Sick Lingo Ultra Reference Manual (SLURM)

Sophia Danielle Kolak - sdk2147
Jay Karp - jlk2225
Benjamin Flin - brf2117

February 2021

Contents

1 Introduction

The Lingo programming language is a functional programming language with rank-n polymorphism and linear types. Our language is heavily inspired by Haskell's implementation linear-core[1], however, we extend some aspects of Haskell's and Ocaml's syntax with a few features such as a specialized syntax for linear arrows. We choose OCaml as inspiration for our syntax design because it is compact and easy to understand. Unlike Haskell's linear core, however, we wanted the design of our language to highlight linearity as primary feature.

2 Lexical Conventions

There are six different kinds of tokens: identifiers, keywords, constants, binary operators, arrows and other special tokens. These categories of tokens are not disjoint. Similar to C, blanks, tabs newlines and comments are ignored except for their use in separating tokens where at least one is required.

2.1 Comments

The characters `(*` introduce a comment. The characters `*)` terminate the comment. Lingo does not allow single-line comments or nested multi-line comments

```
(*
    This is a comment in Lingo, how cool.
*)
```

2.2 Identifiers

Lowercase identifiers must begin with a lowercase ASCII character. The rest of the identifier string can be any combination of letters (uppercase or lower-case) and digits. Identifiers can also end with any number of single quotes `'`. Uppercase identifiers follow the same rules as lowercase identifiers except that they must start with an uppercase ASCII character.

```
let foo' : Int = 10;
(* foo' is a lowercase identifier *)
(* Int is an uppercase identifier *)
```

2.3 Keywords

The following keywords are reserved, and cannot be used as identifiers.

`if, then, else, let, in, data, case, of, where, Unr, One`

[1] https://arxiv.org/pdf/1710.09756.pdf

2.4 Integer Constants

Integer constants consist of a sequence of characters from 0 through 9. Note that integers are sized to 64 bits, so the value of any integer literal x is $x \mod 2^{64}$. The matching regex is `['0'-'9']+`

```
let integer : Int = 42;
```

2.5 Character Constants

Character constants are a single upper or lowercase character surrounded by single quote. The matching regex is `'[a-z, A-Z']'`

```
let a : Char = 'a';
```

2.6 Boolean Constants

Boolean constants are characters with either a value of `true` or `false`.

```
let coolbool : Bool = true;
```

2.7 Special Identifiers

2.8 Unit

If the characters `()` appear in sequence then they form a token called unit. Otherwise, they form two separate tokens left parentheses `(` and right parentheses `)`.

```
let unit : () = (); (* unit *)
let foo : Int = (3 + 4); (* parentheses (not unit) *)
```

2.9 Curly Braces and Vertical Bars

Curly braces { } and the vertical bar | also form tokens.

```
let foo  : Int = id {Int} 1;
let foo' : Int = id Int |Unr| 1;
```

2.10 Forall and ForallM

The tokens `@` and `#` are called forall and forallm respectively form two tokens. Note that each of these tokens can and often appear next to an identifier without whitespace.

```
let id {a} |p| x : @a #p (a -p> a) = x;
```

2.11 Operators

Lingo supports the following tokens as Operators.

```
!= Not Equal
<= Less Than or Equal to
<  Less Than
>= Greater Than or Equal to
>  Greater Than
== Equal
|| Or
&& And
*  Multiplication (a.k.a. Star)
/  Division (a.k.a. Slash)
+  Addition
-  Minus or Negate (a.k.a. Dash)
!  Not
```

The greater than token >, the dash token -, the slash token / and the star token * are all tokens which are also used in syntax outside the context of binary and unary operators, as will be discussed later in this paper.

```
let x : Int = 10 * 10 / 10 + 10 - 10
```

We will discuss each of these operators behaviors and precedence later in the paper.

2.12 Separating and other tokens

The backslash token \, the wildcard token _ the colon : and the semicolon ; are the remaining tokens not present elsewhere in this document.

```
data PartyAnimal {a} |p| where
    Cow     : a -p> PartyAnimal;
    Giraffe : a -p> PartyAnimal;
    Hyenas  : a ->  PartyAnimal;
    Donkeys : a ->  PartyAnimal; (* both : and ; tokens *)
let foo : Int = case (Just 4) of
    Just a -> a + 1;
    _ -> 0; (* wildcard *)
;
```

3 Types

Types are written after certain expressions and top-level statements, and are usually separated from the expression/top-level statement using a colon.

```
let foo : Int = 0;
```

Since Lingo has no type inference, types are needed in most expressions.

There are three primitive types in Lingo: Int, Char, Bool. There is an infix type operator `a -> b` which represents functions from types a to b. This binary operator on types is right associative.

```
let f x : Int -> Int = x;
```

Lingo does support polymorphic types. This allows functions to be variable in the types that they accept. In order to deal with polymorphism, Lingo builds these polymorphic functions by type at compile time. To define a polymorphic function you use { as well as the }. This can either take in multiple type arguments each individually surrounded by these braces (`{a} {b} {c}`), or all of the types can be placed in the same set of braces `{a b c}`. In the functions type declaration, you must also declare types using the forall quantifier (`@`).

```
let foo {a b} f x : @a @b (a -> b) -> a -> b =  f x;
let _ = foo {Int Char} toString 10;
let _ = foo {Char Int} toInt '1';
```

Lingo supports rank-N polymorphism allows for support of any number of quantifiers which can appear before any argument in a lambda. For example:

```
data Tuple {a b} where
    Tuple : a -> b -> Tuple;
let tup {b c} x y : @b @c (@a a -> a) -> Tuple b c
    = Tuple (id {b} x) (id {c} y);
```

4 Multiplicities

Multiplicities describe how an argument is evaluated inside an expression. A multiplicity is either `One` or `Unr`. If a multiplicity of a variable is `One`, then the argument is evaluated in the expression exactly once. If it is `Unr` than can be evaluated any number of times, including zero.

The following expression is well-typed:

```
let x : One Int = 10 in x + 1
```

Arrows can be annotated with a multiplicity. The arrow `(->)` by default has multiplicity `Unr` and `(-*)` has multiplicity `One`.

Multiplicities can also be polymorphic, i.e. there are multiplicity variables. Introduction of multiplicity variables is done through multiplicity lambdas, and multiplicity quantifiers appear in types prepended with a # token. In arrows, the multiplicity variables appear infix.

For example:

```
let id {a} |p| x : @a #p (a -p> a) = x;
```

Multiplicities can also be multiplied. For concrete multiplicities `One` and `Unr`, `One * One` is equivalent to One and all other cases are `Unr`. Multiplication of multiplicities is associative and commutative by construction.

For example:

```
let compose {a b c} |p q| f g x :
    @a @b @c #p #q
    (b -q> c) ->
    (a -p> b) ->
    a -p*q> c = f (g x);
```

The above example expresses the constraint that `x` must be used `p*q` times where `q` is the multiplicity of f in its argument and `p` is the multiplicity of p in its argument.

Below is a more comprehensive grammar for type, multiplicites and arrows:

```
<type> :=  @ <lowercase-id> <type>
        |  # <lowercase-id> <type>
        |  <type> <arrow> <type>
        |  <type> { <type+> }
        |  <type> | <type+> |
<arrow> :=  ->
        |   -*
        |   - <mult> >
<mult> := <lowercase-id>
        | Unr
        | One
        | <mult> * <mult>
```

Note that `<type+>` denotes one or more types in sequence.

5 Expressions

We use the following syntax for specifying grammars. A terminal/token is either written by the characters from which it is constituted or by using a self-descriptive name like `integer-literal`. Non-terminals are given with angle brackets, such as `<expr>`. A production rule is given using `<non-terminal> := token <other-non-terminal>` `...` Anything contained in square brackets `[]` is considered optional.

5.1 Lambda expressions

There are three types of lambda expressions, one for values, types and multiplicities. We use `\` to denote the start of a lowercase lambda abstraction, `/` to denote an uppercase (type lambda) and `|` to denote multiplicity abstraction.

Lambda expression:

```
<expr> := \( lowercase-id : <type> ) <arrow> <expr>
```

Note that the non-terminals `<type>` and `<arrow>` are defined in sections 3 and **??**.

```
\( x : Int ) -* x + 1
```

Type lambda expression:

```
<expr> := /lowercase-id -> <expr>
```

e.g.

```
/a -> f {a} 10
```

Multiplicity lambda expression:

```
<expr> := |lowercase-id -> <expr>
```

e.g.

```
|x -> f |x| 10
```

Type and multiplicity abstraction allow for polymorphism in types and multiplicities respectively which in turn can be passed around and applied to other type and multiplicity abstractions. For more information on types and multiplicities see sections 3 and 4.

5.2 Let-in expressions

A let-in expression allows for binding a new variable equal to an expression inside another expression.

```
let x : Int = 5 in x + 3
```

The above let-in expression is equivalent in terms of operational semantics to the following expression:

```
(\(x : Int) -> x + 3) 5
```

A let-in expression can optionally take in a list of parameters, which in themselves are lists of lowercase ids surrounded by vertical bars, curly braces or just separated by whitespace. (Note that a param-list is any number of params separated by whitespace, and in general, a x-list is any number of xs surrounded by whitespace).

```
<expr> := let <lowercase-id> <param-list> : [<mult>] <type>
    = <expr> in <expr>
<param> := | <lowercase-ids> |
         | { <lowercase-ids> }
         |   <lowercase-ids>
```

The type parameter of the let-in expression is the type of the bound name given as the lowercase identifier after the `let`.

Each parameter enclosed inside vertical bars, curly braces, or no enclosure represent multiplicity abstraction, type abstraction and value abstraction respectively. To illustrate, the following represents pairs of let-in expressions which are equivalent:

```
let id x : Int -> Int = x in id 0
let id : Int -> Int = \(x: Int) -> x in id 0

let id {a} x : @a a -> a = x in id 0
let id : @a a -> a = /a -> \(x: a) -> x in id {Int} 0

let id {a} |p| x : @a #p a -> a = x in id 0
let id : @a #p a -> a = /a -> |p -> \(x: a) -p> x in id {Int} 0

let compose {a b c} |p q| f g x :
    @a @b @c #p #q (b -q> c) -> (a -p> b) -> a -p*q> c
    = f (g x) in (id (succ 0))
```

5.3 If expressions

An if expression takes the form

```
if <expr> then <expr> else <expr>
```

Note that the else is mandatory. The first expression after the if is expected to have a type `Bool`.

5.4 Case expressions

Case expressions are syntactically similar to those in Haskell, however, in Lingo each case expression is concluded with a `;`, they take the following form:

```
<expr> := case <expr> of <casealts>
```

Casealts is a representation of all of the different case alternatives that are involved in matching. They take the form:

```
uppercase-id <lowercase-ids> -> <expr>;
```

Each Casealt statement is made up of an uppercase identifier which allows for argument deconstruction, a list of lowercase identifiers in sequence to pattern match on, as well as the resulting expression to be evaluated when matching to that case. We do not support deep pattern matching for individual cases. e.g.

```
let x : Maybe {Maybe {Int}} |One| = case (Just 4) of
    Just a -> Just a;
    Nothing -> Just 0;
;
```

5.5 Binary expressions

Binary expressions take the form

```
<expr> <binop> <expr>
```

Where the binary operators are:

```
||, &&, ==, !=, <, >, <=, >=, +, -, /, *, !
```

The operators are presented in order of least to most precedence, where the following groups of operators each have equal precedence and are left-to right associative.

```
==, !=
<, >, <=, >=
+, -
*, /
```

In this context, * means multiply, / means divide, and - means subtract. Note that we label these tokens as Star, Slash, and Dash in section 2.11 because each of these tokens are used in other parts of Lingo's syntax.

5.6 Application expressions

Lingo supports application of multiple different kinds. This includes type application, multiplicity application and function application. The following two expression are logically equivalent a display how application works in Lingo. Application is also left associative by default.

```
let foo {a} {b} {c} |p| |q| f g x :
    @a @b @c #p #q (a -p> b) -> ((a -p> b) ->
        (b -q> c)) -> a -p*q> c
    = f g x in
foo {Int} {Char} {Int} |Unr| |Unr|

let foo' {a b c} |p q| f g x :
    @a @b #p #q (a -p> b) -> ((a -p> b) ->
        (a -q> b))-> a -p*q> b
    = (f g) x in
foo' {Int Char Int} |Unr Unr|
```

In Lingo these expressions would both need to be used in a top level declaration, however, this is omitted here for clarity.

5.7 Literal expressions

Expressions can consist of a single lowercase id which can refer to a variable. Expressions can also consist of a integer literal token as well as a character token described in sections 2.4 and 2.5 respectively.

5.8 Precedence of Expressions

Expressions described in this section are generally listed in order of lowest to highest precedence, where the first four (sections 5.1-5.4) are of equal precedence. Application has a higher precedence than any binary operator, for example. The following expression

```
f a + g a
```

is equivalent to

```
(f a) + (g a)
```

Any expression surrounded with parentheses has highest precedence.

6 Algebraic Data Types

Algebraic Data Types allow Lingo to support sum types as well as product types. These data types can take in both type and multiplicity parameters.

```
data Tuple {a b} |p| where
    Tuple : a -p> b -p> Tuple;

data Maybe {a} |p|  where
    Just    : a -p> Maybe;
    Nothing : Maybe;

let x : Maybe {Int} |Unr| = Nothing;
let y : Maybe {Int} |Unr| = Just 10;
```

Here each case, which are separated by semicolons, represents a sum type. In this example, a Tuple represents a product type. `a` and `b` represent type variables and `p` represents a multiplicity variable. We also do not allow quantifiers in each case definition. We can instantiate the type and multiplicity variables in Algebraic Data Types using the above syntax.

7 Types Revisited and Kinds

Kinds describe the 'type' of types. A kind can be either `Mult` or `Type` or an `(->)` from any two kinds. Concrete types such as `Bool` or `Maybe {Int}` have a kind `Type`. All arrows `(->)` are infix operators with kind

```
Type -> Type -> Type
```

which means `(->)` can produce a type if given two other types. The following definition

```
data Maybe {a} |p|  where
    Just    : a -p> Maybe;
    Nothing : Maybe;
```

has kind

```
Type -> Mult -> Type
```

We can instantiate any kind of the form

```
Type -> k
```

where `k` is any kind by application with curly braces, and we can instantiate a kind of the form

```
Mult -> k
```

with vertical bars. Using the definition for Maybe given above, we can build the following:

```
let x : Maybe {Int} |Unr| = Just 1;
```

In Lingo, all kinds are inferred, so there is no syntax for specifying the kind of a type. However, kinds are still useful concept in reasoning about types.

8 Top-level Statements

A top-level statement can consist of a let statement. The following are all valid let statements.

```
let id {a} |p| x : @a a -p> a = x;
let foo : Maybe {Int} |Unr| = Just 1;
```

In general, a let statement follows the same syntax as a let-in expression, with the `in` and the second <expr> omitted. A top-level statement can also be a datatype declaration, defined in section 6.

9 Programs

A program consists a sequence of top-level statements. In a program, all top-level statements are automatically mutually recursive, so the order in which they are defined is irrelevant. The entry point for a program is a let statement with the identifier `main` of type `()`. For example,

```
let main : () = print 0;
```

10 Example Program

Below is an example of a valid program:

```
data List {a} where
    Cons : a -> List {a} -> List;
    Nil : List;

let map {a b} l : @a @b (a -> b) -> List {a} -> List {b}
```

```
    = case l of
        Cons x xs -> Cons (f x) (map {a b} xs);
        _ -> Nil;
;

let fin n : Int -> List {Int}
    = if n == 0 then Nil else Cons n (fin (n - 1));

let ignore x : @a a -> () = unit;

let main : () = ignore (map print (fin 10));
```