αML user manual, v0.3

Matthew R. Lakin

29 March 2010

1 Disclaimer

This manual and the αML implementation itself are works in progress! There may well be bugs—if you find one, please submit a bug report (or other feedback) to

`alphaml.feedback@googlemail.com`.

2 Introduction

αML is a functional-logic programming language for easy prototyping of inductively-defined systems such as type systems and operational semantics of programming languages and calculi. The design and semantics of the language are detailed in [3]. This document describes the language from the perspective of an end-user, with sections on installation and interaction with the toplevel loop. It also provides a reference for the ASCII concrete syntax of the core language and the various syntax extensions which are defined in terms of the core. Code examples in `teletype font` represent actual concrete syntax of the language, whereas metavariables in *italic font* range over syntactic constructs.

The language and its theory were developed as joint work with Andrew Pitts, and funded by UK EPSRC grant EP/D000459/1.

2.1 Installation

The αML source code distribution can be downloaded from

`http://www.cl.cam.ac.uk/~mrl35/`.

It has been successfully compiled under Linux, Mac OS X and Windows Vista (using both the MSVC and `cygwin` ports of OCaml). There is also a binary distribution available for Windows users (32-bit only).

The following are required to compile the system from source:

- **Objective Caml**, version 3.10.2 or greater.
  `http://caml.inria.fr/download.en.html`
- **GNU Make**, version 3.80 or greater.
- **OCamlMakefile.**
  `http://www.ocaml.info/home/ocaml_sources.html`
- **The ocamlgraph library.**
  `http://ocamlgraph.lri.fr/`
- **The vec library.**
  `http://luca.dealfaro.org/Vec-Extensible-Functional-Arrays-for-Ocaml`

To compile the system:

1. First ensure that the OCaml compilers and GNU Make are installed correctly.
2. Compile the ocamlgraph and vec libraries and install them, either manually or using a tool such as findlib. It is simplest to install the library files into the default OCaml library directory and copy the OCamlMakefile into the source directory.
3. If this is not possible then the libraries can be placed into any directory (provided that the files are all together) and the OCamlMakefile can be placed in any directory (which need not be the same as the location of the compiled library files).
4. If the libraries and the OCamlMakefile were installed in alternative locations, the `OCAMLMAKEFILE` and `LIBDIR` variables at the beginning of the Makefile to point to the correct locations (`OCAMLMAKEFILE` should be the path including the file name, whereas `LIBDIR` should be the path to the directory containing the library files).
5. It should be possible to compile the αML interpreter now—do

   ```
   make native-code  or  make nc
   make byte-code  or  make bc
   ```

   to build native- and byte-code versions of the interpreter respectively. To clean the directory, do `make clean`.

If compilation fails, the most likely reason is that the libraries cannot be found because paths are not set correctly. If compiling native-code under Windows, consult the `README.win32` file from the OCaml distribution to ensure that you have all of the necessary ingredients for native-code compilation. In particular, if the final linking phase fails with an error about `flexlink`, try running the final `ocamlopt` command manually but without any `/ccopt` argument.

```
$ ./aml
       _        _            __  __ _
  __ _| |_ __ | |__   __ _|  \/  | |
 / _` | | '_ \| '_ \ / _` | |\/| | |
| (_| | | |_) | | | | (_| | |  | | |___
 \__,_|_| .__/|_| |_|\__,_|_|  |_|_____|
        |_|

Version 0.3

>
```

Figure 1: αML interpreter prompt

2.2 Toplevel loop

Having successfully compiled the system, the αML interpreter is invoked by running the `aml` command, which produces a prompt for user input liek that shown in Figure 1. User input is terminated by dual semicolons ';;', so the input may span multiple lines. A single input phrase at the toplevel may take one of four forms.

1. **An αML expression to evaluate.** If the user enters an expression e, the expression is typechecked and then evaluated (see Section 3 for concrete syntax accepted by the interpreter). The operational semantics of αML involves non-deterministic search, which means that the evaluation of an expression might produce multiple values. If a value is computed, the value and the type are then printed to the terminal—if there are more branches of computation waiting to be explored then there is a prompt to either keep searching or stop:

   ```
   Type ',' (then Enter) to look for more answers, or '.'  to stop.
   ```

   The state produced by evaluating the expression is only stored when the user selects '`.`' to stop searching, or when the final value is found. If one selects to keep searching past the final value, or there are no values at all (i.e. the expression fails finitely), then the response "`no`" is printed to the terminal and any new constraints generated during the evaluation are discarded.

2. **An expression to evaluate and bind.** A `let`-binding has the syntax "`let` x `=` e" and is dealt with in largely the same way as a "normal" expression e. The difference is that when a value is accepted (either explicitly by the user, or implicitly as the single value computed by a deterministic computation) then it is bound to the variable x. The result of evaluating e can then be referred to using x in subsequent expressions, until the binding is shadowed by another binding to x or the compiler state is reset.

3. **A type declaration.** Nametypes and datatypes can be declared by the user. Nametypes are used to represent object-language names. A single command can declare multiple types of the same kind. The syntax for each of these declarations is as follows:

   - `nametype nty_1 and ⋯ and nty_j;;`
   - `datatype dty_1 = K_11 of` T_{11} `|` $\cdots$ `| K_1n of` T_{1n}
     `and` $\cdots$
     `and dty_j = K_j1 of` T_{j1} `|` $\cdots$ `| K_jm of` T_{jm}`;;`

   The syntax of datatype declarations follows that of ML datatype declarations very closely. A particular datatype declaration is mutually recursive, so the types on the right-hand sides may refer to any of the datatypes on the left, or any previously declared types. The grammar of types T is presented in Section 3.2.

   The identifiers for nametype and datatypes must begin with a lowercase letter and data constructors must begin with an uppercase letter—the lexical conventions of the language are described in Section 3.1. The processing of a type declaration may fail if one of the identifiers has already been used as a nametype, datatype or data constructor.

   There is also another kind of type declaration, for *relation types*, which are discussed in Section 3.6 below.

4. **An interpreter directive.** Any user input which begins with the '`%`' character is parsed as a directive to the interpreter. The following directives are accepted by the interpreter:

   - `%breath`: switches the non-deterministic branching search procedure to breadth-first mode (this is the default setting).
   - `%constraints`: prints a representation of the set of all constraints which have been processed by the interpreter.
   - `%debug on/off`: toggles debug output (off by default).
   - `%depth`: switches the non-deterministic branching search procedure to depth-first mode.
   - `%help`: prints out the list of interpreter directives.
   - `%quit`: exits the toplevel loop.
   - `%reset`: resets the internal state of the interpreter, i.e. the generated logic variables, type declarations, variable bindings and constraints.
   - `%use "foo"`: loads the file "`foo`" and runs all of the commands contained therein (which might include more directives, including more "`use`" directives).
     Note: if there is a type error which prevents one of the commands within the file from running, any earlier changes made to the state of the interpreter, for

example by a type declaration, will persist. If the file is used again, the repeated type declaration will fail, so a "`%reset`" must be done before the file is re-run.

- `%walk on/off`: by default, the values produced by the evaluation of expressions are "walked", which actually performs the substitutions implied by the constraint set. This directive toggles this behaviour.

3 Language syntax

In this section we present the lexical conventions and concrete syntax of the αML language as accepted by the interpreter, along with a discussion of the numerous defined extensions and details of their implementation within the compiler.

3.1 Lexical conventions

Most of the identifiers used in αML are lowercase identifiers (*lcase*) which means that they match the regular expression

lc [*lc* | *uc* | *digit* | _ | ']*

i.e. the initial lowercase letter (*lc*) is followed by zero or more lowercase letters, uppercase letters (*uc*), digits (*digit*), underscores ('_') or primes ('′'). These identifiers are used for αML value identifiers, as well as nametypes and datatypes. Data constructors, however, are uppercase identifiers (*ucase*) which match the (very similar) regular expression

uc [*lc* | *uc* | *digit* | _ | ']*

i.e. they begin with an uppercase letter.

The following keywords are reserved and therefore cannot be used for value identifiers.

```
as    and    case    datatype    distinct    exists    fresh    fn    fun
in    let    nametype    no    of    off    on    rec    relation    some
      unbind        unbind_fresh        unit        where        yes
```

The grammar also uses tokens *num* (which stands for an integer constant) and *alpha* (which stands for one or more lowercase letters).

3.2 Core language

The syntax of the core language is defined by the grammar in Figure 2. From low to high, the precedences of the infixed operators are *branch*, *freshness*, *equality*, *projection*, *abstraction*. The expression forms with meta-level binding are the `let`-bindings, anonymous and recursive functions and case analysis. As usual in nominal abstract syntax, the abstraction

e	::=	*lcase*	value identifier
	|	`()`	unit
	|	*ucase* e	data
	|	`<`e`>`e	abstraction
	|	`(`e`,`$\cdots$`,`e`)`	tuple
	|	`fn (`*lcase*`:`T`) -> `e	anonymous function
	|	`fun` *lcase*`(`*lcase*`:`T`):`T `=` e	recursive function
	|	`let` *lcase* `=` e `in` e	let-binding
	|	`case` e `of` *ucase lcase* `->` e	case analysis
		`|` $\cdots$ `|` *ucase lcase* `->` e	
	|	e`.`*num*	projection
	|	$e \cdots e$	application
	|	e `=` e	equality constraint
	|	e `#` e	freshness constraint
	|	e `||` $\cdots$ `||` e	branch
	|	`some` T	logic variable generation
	|	`fail` T	fail
	|	`yes`	succeed
	|	`(`e`)`	bracketed expression
	|	`?`*alpha*	existential variable

Figure 2: Syntax of the αML core language

term-former `<`e`>`e' models the binding of a name in a term but is not itself a meta-level binder.

The "`some` T" syntactic construct returns a new existential variable each time is is invoked. This is as oppposed to the existential quantifier used in [3], which is available as a defined construct (see Section 3.5 below). The result of evaluating "`some`" is an existential variable, represented as `?a`, `?b`, `?c` etc. When the result of a program is pretty-printed, the interpreter attempts to hide these by substituting for them according to the constraint environment (unless the "`%walk`" directive has been disabled). However, in some cases these variables may still be visible in the pretty-printed output, even if "`%walk`" is enabled. If this is the case, the "`%constraints`" directive can provide extra information on the result of the computation.

Largely for debugging purposes, it is possible to enter these variables directly in expressions at the command prompt. For example, the existential variable `?c` refers to the third existential variable generated by the interpreter (since the last `%reset`). If a variable is typed in which has not been generated yet (e.g. `?z` when only three existential variables have been generated) the interpreter signals an error.

T	::=	*lcase*	type identifier
	|	`prop`	proposition type
	|	`unit`	unit type
	|	`[`T`]`T	abstraction type
	|	T `*` $\cdots$ `*` T	product type
	|	T `->` T	function type
	|	`(`T`)`	bracketed type

Figure 3: Syntax of αML types

3.3 Types

The αML type system is simple and monomorphic—see [3] for a brief discussion of the typing rules for the non-standard constructs. The grammar of types is presented in Figure 3. In particular, for an abstraction `<`e`>`e' the type of e should be a *nametype* and e' should have an *equality type*. Equality types are a subset of types which have a decidable notion of equality, so that α-equivalence constraints between terms of these types can be checked for satisfiability. Equality types are built from the unit and name types by tupling, abstraction and data construction. In particular, existential variables may only be generated at equality types, and it is worth pointing out that *the only values of nametypes are existential variables*.

The type inference problem is rendered trivial by the requirement that programs be annotated with type information at certain points, such as the arguments of recursive functions and the generation of new existential variables. The failing expression (`fail` T) is also annotated with a type, so that expressions of any type can be made to fail finitely.

A type identifier *lcase* could refer to a nametype or a datatype, depending on the types that have been declared already. The abstraction type-former has higher precedence than the function type, which is in turn higher than the product type. Function types associate to the right, as usual.

3.4 Comments

αML comments are ML-style multi-line comments. They are opened by "`(*`" and closed by "`*)`". Comments can appear anywhere and they can be nested, but nested comments must be terminated properly.

3.5 Language extensions

The implementation supports various language extensions on top of the language defined in [3], most of which are defined straightforwardly in terms of the core language.

- **Wildcard variables.** The bound variable in a `let`-expression may be replaced by the wildcard '_' which discards the value instead of binding it. This can be used to

simulate sequential evaluation.

- **Conjunction.** Two expressions can also be evaluated sequentially using the syntax "e `&` e". This is intended to be suggestive of logical conjunction, for use in the specification of inductive definitions (see Section 3.6 below).

- **Function definition.** The definition and `let`-binding of anonymous and recursive function values is simplified by these syntactic sugars.

  `let` [`rec`] *lcase lcase* $\cdots$ *lcase* `=` e `in` e

  The value identifiers for the arguments are bound as normal—in the recursive case (when the optional `rec` is included) the name of the function is also bound in its body.

- **Existential quantification.** The scoped existential quantification operator used in [3] is available in αML using the syntax "`exists` *lcase*`:`T `in` e". It is definable in terms of `let`-binding and the `some` operator. The value identifier is bound in the body of the quantifier.

- **Fresh name generation.** Fresh name generation "`fresh nty`" is not definable in terms of the core language. It is implemented as an additional core feature, as outlined in [3].

- **Distinct names.** Given a list of variables `x_1`,...,`x_n` there is a defined shorthand to assert that they are pairwise distinct: "`distinct (`e`,`$\cdots$`,`e`)`". This expression only type-checks if all of the variables are of nametypes (they do not all need to be the *same* nametype).

- **Unbinding.** The language provides standard built-in deconstructors for tuples and data terms. We provide a deconstructor for abstractions

  `unbind` e `as <`*lcase*`>`*lcase* `:` `[`*lcase*`]`T `in` e

  which generates metavariables to stand for the bound name and the body of the abstraction and uses an equality constraint to relate these to the term being unbound. These are referred to by value identifiers which are bound in the body of the expression. The type annotations are required because the `some` operator is used to generate the metavariables which itself requires a type annotation. No freshness constraints are produced, but can be added manually to assert that the bound name should be locally fresh for some set of names.

- **Fresh unbinding.** This construct is very similar to the previous unbinding operator, but uses `fresh` (as opposed to `some`) to generate the variable standing for the bound name.

  `unbind_fresh` e `as <`*lcase*`>`*lcase* `:` `[`*lcase*`]`T `in` e

This means that the bound name is globally fresh for all metavariables that have been created so far, and provides a flavour of the "generative unbinding" functionality provided by FreshML [5].

3.6 Inductive definitions

αML includes a sub-language of *inductive definitions* over terms involving binders. These are expressed in an ASCII designed to resemble inference rules (and which are reminiscent of the concrete syntax of the `ott` language [4], but without that system's support for concrete object-language syntax). The translation of schematic inductive definitions into αML recursive functions is described informally in [3].

An expression corresponding to an inductive definition is delimited by double braces: "{{" at the beginning and "}}" at the end. Within these delimiters there must be one or more *inductive definitions*, each of which takes the form

$$\frac{e}{ucase\texttt{(}p\texttt{,...,}p\texttt{)}} \quad \texttt{[}lcase\ \texttt{where x1,}\cdots\texttt{,xn:}T_1\ \texttt{, }\cdots\texttt{ ,y1,}\cdots\texttt{,ym:}T_m\texttt{]}$$

The uppercase symbol is a *relation symbol*. These are declared using the following syntax.

$$\texttt{relation R\_1 <: }E_1\texttt{ and }\cdots\texttt{ and R\_n <: }E_n$$

When a relation symbol `R_i` is applied to an expression of the appropriate equality type E_i, the resulting term is of a special datatype "`rel`", which is reserved for terms corresponding to these instances of inductively defined relations. The premise of every inference rule must have type `rel`. The conclusion of the rule contains of *patterns* p (enclosed by parentheses), defined as in [3]:

p	::=	*lcase*	variable
	|	`()`	unit
	|	*ucase* p	data
	|	$(p,\cdots,p)$	tuple
	|	<*lcase*>p	abstraction

On the right-hand side of the middle line the lowercase label is *optional*—it is for documentation only. If it is omitted, the subsequent "`where`" keyword must be omitted also. A type annotation "`xi:`ety_i" is required for any value identifier appearing in the rule which is scoped to within that rule (it is possible to write rules with free variables, e.g. in a definition which is parameterised by an argument). If neither a label nor any type annotations are required for a particular rule, then the square brackets should be omitted completely.

Although the syntax is intended to be suggestive of inference rules, the parser is not actually whitespace dependent. However, the relative position of the elements *is* important. In particular, if the list of type annotations is long it can be split over multiple lines but the conclusion of the rule must still be on the line below the end of the list of type annotations. The files in the `examples/` directory of the αML distribution contain examples of code making heavy use of the inductive definition syntax.

3.7 Ground trees

The αML system implements the translation of ground abstract syntax trees described informally in [2]. This syntactic form is delimited by "`[|`" and "`|]`", and the syntax of the ground trees themselves is given by the following grammar.

g	::=	*lcase*	name
	|	`()`	unit
	|	*ucase* g	data
	|	`(`g`,`$\cdots$`,`g`)`	tuple
	|	`<`*lcase*`:`*lcase*`>`g	abstraction

The concrete syntax is similar to a fragment of the core language grammar (Figure 2): the only difference is that every variable in abstraction position must have a type annotation of the *nametype* of that particular bound name. This is because the translation uses existentially-quantified variables to represent the bound names of the ground tree.

The crucial difference between the sub-language of ground trees and the rest of the language is the way that value identifiers are interpreted. Here, the lowercase value identifiers (which must all be of nametypes) are interpreted as *distinct concrete names*, like the atoms used in FreshML [5]. The translation into core αML ensures that these expressions are operationally equivalence precisely when the corresponding ground trees are α-equivalent. For example, the representation of the λ-term $\lambda a.\, \lambda b.\, (a\, b)$ is

```
[| Lam <a:var>(Lam <b:var>(App (Var a,Var b))) |].
```

A note is required on the treatment of *free names* of ground trees. In order to achieve the correspondence between α- and operational equivalence, the free names of a ground tree g must be interpreted as *free value identifiers* of its encoding. Therefore, if a ground tree has free names then *existential variables with the same identifiers must have already been declared*, all at nametypes. For example, to encode the open λ-term $\lambda a.\, (a, b)$ would require the following toplevel interaction.

```
let b = some var;;
[| Lam <a:var>(App (Var a,Var b)) |].
```

It is worth pointing out that the evaluation of an encoded ground tree may fail finitely. This is because existential variables corresponding to the free names must have been declared already, so there could already be constraints between these names. The translation of ground trees requires all of the names appearing in the tree to be pairwise distinct, so if any two of the free names have already been constrained to be equal then the expression cannot be evaluated successfully. For example, the following interaction leads to finite failure:

```
let b = some var;;
let c = some var;;
b = c;; (* constrain two names to be equal *)
[| Lam<a:var>(App(Var b,Var c)) |];;
```

4 Examples

The `examples/` directory of the source distribution contains a few examples of systems defined in αML:

- `lam.aml`: "free variable" and reduction relations for λ-calculus.
- `peano.aml`: an implementation of Peano numbers and some trivial numeric functions (no binding features used).
- `poplmark.aml`: solution to part 3 of the POPLmark challenge [1], for the F_{sub} language *without* records.

References

[1] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLmark Challenge. In *Proceedings of TPHOLs 2005: the 18th International Conference on Theorem Proving in Higher Order Logics (Oxford), LNCS 3603*, pages 50–65, August 2005.

[2] M. R. Lakin. Representing names with variables in nominal abstract syntax. TAASN 2009 Workshop, 2009.

[3] M. R. Lakin and A. M. Pitts. Resolving inductive definitions with binders in higher-order typed functional programming. In G. Castagna, editor, *18th European Symposium on Programming (ESOP '09)*, volume 5502 of *Lecture Notes in Computer Science*, pages 47–61. Springer, Mar 2009.

[4] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. In *Proceedings of ICFP 2007: the 12th ACM SIGPLAN International Conference on Functional Programming (Freiburg)*, page 12pp, October 2007. 12pp.

[5] M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programming with binders made simple. In *ICFP 2003 Proceedings*, pages 263–274. ACM Press, 2003.