, `::expr`, and `::stmt`.

The names of all multi-specs in MASR, nested or not, begin with `::asr-` and end with either `term` or `<some-term>-head`. Examples: `::asr-term` and `::asr-ttype-head`. There is only one level of nesting: terms above heads.

Here is the `::asr-term`-entity spec for `::ttype`. The *nested* multi-spec is named `::asr-type-head`.

```
(defmethod term ::ttype [_]
  (s/keys :req [::term ::asr-ttype-head])) ;; entity spec
```

where

```
(defmulti ttype-head ::ttype-head)    ;; tag fetcher
(defmethod ttype-head ::Integer ,,,) ;; tag match
(defmethod ttype-head ::Real    ,,,) ,,,
(s/def ::asr-ttype-head ;; name of the multi-spec
      ;; ties together a defmulti and a tag fetcher
      ;;             defmulti   tag fetcher
      ;;             ---------- ------------
      (s/multi-spec ttype-head ::ttype-head))
```

Here is a conforming example in full-form:

```
(s/valid? ::asr-term
          {::term ::ttype,
           ::asr-ttype-head
           {::ttype-head ::Real, ::real-kind 4,
            ::dimensions [[6 60] [1 42]]}})
```

Likewise, here is the `::asr-term` spec for `::symbol`:

```
(defmulti symbol-head ::symbol-head)
(defmethod symbol-head ::Variable [_]
  (s/keys :req [::symbol-head ::symtab-id ::varnym ,,,]))
(defmethod symbol-head ::Module   [_] ,,,)
(defmethod symbol-head ::Function [_] ,,,) ,,,
(s/def ::asr-symbol-head
      (s/multi-spec symbol-head ::symbolhead))
```

Here is a conforming example for `::Variable` in full-form, abbreviated:

```
(s/valid?
 ::asr-term  {::term ::symbol,
   ::asr-symbol-head
   {::symbol-head ::Variable
    ::symtab-id (nat 2)
    ::varnym    (identifier 'x)
    ::intent    (intent 'Local)
    ::ttype     (Integer 4 [[0 42])) ,,, }}
```

2.9 Light Sugar, Heavy Sugar, Legacy Sugar

Light-sugar forms are shorter than full-form, but longer and more explicit than *heavy-sugar*. Light sugar employs functions with keyword arguments and defaults. Heavy sugar employs functions with positional arguments and defaults only at the end of an argument list. Heavy-sugar functions are thus more brittle, especially for long specs with many arguments, with high risk of writing arguments out of order. *Legacy sugar* is order-dependent and compatible with `--show-asr` output from current LCompilers. It will be deprecated when MASR is integrated with LCompilers.

The names of light-sugar functions, like `Integer-`, have a single trailing hyphen. The keyword arguments of light-sugar functions are partitioned into required and optional-with-defaults. The keyword argument lists of light-sugar functions do not depend on order. The following two examples both conform to `::asr-term` and to `::ttype`:

```
(Integer- {:dimensions [], :kind 4})
(Integer- {:kind 4, :dimensions []})
```

The names of heavy-sugar functions, like `Integer` or `Variable--`, have either zero or two trailing hyphens. The difference concerns legacy. In addition to heavy sugar `Variable--`, MASR exports `Variable`, a macro for legacy `libasr --show-asr` syntax. Both produce identical full-forms.

For example, The following is heavy sugar for a *Variable*, representing the more progressive, desired form:

```
(Variable-- 2 'x (Integer 4)
            nil [] Local
            [] []  Default
            Source Public Required
            false)
```

and here is a legacy version of the same instance:

```
(Variable 2 x []
          Local () ()
          Default (Integer 4 []) Source
          Public Required false)
```

Notice no quote mark on the name of the variable. That's the way `--show-asr` prints it.

For specs where MASR heavy sugar and ASDL legacy are identical, like `Integer`, there is only one function with no trailing hyphens in its name.

Heavy-sugar functions employ positional arguments that depend on order. Final arguments may have defaults. For example, the following examples conform to both `::asr-term` and to `::ttype`:

```
(Integer)
(Integer 4)
(Integer 2 [])
(Integer 8 [[6 60] [1 42]])
```

2.9.1 Term Entity-Key Specs

For recursive type checking, as in `::Variable`, it is not convenient for terms to conform *only* to `::asr-term`. Therefore, we define redundant *term-entity-key* specs, like `::tterm`.

Entity-key specs for asr-terms are defined as follows:

```
(s/def ::ttype
  (s/and ::asr-term  ;; must conform to ::asr-term
         ;; and have tag ::ttype
         #(= ::ttype (::term %)))) ;; lambda shorthand
```

Because we have several such definitions, we write a helper function and a macro:

```
(defn term-selector-spec [kwd]
  (s/and ::asr-term
         #(= kwd (::term %)))) ;; lambda shorthand
(defmacro def-term-entity-key [term]
  (let [ns "masr.specs"
        tkw (keyword ns (str term))]
    `(s/def ~tkw    ;; like ::tterm
       (term-selector-spec ~tkw))))
```

Remember the name, `term-selector-spec`, of the helper function. We reuse it in the `def-enum-like` macro in the next section.

Here are some invocations of that macro:

```
(def-term-entity-key dimension)
(def-term-entity-key abi)
(def-term-entity-key ttype)
```

Here are some examples of extra conformance tests for sugared instances via term entity-key specs:

```
(s/valid? ::asr-term   (dimension []))       := true
(s/valid? ::asr-term   (dimension '(1 60))) := true
(s/valid? ::asr-term   (dimension '()))      := true

(s/valid? ::dimension (dimension []))       := true
(s/valid? ::dimension (dimension '(1 60))) := true
(s/valid? ::dimension (dimension '()))      := true

(s/valid? ::asr-term   (Integer 4))          := true
(s/valid? ::asr-term   (Integer 4 []))       := true

(s/valid? ::ttype      (Integer 4))          := true
(s/valid? ::ttype      (Integer 4 []))       := true
```

2.9.2 Enum-Like Specs

Entity-key specs are defined automatically for all *enum-like* terms via the `enum-like` macro:

```
(defmacro enum-like [term, heads]
  (let [ns "masr.specs"
        trm (keyword ns "term")      ;; like ::term
        art (keyword ns "asr-term") ;; like ::asr-term
        tkw (keyword ns (str term)) ;; like ::intent
        ,,,]
    `(do ,,,           ;; the entity-key spec
       (s/def ~tkw    ;; like ::intent
         (s/and ~art ;; like ::asr-term
                (term-selector-spec ~tkw)))
       ,,, )))
```

The macro, incidentally, defines and registers entity-key specs, as explained in the prior section.

Here are some examples of extra conformance tests for automatically defined term entity-keys for enum-like specs:

```
(s/valid?  ::intent  (intent 'Local))  := true
(let [iex (intent 'Local)]
  (s/conform ::asr-term iex)          := iex
  (s/conform ::intent iex)            := iex)
```

2.9.3 Term-Head Entity-Key Specs

For terms like `::symbol`, `::ttype`, and `::stmt`, which have multiple heads like `::Variable`, `::Integer`, and `::Assignment`, it is convenient to define redundant entity-key specs like the following:

```
(s/def ::Variable                    ;; head entity key
  (s/and ::asr-term                  ;; top multi-spec
    #(= ::Variable                   ;; nested tag
        (-> % ::asr-symbol-head ;; nested multi-spec
              ::symbol-head))))      ;; tag fetcher
(s/def ::Integer                     ;; head entity key
  (s/and ::asr-term                  ;; top multi-spec
    #(= ::Integer                    ;; nested tag
        (-> % ::asr-symbol-head ;; nested multi-spec
              ::ttype-head))))       ;; tag fetcher
(s/def ::Assignment                  ;; head entity key
  (s/and ::asr-term                  ;; top multi-spec
    #(= ::Assignment                 ;; nested tag
        (-> % ::asr-stmt-head        ;; nested multi-spec
              ::stmt-head            ;; tag fetcher
```

We define these with macros, `def-term-head--entity-key` and `def-ttype-and-head`. The definition of these macros are found in the file `specs.clj`. An example of conformance to `::Variable` is found above, in Section 2.9.

3 Abstract Interpretation

MASR is a full programming language in its own right. It is, in fact, a Domain-Specific Language (DSL) embedded in Clojure. An interpreter for MASR may be regarded as a reusable abstract interpreter for the surface languages, initially LFortran and LPython.