Lingo Programming Language Proposal

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

January 2021

1 Overview of Lingo

The Lingo programming language is a functional programming language with rank-n polymorphism, typeclasses and linear types. Our language is heavily inspired by Haskell's implementation linear-core[1], however, we extend Haskell's syntax with a few features such as a specialized syntax for linear do notation, and a way of writing linear arrows. We choose Haskell 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.

The Goals for this project are:

- Haskell-style syntax with common features such as do-notation, list comprehension, algebraic data types with polymorphism in multiplicity and type. In order to implement do-notation we will have limited type inference to infer the type of linear and non-linear binds.
- Type safety throughout the language to ensure safe, reliable and understandable code.
- Linearity in order to provide safe manual memory management and safe usage protocols of external resources such as file operations and networking.

2 Language Details

2.1 Linearity

We extend the normal function arrow $\rightarrow$ with a linear one $\multimap$. The linear arrow $f : A \multimap B$ functions the same as a normal function arrow, with one additional guarantee: if the output $(f\ u) : B$ is consumed once, then the input $u : A$ is consumed once. This guarantee is enough to ensure that the programmer follows a resource protocol.

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

The following examples show how implementing linearity can be useful for writing safe code. In the illtyped function, because malloc returns of Ptr of usage 1, the output value can only be used once. When you try to apply this value to the someOp' function, the compiler realizes that p can only be used once, and therefore does not compile. In the welltyped function, malloc returns a pointer of usage 1, and then passes it to a function which only consumes this pointer once.

```
malloc  : Int -> IO 1 Ptr
someOp  : Ptr ⊸ IO 1 Ptr
someOp' : Ptr -> IO 1 Ptr
free    : Ptr ⊸ IO ω ()

illtyped : IO ()
illtyped = do
    p  ⟜ malloc 10
    p' <- someOp' p
    free p'


welltyped : IO ()
welltyped = do
    p  ⟜ malloc 10
    p' ⟜ someOp p
    free p'
```

We also refer to these usages as Multiplicities, where someOp has a multiplicity of 1, and someOp' has a multiplicity of ω, or unrestricted. Note that IO is parameterized by it's multiplicity 1 or ω, which indicates how the resultant value must be used. This is true of any Monad in our language. From that fact, we can derive a class definition for Monad in lingo consisting of the following:

```
class Applicative m => Monad m : Mult -> Type -> Type where
    (>>=) : ∀ a b. ∀ₘ p q. m p a -> (a -p> m q b) -> m q b
```

In the monad definition, the p in

`(a -p> m q b)`

exactly encodes the constraint that the a in

`m p a`

must be used with multiplicity p. Note that multiplicities are not of kind ‘Type’ but have their own kind.

2.2 Data Types and Operations

Primitive data types,

Data Type:	Description:	Typeclasses:
Int{8, 16,32 64}	An integral type of size {1,2,4,8} bytes	Eq, Ord, Semigroup, Monoid, Semiring, Ring, etc.
Char	A character type of size 1 byte, the same as Int8	Eq, Ord, Semigroup, Monoid, Semiring, Ring, etc.
Float	32-bit floating point number	Eq, Ord, Semigroup, Monoid, Semiring, Ring, Field etc.
Double	64-bit floating point number	Eq, Ord, Semigroup, Monoid, Semiring, Ring, Field etc.
Bool	Boolean values, either True or False	Eq, Ord
Ptr	Can only be obtained linearly and allows tracking and keeps track of pointer size	Eq, Ord

Data types can be declared as follows:

$$
\begin{array}{l}
\texttt{data D}\ p_1\ \ p_2\ \ \ldots\ \ p_n\ \ \texttt{:}\ k_1 \to k_2 \to \ldots \to k_n\ \texttt{where} \\
\qquad (c_k\ \ \texttt{:}\ \ A_1 \to_{\pi_1} \ldots \to A_{n_k} \to_{\pi_{n_k}} D)_{k=1}^{m}
\end{array}
$$

This reads as a data constructor D has kind $k_1 \to k_2 \to \ldots \to k_n$ with constructors c_k of type $A_1 \to_{\pi_1} \ldots \to A_{n_k} \to_{\pi_{n_k}} D$ with k ranging from 0 to m.

One simple usage could be:

```
data Maybe p a : Mult -> Type -> Type where
    Just    : a →p Maybe
    Nothing : Maybe
```

Note that this definition of Maybe allows us to define the Monad instance given the monad definition in 2.1

2.3 Keywords

The following keywords are reserved in Lingo. `if, then, else, let, in, data, case, of, where, module`

2.4 Comments

Our commenting system is the same as Haskell's in that we use double dash – for single line comments and {- -} for multi-line comments. We also support nested comments.

```
-- x = 42, this is a single line comment
main : IO ()
main = do
    putStrLn "Stephen Edwards 42"

{-
This is a multi-line comment

why isnt this working!?
main : Int -> Int
main = main + 1
-}
```

3 Design Decisions

Lingo will have limited type inference. This means that most if not all expressions have to be annotated. Type inference in System-F polymorphism is not trivial on its own to implement since it is undecidable in general, let alone the extension to type-polymorphism to multiplicity-polymorphism. Although recent work has solved[2] inference for rank-1 multiplicity polymorphism, we believe that inferring multiplicites in a rank-n setting with top-level annotations and typeclasses remains open.

Because we have little type inference, we sometimes have to provide type parameters to type or multiplicity abstractions. For example:

```
id : ∀ a. a -> a
id x = x

y : Int
y = id {Int} 0

-- Types can be inferred in simple cases
z : Bool
```

[2] https://arxiv.org/pdf/1911.00268.pdf

```
z = id False
-- equivalent to
-- z = id {Bool} False
```

The identity function is parameterize over the type. In most cases, the type is easy to infer from the parameter passed, however, in more complex cases the implicit type parameter will have to explicitly provided.

Although Lingo provides low level interfaces for memory manipulation, algebraic data types are garbage collected. This simplifies many of the types and makes programming as simple as it would be in Haskell.

4 Standard Library

We plan to include a small standard library to ship with Lingo. This will be similar to Haskell's prelude, including folding, mapping etc.

5 Mutliplicity Checker

Below is our rudimentary multiplicity checker, which validates that multiplicities are used in the correct way in the correct context. It is only a proof of concept at this point.

```haskell
import Control.Monad.Except
import Control.Monad.State
import Data.List ((\\))
import Data.Map (Map)
import qualified Data.Map as M
import Data.Maybe (fromJust)

data Base
  = Int' Int
  | Bool' Bool
  deriving (Show)

data Binop = Add | Subtract | Multiply
  deriving (Show, Eq)

data Expr
  = BaseExpr Base
  | Var String
  | Lam String Mult Type Expr
  | App Expr Expr
  | MLam String Expr
  | MApp Expr Mult
  | Op Binop Expr Expr
```

```haskell
  | If Expr Expr Expr
  deriving (Show)

data Mult
  = One
  | Unr
  | MVar String
  | Plus Mult Mult
  | Times Mult Mult
  deriving (Show, Eq)

data BaseT = TInt | TBool
  deriving (Show, Eq)

data Type
  = TBase BaseT
  | TLam Mult Type Type
  | Forall String Type
  deriving (Show, Eq)

data Err
  = NotInScope String
  | Mismatch Type Type
  | NonLinear String
  | NotAFunction Type
  | NotAMLam Type
  | Unsatisfiable Constraint
  deriving (Show)

-- p <= q
data Constraint = Constraint Mult Mult
  deriving (Show, Eq)

type Env = Map String Type

type Usage = Map String Mult

type Check = ExceptT Err (State [Constraint])

mult :: Mult -> Usage -> Usage
mult m = fmap (simp . Times m)

times :: Usage -> Usage -> Usage
times a b = simp <$> M.unionWith Times a b

plus :: Usage -> Usage -> Usage
plus a b = simp <$> M.unionWith Plus a b

```

```haskell
simp :: Mult -> Mult
simp m = case m of
  (Plus _ _) -> Unr
  (Times a One) -> a
  (Times One a) -> a
  (Times Unr _) -> Unr
  (Times _ Unr) -> Unr
  a -> a

type Subst = (Mult, Mult)

class Substitutable a where
  subst :: a -> Subst -> a

instance Substitutable Mult where
  subst (MVar x) (MVar y, z) = if x == y then z else MVar x
  subst (Times a b) x = Times (subst a x) (subst b x)
  subst (Plus a b) x = Plus (subst a x) (subst b x)
  subst a _ = a

instance Substitutable Type where
  subst (TLam m t t') x = TLam (m `subst` x) (subst t x) (subst t' x)
  subst (Forall p t) x = Forall p (subst t x)
  subst t _ = t

instance Substitutable Constraint where
  subst (Constraint x y) z = Constraint (subst x z) (subst y z)

instance Substitutable a => Substitutable [a] where
  subst l a = flip subst a <$> l

reduce :: [Constraint] -> Check [Constraint]
reduce [] = return []
reduce (c : cs) = do
  cs' <- reduce1 c cs
  cs'' <- reduce cs
  return (cs' ++ cs'')
  where
    reduce1 :: Constraint -> [Constraint] -> Check [Constraint]
    reduce1 c@(Constraint One _) cs = return []
    reduce1 c@(Constraint _ Unr) cs = return []
    reduce1 c@(Constraint Unr One) cs = throwError (Unsatisfiable c)
    reduce1 c cs = return [c]

addConstraint :: Constraint -> Check ()
addConstraint c = do
  modify (c :)
  reduceConstraints
```

```haskell

reduceConstraints :: Check ()
reduceConstraints = get >>= reduce . s >>= put
  where
    s = fmap (\(Constraint x y) -> Constraint (simp x) (simp y))

check :: Expr -> Env -> Check (Type, Usage)
check (BaseExpr (Int' _)) env = return (TBase TInt, M.empty)
check (BaseExpr (Bool' _)) env = return (TBase TBool, M.empty)
check (Var x) env =
  case M.lookup x env of
    Just t -> return (t, M.singleton x One)
    _ -> throwError (NotInScope x)
check (Lam x p t e) env = do
  (t', u) <- check e $ env `M.union` M.singleton x t
  let m = fromJust $ M.lookup x u
  addConstraint $ Constraint m p
  return (TLam p t t', u M.\\ M.singleton x Unr)
check (App e1 e2) env = do
  (t1, u1) <- check e1 env
  (t2, u2) <- check e2 env
  case t1 of
    (TLam q a b)
      | t2 == b -> return (b, u1 `plus` (q `mult` u2))
      | otherwise -> throwError (Mismatch t2 b)
    _ -> throwError (NotAFunction t1)
check (MLam p e) env = do
  (t, u) <- check e env
  return (Forall p t, u) -- TODO: Subtract any existing p from env
check (MApp e1 q) env = do
  (t1, u) <- check e1 env
  case t1 of
    (Forall p t) -> do
      modify (`subst` (MVar p, q))
      reduceConstraints
      return (t `subst` (MVar p, q), u)
    _ -> throwError (NotAMLam t1)
check (Op binop e1 e2) env = do
  (t1, u1) <- check e1 env
  (t2, u2) <- check e2 env
  if t1 /= TBase TInt
    then throwError (Mismatch t1 (TBase TInt))
    else
      if t2 /= TBase TInt
        then throwError (Mismatch t2 (TBase TInt))
        else return (TBase TInt, u1 `plus` u2)
check (If b e1 e2) env = do
  (t, u) <- check b env
```

```haskell
  if t /= TBase TBool
    then throwError (Mismatch t $ TBase TBool)
    else do
      (t1, u1) <- check e1 env
      (t2, u2) <- check e2 env
      if t1 /= t2
        then throwError (Mismatch t1 t2)
        else return (t1, u `plus` (u1 `times` u2))

runCheck :: Expr -> Either Err (Type, Usage, [Constraint])
runCheck e = do
  let (a, s) = runState (runExceptT $ check e M.empty) []
  (t, u) <- a
  return (t, u, s)

-- Ex. runCheck $ App (MApp (MLam "p" (Lam "x" (MVar "p") (TBase TInt) (Var "x")))
--    One) (BaseExpr (Int' 𝟘))
-- ((Λp.(λx :p int. x)) ω) 𝟘
-- Ex. runCheck $ (MLam "p" (Lam "x" (MVar "p") (TBase TInt)
--    (If (BaseExpr $ Bool' True) (Var "x") (Op Add (Var "x") (Var "x") ))))
-- Λp.(λx :p int. if True then x else (x + x)) : (∀p. int →p int, ω ≤ p)
-- Thus, applying the above to a multiplicity is only well-typed if and only if p = ω
```