CSC 410 Project Proposal

Elliott Lawrence

Overview

Purely functional programming languages have the nice property that every function is a function in the mathematical sense. A consequence of this is that functional programs can be easily represented as a graph, and the execution of such a program involves reducing the graph down to a single node, which is the program's output.

My proposal is to take advantage of this graphical representation and design a visual programming language which allows users to create programs with flow charts rather than code. The language will be modeled after Haskell, and will have many of the same basic features and properties. In particular, the language will be purely functional and will have lazy, non-strict semantics and strong, static typing.

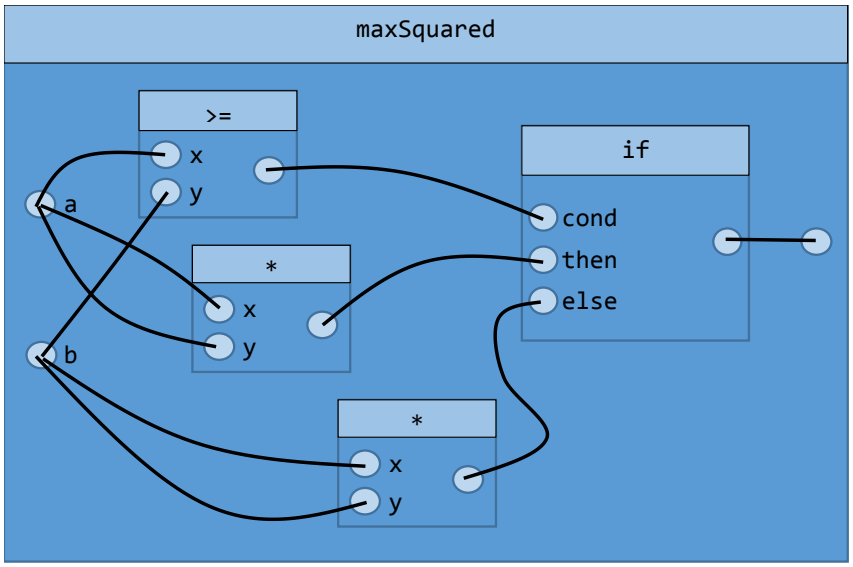
The goal of this language is not necessarily to improve programmer efficiency, but rather, to offer a different way of visualizing programs that emphasizes their functional nature as well as the flow of inputs and outputs.

Example

The user interface will be very similar to Quartz Composer, where functions are represented as boxes, and inputs and outputs are represented as lines connecting the boxes. For a taste of what this would look like, the following Haskell code:

```
maxSquared a b = if a >= b then a*a else b*b
```

would be represented visually as:



Project Steps

Due to the scope of this project, it may take two semesters to implement. Here is a rough sketch of the steps that need to be taken:

- Read the 2010 Haskell Language Report, figure out all the features Haskell has that I would like to implement, determine how they will be represented visually, and write a simple language specification to work off of.
- Create the UI, which will be a grid where users can drag and drop elements to create their programs. The UI will be able to convert these visual diagrams into a Haskell data type that represents the abstract syntax tree of a program, and vice versa.
- Write the type checker, which will perform Hindley-Milner type inference on the AST.
- Write a module that performs a series of transformations on the AST to convert it into a much simpler core language data type, as described in Simon Peyton Jones's book *Implementing Functional Languages*. The core language consists of function definitions, let expressions, case expressions, data constructors, and lambda abstractions.
- As described in Peyton Jones's book, a common way of implementing graph reduction that also allows for a number of optimizations is through a low-level architecture called the G-machine. The core language will need to be compiled into a data type that represents a sequence of instructions to be executed by this G-machine.
- Write the compiler that will translate G-machine instructions into LLVM code. This will allow programs to be executed on a number of different architectures without having to write a separate compiler for each one.