Lingo:
A prototypical functional language for linearity polymorphism

Benjamin Flin, Jay Karp, Sophia Kolak
Agenda

1. Lingo Overview
2. Linear Typing Motivation
3. Compiler Architecture
4. Testing
5. Demo
6. Future Work
Lingo

- Linear Typing
- Functional/Strong type
- Algebraic Data Types
- Pattern matching
- Rank-n polymorphism
- First class functions / lambda calculus
- C interoperability
Lingo in one Slide
Lingo in One Slide

- External Function Declaration
- Abstract Data Types
- Cases + Pattern Matching
- Type and Multiplicity Polymorphism
- Function Application
- Multiplicities and Types

```
print_int : Int -> Int;

data Stephen a #p where
    A : a -> Stephen a #p;
    Edwards : Maybe a #p;

foo : Int -> Stephen Int #One
    = \x. A @Int #One x;

foo' m
    : Stephen Int #One -> Int
    = case m of
        A i -> i;
        _ -> 0;

main : Int = print_int (foo' (A @Int #One 1));
```
Motivation for Linear Typing

- Want a type system which eliminates classes of bugs violating resource protocols, i.e. double free, closing file pointers, use after free, etc.

- In order to do this, we need to characterize whether/how a function will use or evaluate its argument.
Linear arrows

- Extend the normal function arrow \( \rightarrow \) with a linear one \(-o\).
- Linear arrow from \( a \rightarrow o b \) functions same as a normal function arrow, with one additional guarantee: if the output of an application, is evaluated once, then the input will be evaluated exactly once.
- Guarantee is enough to ensure that the programmer follows a resource protocols.
  - \( f : a \rightarrow o b \)
  - \( u : a \)
  - \( f u : b \) (* \( u \) is used linearly *)

\[
(a \rightarrow b) \\
(a \rightarrow o b)
\]
Linearity Example

\[
\text{malloc} : \text{Int} \rightarrow (\text{Mem} \rightarrow () \rightarrow ()
\]

\[
\text{free} : (\text{Mem} \rightarrow ()
\]

\[
\text{set} : \text{Int} \rightarrow \text{Byte} \rightarrow \text{Mem} \rightarrow (\text{Mem} \rightarrow () \rightarrow ()
\]

\[
\text{malloc} 5 (\lambda m. \text{set} 0 0 (\lambda m. \text{free} m))
\]
A problem with compose

How should the following be typed?

\[
\text{compose} : (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c = \lambda f.\lambda g.\lambda x.f \ (g \ x)
\]

Need 4 different types for the same function body. We answer with linearity polymorphism.
System F

- Extends simply-typed lambda calculus with a new type of abstraction for types denoted by uppercase lambda.
- Types can be applied and they are substituted on the type-level.
- **Idea:** Use this type system to characterize polymorphism in linearity

\[
\begin{align*}
id & = \Lambda \alpha. \lambda(x : \alpha).x : \forall \alpha. \alpha \to \alpha \\
\text{foo} & = \text{id} \ \text{Int} \ 0 : \text{Int}
\end{align*}
\]
Compose again

\[ \forall m.p. \forall m.q. \forall a. \forall b. \forall c. (b \rightarrow p \ c) \rightarrow (a \rightarrow q \ b) \rightarrow a \rightarrow_{pq} c = \Lambda m.p. \Lambda m.q. \Lambda a. \Lambda b. \Lambda c. \Lambda f. \Lambda g. \Lambda x.f \ (g \ x) \]

```haskell
print_int : Int -> Int;
succ x : Int -> Int = x + 1;

compose p q a b c f g x : #p #q @a @b @c (b -p> c) -> (a -q> b) -> a -p*q> c = f \ (g \ x);
main : Int = (compose #Unr #Unr @Int @Int @Int print_int succ) 100;
```
Demo compose
Typing rules

\[
\frac{\text{x} \in \Gamma}{\Gamma \vdash \text{x} : A \leadsto \{\text{x} \mapsto 1\}} \quad \text{var}
\]

\[
\frac{\Gamma, \text{x} : A \vdash t : B \leadsto \{\text{x} \mapsto \mu, U\} \quad \mu \leq \pi}{\Gamma \vdash \lambda(\text{x}:_{A})t : A \to_{\pi} B \leadsto \{U\}} \quad \text{abs}
\]

\[
\frac{\Gamma \vdash t : A \to_{\pi} B \leadsto \{U\} \quad \Gamma \vdash u : A \leadsto \{V\}}{\Gamma \vdash t \, u : B \leadsto \{U + \pi V\}} \quad \text{app}
\]

\[
\frac{\Gamma \vdash t : A \leadsto \{U\} \quad \text{p fresh for } \Gamma}{\Gamma \vdash \lambda p. t : \forall p. A \leadsto \{U\}} \quad \text{m.abs}
\]

\[
\frac{\Gamma \vdash t : A \leadsto \{U\}}{\Gamma \vdash t \, \pi : A[\pi / p] \leadsto \{U\}} \quad \text{m.app}
\]
Compiler Architecture

[Diagram showing the compilation process from Source Code to Executable]

- Source Code
  - Scanning/Parsing
  - Source AST
    - Elaboration
    - Core AST
      - Typechecking
      - Semantically Checked AST
        - Monomorphization
        - Monomorphized AST
          - Closure Conversion
          - Closed AST
            - Code generation
              - LLVM
                - llc
                - ASM + Library
                  - gcc
                  - Executable
Elaboration

- Source Ast -> Core Ast
- During elaboration, all syntax is expanded into a form which the typechecker can understand.
- This includes
  - Converting all variable names to Debruijn indices
  - Each lambda into their corresponding abstractions (value, type, or multiplicity)
  - Variable lists into nested lambdas.
Typechecking

- Core ast -> semantically checked $\text{Sast}$ with annotated types.
- linearity checking and type checking happens.
- Multiplicities are stripped out
- If there is a type or multiplicity mismatch of any sort, or any other type related error, the typechecker will throw an error.
Monomorphization

- polymorphic / rank-n code -> non-polymorphic / rank-1 code.
- LLVM doesn't have polymorphism!
- Our strategy: BoxT, a single type which represents all polymorphic variables.
- Convert to and from BoxT through the use of new expressions Box and Unbox.
- Box: turns any value into a BoxT
- Unbox: turns any Box value back into a given type.
- Application to a lambda of type BoxT -> BoxT boxes the argument and unboxes the result.
- Later down the pipeline during code gen, this will result in a simple cast in LLVM.
Closure Conversion

- **Mast** -> performs lambda lifting until everything is a top level declaration
- Uniquely named lambdas
- Takes every free variable and adds it to a closure environment
Code Generation

- Conversion of Closure Converted Cast -> llvm IR
- Abstract data types become tagged unions, i.e. \{ i64, i8* \}. The first parameter, the tag, tells us what constructor is pointed to as the second parameter.
- Everything becomes function application and building closures.
- External C Calls

```c
struct List_Int {
    int tag;
    union {
        struct Nil {} *nil;
        struct Cons {
            int x;
            struct List_Int *xs;
        } cons;
    } value;
};
```
Testing

- Testing run inside docker container before every git commit
- Passing and Failing Tests
- Syntax
- Memory Allocation
- Function Composition
- Polymorphism
- Basic Operations
- Abstract Data Types
Demo Safe File (Time Permitting)
Future Work

- Better monomorphization? (may be intractable)
- Qualified types and interfaces
- Replace current type checker with quantitative type theory (dependent types + linear types)
  - This actually may have been easier to implement in the long-run as types and terms exist on the same level. Typechecking will become a little more costly, however.
- Garbage Collection / better memory management
- Fix bugs (hopefully there are no big ones :)
  - “A segfault a day keeps the sanity away” - Ben Flin
Thank You

Questions?