ML–Lucid, an intensional functional language

John A. Plaice
Department of Computer Science, University of Ottawa
150 Louis Pasteur, Ottawa, Ontario, CANADA K1N 6N5
e-mail: plaice@csi.uottawa.ca

Abstract

In this paper, we define the ML–Lucid programming language, a generalization of Lucid and of the functional part of ML. ML–Lucid is a multi-dimensional language, and dimensions can be defined to be any ML data type. All functional ML objects can be used as the atomic objects of Lucid.

The semantic of ML–Lucid are given using a Kripke semantics, where the universe of possible worlds is the arbitrary cross-product of ML data types.

All of Lucid's intensional operators can be defined in ML–Lucid, and new ones can easily be defined.

1 Introduction

There have been several proposed extensions to Lucid to make it multi-dimensional, to generalize the concept of stream. The two most recent are Du's mLucid[3] and Faustini and Jagannathan's Indexical Lucid [4].

Such extensions allow one to treat the elements of an array in a manner orthogonal to computations over time, using an elegant syntax, and have the advantage of not restricting one to finite arrays.

The semantics of these languages are all based on Kripke models, where the universe of possible worlds is N^ω. In any given program, only a finite number of dimensions will be used, and all the other dimensions are assumed to have value 0.

However, there is no reason that the universe of possible worlds could not be more complex. After all, Tennent [8] uses possible worlds to give an elegant semantics for concepts such as local-variable declarations and Allwein and Dunn [1] use possible worlds to give a semantics to linear logic. In both cases, the universe is more complex.

In general, a possible world can be considered to be some sort of context: time, place, state, point in a computation, etc., or any combination thereof. For each of these components, one would wish to associate a normal data type. So the universe should be the arbitrary cross-product of definable data types.

All of the existing Lucid interpreters also do not allow higher-order functions, the most important reason being lack of an elegant way of implementing them. However, Wadge [9] gave an elegant way of implementing Higher Order Lucid, using a generalization of the place codes which are used in most Lucid interpreters.

At the same time, standard functional languages, such as ML and Haskell, have nice typing facilities and higher-order functions. However, they do not have natural ways of defining multi-dimensional objects. For example, Ashcroft [2] shows that one has to stand on one's head to define multi-dimensional objects in Haskell.

ML–Lucid is a combination of the two languages. The basic objects of the language are ML functional objects, and the dimensions can be any ML data type. ML–Lucid is being implemented in ML, and will allow higher-order functions.

2 The ML–Lucid language

ML–Lucid is a language which is designed to be, with little change, used on top of a functional language. The current implementation is based on ML, but one could imagine a different implementation.

The link between ML–Lucid and the host language is done by defining the available types, constants and functions which are available to ML–Lucid from the host language. Essentially, a multi-sorted signature is defined, which can then be used by the ML–Lucid program.

2.1 Syntax

There are therefore two parts to an ML–Lucid program: the definition of a signature, and an ML–Lucid expression.

The global syntax for a program is therefore:

program ::= *signature* **%%** *expression*

The signature is easily defined. In what follows, *id* will always stand for an identifier.

signature ::= *sigdec*+
sigdec ::= `type` *id* `=` `external` *id*
| `constant` *id* : *type* `=` `external` *id*
| `function` *id* : *type* `=` `external` *id*

`type` is used to define sorts, `constant` to define constant symbols, `function` for functional symbols. For each identifier, the identifier following `external` refers to the name in the host language. Here is an example:

```
type     integer = external int
function plus:  (integer*integer -> integer) = external int_plus
function times: (integer*integer -> integer) = external int_times
```

Once the sort has been defined, then an ML–Lucid expression can be defined. ML–Lucid being a member of the ISWIM [7] family of languages, the free variables of an expression are its input. Also, sets of mutually recursive functions can be used to define local variables and functions.

The abstract syntax for an expression is as follows:

expression (E)	::=	k	constant
		id	variable
		$id.d$	intensional variable
		$(E, \ldots, E)$	tuple creation
		`#i` E	component selection
		`if` E `then` E `else` E	
		$E\ E$	function application
		`let` *decs* `in` E `end`	

The intensional variable $id.d$ means a variable which is annotated with a dimension or a dimension variable (d). Since this is an abstract syntax, we do not have to worry about ambiguity. Function application ($E\ E$), is left-associative in the concrete syntax.

Declarations are treated in the order in which they appear. One can define a new type for dimensions, variables and constants to denote dimensions of a particular type, as well as declare functions, intensional functions and variables.

decs	::=	*dec*+	
dec	::=	`dimtype` $id = (type, exp, func)$	
		`dimvar` $id : id$	
		`dimension` $id : id$	
		`fun` $id.d\ id = E$	intensional function
		`fun` $id\ id = E$	function declaration
		`val` $id = E$	variable declaration

The three arguments for `dimtype` are:

- The type of all dimensions of `dimtype` *id* is *type*.
- The origin of all dimensions of `dimtype` *id* is *exp*.
- The function *func* converts an element of *type* into an integer so that an expression can be hashed.

A `dimtype` declaration automatically generates two intensional operators. For each dimension d of `dimtype` *id*, the operators `index.d` and `org.d` are defined. In an expression `org.d` will give the value of *exp*, and `index.d` will give the value of dimension d in the current context.

Note that the concrete syntax of the language is more flexible. Operators can be defined to be infix, and the argument to a function need not be a single identifier. However, for the abstract syntax, this is sufficient.

All of the standard Lucid intensional operators can be defined using this syntax. The initial environment is presented in §3.

2.2 Semantics

Possible worlds Kripke [6] saw a possible world as the world that we live in, in which a certain set of parameters had been consciously changed. In computer science terms, this means that one can effectively get from one possible world to another.

For a given problem, the universe of possible worlds can be arbitrarily complex. The universe for ML–Lucid is the arbitrary cross-product of an arbitrary number of dimensions. Each dimension can be defined by using an ML datatype, even an abstract data type; each dimension is assumed to have a special element called an origin.

A possible world is an arbitrary cross-product of pairs $\rho = \prod_i D_i : d_i$ where the D_i are dimensions and the d_i are values associated with each dimension. Should a d_i be the origin of its dimension, written `org.`D_i, then it need not be written. In this way, infinite-dimensional universes may be manipulated, assuming that only a finite number of dimensions have been touched.

The universe U is the sum of all possible worlds.

Multi-dimensional objects A multi-dimensional object ϕ of type τ is a function from a universe U to τ. The order defined on objects is the natural extension on the order defined on objects of domain $\tau_\perp$. Write $\mathbf{SS}_\tau$ for the domain of multi-dimensional objects.

A *tuple* is a tuple of multi-dimensional objects: if $\phi_1, \phi_2, \ldots, \phi_n$ are multi-dimensional objects, then $\phi_1 \bullet \phi_2 \bullet \cdots \bullet \phi_n$ is a tuple object, using the induced order. The domain $\mathbf{S}$ of all objects is therefore:

$$\mathbf{S} = \sum_\tau \mathbf{SS}_\tau + \prod_{n \in \mathbf{N}} \mathbf{S}$$

The operator π_i is used to access the i^{th} element in the tuple ϕ.

Semantic functions The semantics of an expression E is $[\![E]\!]\zeta_0$, where ζ_0 is the initial environment containing any ML functions defined by the user, along with the initial intensional operators.

$$\begin{aligned}
[\![k]\!]\zeta &= \lambda\rho \cdot k \\
[\![id]\!]\zeta &= \zeta(id) \\
[\![id\,.\,d]\!]\zeta &= \zeta(id)d \\
[\![(E_1, \ldots, E_n)]\!]\zeta &= [\![E_1]\!]\zeta \bullet \ldots \bullet [\![E_n]\!]\zeta \\
[\![\texttt{\#i } E]\!]\zeta &= \pi_i([\![E]\!]\zeta) \\
[\![E_0\ E_1]\!]\zeta &= ([\![E_0]\!]\zeta)([\![E_1]\!]\zeta) \\
[\![\texttt{if } E_0 \texttt{ then } E_1 \texttt{ else } E_2]\!]\zeta &= \text{if } [\![E_0]\!]\zeta \text{ then } [\![E_1]\!]\zeta \text{ else } [\![E_2]\!]\zeta \\
[\![\texttt{<<}\ E_0, d_1\ \texttt{<-}\ E_1, \ldots, , d_n\ \texttt{<-}\ E_n\ \texttt{>>}]\!]\zeta &= \lambda\rho \cdot [\![E_0]\!](\rho[[\![E_1]\!]\ /\ d_1] \ldots [[\![E_n]\!]\ /\ d_n]) \\
[\![\texttt{let } decs \texttt{ in } E \texttt{ end}]\!]\zeta &= [\![E]\!]\big(\zeta[[\![decs]\!]\zeta']\big)
\end{aligned}$$

$$
\begin{aligned}
[\![\texttt{val } id = E]\!]\zeta &= \zeta[[\![E]\!]\zeta \ / \ id] \\
[\![\texttt{fun } id_1\ id_2 = E]\!]\zeta &= \zeta[\lambda id_2 \cdot [\![E]\!]\zeta \ / \ id_1] \\
[\![\texttt{fun } id_1.d\ id_2 = E]\!]\zeta &= \zeta[\lambda d \cdot \lambda id_2 \cdot [\![E]\!]\zeta \ / \ id_1] \\
[\![dec_1 \ \ldots \ dec_n]\!]\zeta &= [\![dec_n]\!](\ldots([\![dec_1]\!]\zeta)\ldots)
\end{aligned}
$$

`dimtype` was explained above. `dimvar` and `dimension` are esentially for type-checking, and their semantics is not shown.

3 The initial environment

The initial environment is required to run Lucid. Both mLucid and Indexical Lucid allow arbitrary numbers of integer dimensions, and supply a number of predefined operators, for each of these dimensions.

Lucid functions are compiled by replacing function calls with place-codes which denote positions in calling trees (see §5). The place-codes are recorded using integer lists, and the functions `call` and `actuals` are used to manipulate place-codes.

The initial environment therefore allows integer dimensions and integer list dimensions. There is one pre-defined integer dimension, `t`.

```
    function intToInt : int->int = external intToInt
    function intlistToInt : int list -> int = external intlistToInt

    %%

(* int is the type, org.d = 0, and intToInt is used for hashing *)
(* index.d is also defined *)
    dimtype   pwint     = (int, 0, intToInt)
    dimension t:pwint

    dimvar    d:pwint
    and       d1:pwint
    and       d2:pwint

    infix 4 wvr.d upon.d asa.d
    infix 3 fby.d bef.d

    fun first.d e       = <<e, d <- org.d>>
    fun next.d e        = <<e, d <- index.d+1>>
    fun prev.d e        = <<e, d <- index.d-1>>
    fun e0 fby.d e1     = if index.d<=org.d
                          then e0
                          else <<e1, d <- index.d-1>>
```

```
fun e0 bef.d e1      = if index.d<org.d
                       then <<e0, d <- index.d+1>>
                       else e1
fun e0 at.d e1       = <<e0, d <- e1>>
fun lparent.d e      = <<e, d <- index.d*2>>
fun rparent.d e      = <<e, d <- index.d*2+1>>
fun e0 wvr.d e1      = if first.d e1
                       then e0 fby.d (next.d e0) wvr.d (next.d e1)
                       else (next.d e0) wvr.d (next.d e1)
fun e0 upon.d e1     = e0 fby.d
                       if first.d e1
                       then (next.d e0) upon.d (next.d e1)
                       else e0 upon.d (next.d e1)
fun e0 asa.d e1      = first.d(e0 wvr.d e1)
fun to.d1.d2 e       = e at.d2 index.d1

(* int list is the type, org.c = [] and intlistToInt is used for hashing *)
(* index.c is also defined *)
    dimtype   pwintlist = (int list, [], intlistToInt)
    dimvar    c:pwintlist
    fun call.c(i,e)     = <<e, c <- i::index.c>>
    and actuals.c e     = <<nth(e, hd index.c), c <- tl index.c>>
    end
```

4 Examples

The following example computes the sum of the first eight even integers. `data` is the stream of even integers, in the `h` dimension. `binarySum`, at each time, adds pairs. The `lchild` and `rchild` operators can be viewed as accessing nodes in a full binary tree.

```
let dimension h:pwint
val data = 2*index.h
val binarySum = data fby.t (lparent.h binarySum + rparent.h binarySum)
in binarySum at.t 3 end
```

The result is, of course, 56.

The next example is an adaptation of a tournament computation approach to multiply two matrices, outlined in Faustini and Jagannathan's GLU paper [5]

```
let
   dimvar    a,b,c:pwint
   dimension i,j,k:pwint
   fun lll.a.b.c e = <<e, a<-index.a*2, b<-index.b*2, c<-index.c*2>>
```

```
    fun nnn.a.b.c e = <<e, a<-index.a+1, b<-index.b+1, c<-index.c+1>>
    fun nn.a.b e    = <<e, a<-index.a+1, b<-index.b+1>>

    val size = MINIMUM_ORDER fby.t size+size
    val p =
      let val conquer = combine(add(p,        next.k p),
                                add(next.j p, nn.j.k p),
                                add(next.i p, nn.i.k p),
                                add(nn.i.j p, nnn.i.j.k p))
      in
      mult(A at.j index.k, B at.i index.k) fby.t lll.i.j.k conquer
      end
in (display p) asa.t (size >= TARGET_ORDER)
end
```

With intensional variables, one can create new operators, such as `nn.a.b`, equivalent to `next.a next.b`, or `lll.a.b.c`, equivalent to `lparent.a lparent.b lparent.c`.

5 Implementing Higher Order Lucid

In his description of Higher Order Lucid, Wadge [9] gives the following second-order program:

```
let fun ffac(H,N) = if N<1 then 1 else H(N) * ffac(H,N-1)
    fun sq(P) = P*P
    fun cb(Q) = Q*Q*Q
in ffac(sq,3) + ffac(cb,4)
end
```

The second order parameter `H` can be compiled out, by introducing the predefined intensional operators `call` and `actuals`:

```
let dimension s:pwintlist
    fun ffac N = if N<1 then 1 else (H N) * call.s(0,ffac (N-1))
    fun H Z = actuals.s[sq Z, cb Z, H Z]
    fun sq P = P*P
    fun cb Q = Q*Q*Q
in call.s(1,ffac 3) + call.s(2,ffac 4)
end
```

The i^{th} call to `ffac` is replaced by `call.s(i,ffac ...)`. The arguments are placed in the `actuals.s` list. The first order parameters can also be compiled out, by introducing a second place-code:

```
let dimension f:pwintlist
    and       s:pwintlist
  val ffac = if N<1 then 1 else call.f(0,H) * call.s(0,call.f(0,ffac))
  val N = actuals.f[3,4,N-1]
  val H = actuals.s[call.f(0,sq), call.f(0,cb), call.f(1,H)]
  val Z = actuals.f[N,Z]
  val sq = P*P
  val P = actuals.f[Z]
  val cb = Q*Q*Q
  val Q = actuals.f[Z]
in call.s(1,call.f(1,ffac)) + call.s(2,call.f(2,ffac))
end
```

This procedure can be used to replace all function calls by calls to intensional operators `call.c` and `actuals.c`.

6 Implementing ML–Lucid

As for all variants of the Lucid family, ML–Lucid uses a warehouse to store its values. Conceptually, the warehouse stores triples of (context,identifier,value). In practice, what is done is to hash the context, finding it in a hash-table. For each context, there is then a symbol-table containing (identifier,value) pairs.

7 Conclusions

ML–Lucid is a language which allows one to combine all of the advantages of programming in a functional language with abstract data types, as well as those of programming in a dataflow language. Multi-dimensional dataflow problems can be solved for any number of dimensions, and for dimensions of any type.

ML–Lucid also allows the definition of new intensional operators, not just for a particular dimension, but for any number of dimensions, and not necessarily of the same type. This can be done through the `dimvar` and `fun` declarations.

ML–Lucid is also sufficiently powerful for the implementation of Higher Order Lucid, just by using the predefined `call` and `actuals` intensional operators.

Acknowledgments I would like to thank Chimeless Baltcha, Monica Lacayo, Charlotte Nankam Foka and John Ng for their help in building the first ML–Lucid interpreter.

References

[1] Gerard Allwein and J. Michael Dunn. Kripke models for linear logic, 1992. Available by ftp. Contact gtall@ogre.cica.indiana.edu.

[2] Ed Ashcroft. Lucid is second-order: Explaining lucid and intensionality to functional programmers. In *4th International Symposium on Lucid and Intensional Programming*, pages 1–8. SRI International, Menlo Park, Californaia, U.S.A., 1991.

[3] Weichang Du. *The Indexical Parallel Programming Paradigm.* PhD thesis, University of Victoria, B.C., Canada, 1991.

[4] A. A. Faustini and R. Jagannathan. Indexical Lucid. In *4th International Symposium on Lucid and Intensional Programming*, pages 19–34. SRI International, Menlo Park, Californaia, U.S.A., 1991.

[5] A. A. Faustini and R. Jagannathan. Tournament computations in GLU. In *4th International Symposium on Lucid and Intensional Programming*, pages 98–109. SRI International, Menlo Park, Californaia, U.S.A., 1991.

[6] Saul A. Kripke. *Naming and Necessity.* Harvard, 1972,1980.

[7] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

[8] R. D. Tennent. *Semantics of Programming Languages.* Prentice-Hall, 1991.

[9] Bill Wadge. Higher order Lucid. In *4th International Symposium on Lucid and Intensional Programming*, pages 62–69. SRI International, Menlo Park, Californaia, U.S.A., 1991.