AFR-0

The Amlantis System Overview

Version 0.11

Markéta Lisová

August 8, 2020

Special Thanks To

Sabina Eva Lisová

Marie Strnadová

Ondřej Profant

Vojtěch Biberle

Contents

I Introduction 1

1 A Brief Introduction to the Amlantis System 3

1.1 The Amlantis Languages 4

II The Amlantis System Environment 5

2 Languages 7

2.1 Syntax Model 7

2.1.1 Syntax Objects 8

2.2 Readers 9

2.2.1 Root Reader Abilities 9

2.3 Expanders 9

III The Aml/Base Language 11

3 Syntax 13

3.1 The Aml/Base.Lang.Reader 13

3.1.1 Delimiters and Dispatch 13

3.1.2 Reading via an Extension 14

3.1.2.1 S-Expression Reader Language 15

3.1.2.2 Chaining Reader Language 15

IV The Amlantis System Tools 17

4 The Top-Level System or REPL – aml 19

5 The Compiler – amlc **21**

6 The Runtime System – amlrun **23**

7 The Debugger – amldebug **25**

8 The Profiler – amlprof **27**

9 The Builder – amlbuild **29**

10 The Book Workspace – amlbook **31**

Part I

Introduction

Chapter 1

A Brief Introduction to the Amlantis System

Amlantis System is a collection of specifications of programming and data definition languages and their related tools, runtimes and libraries.

Those specifications do not always require particular implementations, but attempt to avoid undefined behaviours as much as possible.

Amlantis also provides an open source reference implementation of those specifications, but it is indeed possible for other people to write their own implementations or forks[1] of this default implementation, probably focusing on optimising other aspects of the system, maybe exploring new options of future development to be pull-requested into the default implementation.

A Few Notes on the Name

Amlantis' name has quite some history. The project started being named *Coral*, but that collided with another language of a similar name, *CORAL 66*. Then it got renamed to *Gear*, but that again collided with another language of the same name, which seemed inactive at the time, *zippers/gear*. Than an idea was born and Aml was named *Amlantis*, which is whatever you want it to be. It could be a misspelling of *Atlantis*[2], it could be an acronym like *A ML Language*, or maybe even something like *A ML Language And Neat Technology Improvement System*, or maybe *Caml* without the *C*. For the meaning of the cryptic *ML* part, search for the *Standard ML* or *OCaml*.

[1] Forks and pull requests are preferred way of help!

[2] Intentionally – because otherwise, it would be named Atlantis, but there is already a city of that name.

1.1 The Amlantis Languages

The Amlantis System is a home to more than only one programming language – in fact, it can be home to any number of languages, ones that Amlantis users can either extend from existing languages create or even create new ones entirely. Every language build within the Amlantis System shares a common type system and value models, and thus are interoperable.

The minimal Amlantis System contains these languages:

- `Aml/Base`, a minimal Lisp-like[3] language that is easy to parse and limited in functionality.
- `Aml/Core`, an ML-like[4] language that is easier to read and harder to parse, fully featured in functionality.
- `Aml`, an ML-like[5] language that extends `Aml/Core`, is easier to read and the hardest one to parse, also fully featured in functionality, and introducing more grammar than its parent `Aml/Core`.

More languages may be added to this list over time.

[3] One might say that `Aml/Base` is a grandchild of Racket and Clojure.
[4] One might say that `Aml/Core` is a grandchild of OCaml, SML and F♯.
[5] One might say that `Aml` is a grandchild of `Aml/Core`, Haskell and Ada.

Part II

The Amlantis System Environment

Chapter 2

Languages

Amlantis System provides multiple interfaces to talk to it with. Those interfaces would be:

- A REPL console, where user's text input is immediately read, evaluated, printed and requested again.
- Source code, where user's text input is stored for reuse in general, usually to either do some scripting, or building modules and complex applications.
- Bytecode or native (machine) code files, which originate from textual source code.

More ways to interact with the Amlantis System might be possible in the future.[1]

Either way, for now, text-based interface to all Amlantis System tools require it to have a component that is able to read the text. We call them *Language Readers* (or shortly just *Readers* for the rest of this chapter). After reading the text, it has to be given meaning and transformed into a system of values that the Amlantis System understands by *Language Expanders* (or shortly just *Expanders* for the rest of this chapter). The result of that can be eventually evaluated.

2.1 Syntax Model

The syntax of programs in the Amlantis System is determined by the following process:

- A *read* pass, which comprise processing a text source port into a syntax object. Such syntax object does not contain any bindings. The read pass is implemented by Readers (§2.2).

[1] Some ideas for that would include natural speech interface (at least in English).

- An *expand* pass, where the syntax object from previous pass is transformed into a syntax object with full program information. All bindings are defined in such syntax object. The expand pass is implemented by Expanders (§2.3).

A fully-featured language provides implementation for both read and expand passes. Such implementation may as well reuse implementation of another language. A language may as well provide just its Reader, in case the resulting value can be expanded by the expander of `Aml/Base`. Alternatively, a language may provide just its Expander, providing bindings that may differ from those in `Aml/Base`.

2.1.1 Syntax Objects

A *syntax object* is a combination of an arbitrary Amlantis System value with lexical information, source-location information and syntax properties.[2]

Although syntax objects may contain any value, the expand pass (§2.1) will need to be able to handle the value and any value it itself contains. Therefore, it is recommended to limit the values embedded into syntax objects to the following types:

- Atoms:
  - Booleans.
  - Numbers.
  - Characters.
  - Strings.
  - Bytes and byte strings.
  - Symbols and Keywords.
  - Unit and Undefined.
- Basic collection and composite objects:
  - Tuples.
  - Pairs and Lists.
  - Vectors.
  - Dictionaries.

[2] Racket, where inspiration for this mechanism comes from, also has tamper status within those syntax objects. We may decide to add that later.

2.2 Readers

Every Reader needs a source port to read from. Such a port could be console input for the REPL, a file on local filesystem, a TCP connection, socket, or just an in-memory string[3]. Reading program source starts with a top-most Reader, which we may call the *Root Reader*, which is implicitly (unless defined otherwise) `Aml/Base.Lang.Reader`.

Readers are required to implement two methods: `read` and `read-syntax`.

The `read` method must accept a source port as its only argument, and it can return any type of value.

The `read-syntax` method must accept a source name argument along with second argument of the source port, and it must return either the `System/IO.Eof` value, or a `System/Lang.Syntax_Object` value.

The Root Reader's `read-syntax` method is used on the top-most initial port.[4]

2.2.1 Root Reader Abilities

The Root Reader can recognise basic delimiters, such as all whitespace and newlines, which aid it in switching to another source code language. Given that the Root Reader is in fact `Aml/Base.Lang.Reader`, its abilities are not limited to these.

Delimited by those, it can recognise the following forms:

- Shebang, such as `#!/usr/bin/env aml`, to point to the entry-level Amlantis System tool.
- Reader switch, `#reader` *name* ..., which specifies a reader to use for the following forms, and switches to it. The *name* should be a simple identifier.[5]
- Language switch, `#lang` *name* ..., which expands right away into `#reader` *name*`.Lang.Reader` ....[6] The *name* should be a simple identifier.[7]

2.3 Expanders

[3] More about strings later.
[4] Might be different in future, when we allow it to use `read` instead. That is not very useful now though.
[5] Specification for identifiers will be provided later.
[6] It may expand to other forms as well in certain order, due to be defined later.
[7] Specification for identifiers will be provided later.

Part III

The `Aml/Base` Language

Chapter 3

Syntax

The syntax of `Aml/Base` is based on Lisp-1 and blatantly borrowing from languages such as Scheme, Racket or Clojure.

3.1 The `Aml/Base.Lang.Reader`

3.1.1 Delimiters and Dispatch

Along with whitespace (as defined by Unicode ”White_Space” property), the following characters are delimiters:

```
( ) [ ] { } " ' ` , ~ ;
```

The # character has got a special meaning, determined by the following character or characters; see below for details.

After skipping whitespace, the `Aml/Base` reader dispatches based on the next character or characters in the source port this way:

`(`	starts a pair or list.
`[`	starts a pair, list, or array.
`{`	starts a structure.
`)`	matches `(` or raises error.
`]`	matches `[` or raises error.
`}`	matches `{` or raises error.
`"`	starts a string.
`'`	starts a quote.
`` ` ``	starts a quasiquote.
`,`	starts a (possibly splicing) unquote.
`~`	starts a keyword.
`;`	starts a line comment.
`#y`	true.
`#n`	false.
`#t` or `#T`	true.
`#f` or `#F`	false.
`#(`	starts a set.
`#[`	starts a vector.
`#[`	starts a vector.
`#\`	starts a character.
`#"`	starts a byte string.
`#:`	starts a symbol.
`#'`	starts a syntax quote.
`` #` ``	starts a syntax quasiquote.
`#,`	starts a syntax (possibly splicing) unquote.
`#<<`	starts a string.
`#|`	starts a block comment.
`#;` or `#_`	starts an S-Expression comment.
“`#! `”	starts a line comment; note that the space is necessary.
`#!/`	starts a line comment.
`#!`	may start a reader extension use; see Reading via an Extension.
`#reader`	starts a reader extension use; see Reading via an Extension.
`#lang`	starts a reader extension use; see Reading via an Extension.
#other	starts a handler defined in current readtable or raises error.
otherwise	starts a symbol.

3.1.2 Reading via an Extension

When the reader dispatches on the `#reader` form, it recursively applies another reader to the current source port.

First, the reader recursively reads the next datum after `#reader`, and uses it as path to the another reader. Such reader is then loaded, and `read` is used when this reader is in `read` mode, or else, `read-syntax` is used when this reader is in `read-syntax` mode.

The `#lang` reader form is similar. It must be followed by a single whitespace character (preferably a single space, ASCII 32), and then followed by an identifier form. The complete form is then terminated by a new line, or end-of-file. A sequence `#lang` *name* is equivalent to `#reader` *name*`.Lang.Reader`.

For compatibility with e.g. R⁶RS, `#!` is an alias for `#lang` followed by a space when it is followed by alphanumeric ASCII, `+`, `-` or `_`.

3.1.2.1 S-Expression Reader Language

`#lang s-exp` *path*

3.1.2.2 Chaining Reader Language

`#lang reader` *path*

Part IV

The Amlantis System Tools

Chapter 4

The Top-Level System or REPL – `aml`

This chapter will describe the top-level system for the Amlantis System, which serves as universal entry point for running programs designed for this system.

Without input files specified in arguments, it permits interactive use of the Amlantis System through a mechanism known as read-eval-print loop (REPL). With this mode, the system repeatedly reads user's input, evaluates it with the context of every previous input of the same REPL session, prints the inferred type of the result of such evaluation along with the text representation of the result, if any, and repeats.

Chapter 5

The Compiler – `amlc`

In this chapter, we're going to describe the Amlantis System Compiler `amlc`, which compiles source code files to bytecode files. The compilation process involves reading through the input source files, expanding them by evaluating the read programs by using the specified language, and serializing the type system's state along with modules' state to bytecode files, which are then to be run by `amlrun` (§6) or `aml` (§4).

Chapter 6

The Runtime System – `amlrun`

This chapter will describe the `amlrun` tool, which reads files output by `amlc` (§5) and executes a specified executable[1].

[1] Due to be described later, but those could be e.g. a specific function, or a script file.

Chapter 7

The Debugger – `amldebug`

In this chapter, we'll describe the Amlantis System's own debugger tool.

Chapter 8

The Profiler – `amlprof`

This chapter will describe the Amlantis System's profiling tool.

Chapter 9

The Builder – `amlbuild`

In this chapter, we'll describe the Amlantis System's builder tool.

Chapter 10

The Book Workspace – `amlbook`

This chapter will describe the Amlantis System's "Book Workspace" tool, which is a planned addition to the whole toolchain and supposed to be accompanied by a GUI. The purpose will be to work with source texts that intertwine program source code in snippets with rich text documents, and ability to selectively execute these program source codes, seeing results right below each snippet.