1 Typed Lambda Calculus

We propose to extend the language `p0` with function declarations by λ-expressions:

1.1 Description

Introducing λ-expressions into `p0` will imply that we have first-class functions. A Lambda expression can be assigned to any variable, parameter or returned in a method body. All lambda expressions only take one parameter as its input and lambda expressions may return other lambda expressions as well.

1.2 Examples

```
class Functions {
  var add : (Int -> Int -> Int) = λx : Int. λy : Int. x + y;
  var factorial : (Int -> Int)
    = λn.
        if (n == 0)
            1
        else
            n * (factorial $ n - 1);

  def add() : (Int -> Int -> Int) = {add}
  def fact() : (Int -> Int) = {fact}

  def method(Boolean b) : (Int -> Int -> Int) = {
    var sub : (Int -> Int -> Int) = null;
    sub = λx : Int. λy : Int. x - y;
    if (b)
      add
    else
      sub
  }
}
object Main extends App {
  var fns : Functions = new Functions();
  println(fns.method(true) $ 5 $ 2); // 7
  println(fns.method(false) $ 5 $ 2); // 3
  println(fns.add() $ 3 $ 2); // 5
  println(fns.fact() $ 6) // 720
}
```

1.3 Extensions

The lexer would then be extended with three new tokens `λ` and `\` representing a lambda, the arrow `->` as well as the keyword `lambda`.

The parser would be extended with:

```
Type :: =
    ...
    | Type -> Type
    | (Type)
Expression ::=
   ...
   | λ Identifier : Type. Expression
   | Expression $ Expression
```

Where the `$` application operator has the lowest precedence in `p0`.

Before static analysis we propose to add a new phase that performs lambda-lifting. The lambda lifting phase will convert all lambda expressions into their closures and create a class for each closure where all free variables in the lambda expression will be applied through the constructor of the class. The lambda-lifting will however be performed after assigning the symbols to each identifier present in the AST.

In the name analysis we lax the restriction of re-assignment of method parameters inside the lambda expressions. The programmer should note that the values of all free variables inside a lambda expression will be copied during creation. As per usual objects will only have their references copied.

Lambda expressions are considered **"constant"** in variable declarations if the outermost lambda have no free variables.

In the type-checking phase the types of lambda expressions are type checked through their generated class declarations in the lambda-lifting phase. Here we add the type checking rules:

$$\lambda \ \frac{e : R}{\lambda x : T.e : (T \to R)}$$

$$\lambda_{<:} \ \frac{T :> T' \qquad R <: R'}{(T \to R) <: (T' \to R')}$$

$$\$ \ \frac{x : T' \qquad T :> T'}{(f : (T \to R) \ \$ \ x) : R}$$

1.4 Further Extensions

The future extensions that greatly increase the usefulness of the lambda expressions are:

- Tail Call elimination
- Parametric polymorphism
- Algebraic Data Types
- Lazy Evaluation
- Type Inference
- Differentiate Expressions and Statements
- Differentiate Mutable and Immutable Objects

1.4.1 Tail Call elimination

Lambda expressions lend themselves to many recursive or compositional algorithms with small functions and many calls. This means that much code that makes use of the lambda expressions is likely to have greater performance if there is less stack use and calls are faster. Some algorithms are unfeasible to implement recursively using a stack with limited depth on modern hardware.

1.4.2 Parametric polymorphism

Some higher-order functions such as `map` or `fold` only come to their full potential if possible to apply to generic types. This can be made possible by introducing type variables.

1.4.3 Algebraic Data Types

In order to increase pure-ness while still retaining usability of the language Algebraic Data Types together with pattern matching significantly allieviates some of the mental strain on the programmer when programming functionally.

1.4.4 Lazy Evaluation

Some problems such as self-referring data structures are not possible to implement with strict evaluation. If one were to remove the quite imperative assignment operator some algorithms are made impossible. Introducing lazy evaluation would then re-introduce some of the expressiveness lost by removing the imperative paradigm.

1.4.5 Type Inference

It is annoying and time consuming to write out all types and the inclusion of type inference in the compiler would lessen the burden of writing programs in `p0`.

1.4.6 Differentiate Expressions and Statements

The assignment operation is incongruent with the construct being called `Expressions` since the assignment operation never yields any value but is soley side-effectful. If one were to have such operations they would better fit inside a `Statement` construct and the imperative constructs while, sequential composition (;) and assignment should be confined to the `Statement` construct.

1.4.7 Differentiate Mutable and Immutable Objects

To reduce the amount of bugs due to mutable data there should be a way to classify some objects as immutable and this should probably be the default. ADTs should always be immutable and other objects (classes) should have some kind of transactional imperative construct.