Flo: A visual, purely functional programming language

Elliott Lawrence

In CSC 410 I designed and started working on flo, a visual, purely functional programming language. The syntax and semantics are heavily inspired by Haskell, whereas the visual editor is modeled after Apple's Quartz Composer. Instead of writing out lines of code, programmers create flow charts that model the execution of their program. Flo features strong static typing, non-strict semantics, parametric polymorphism, algebraic data types, and other features commonly shared by purely functional languages, presented in a visual manner that is intuitive and extremely simple to learn.

Last semester I mainly focused on creating the visual editor. I made a simple compiler that converts flo programs into Haskell code and runs the Haskell compiler to generate an executable. I still need to add support for algebraic data types to this implementation, and once that is done, I'll start working on my main goal for the semester, which is to make a proper compiler based on the one used by Haskell. This implementation is called the Spineless Tagless G-machine (STG), and it is described in a paper by Simon Peyton Jones[1].

The compilation process consists of the following steps:

- First, a .flo file (really just a JSON file) is parsed and converted into a data type called FloProgram, which represents the abstract syntax tree (AST).
- Next, type checking is performed. Flo uses strong static typing, but does not require explicit type annotations, so I'll be using Hindley-Milner type inference to infer and check types. The specific algorithm I'll be using is called Algorithm W[2], but it will need to be extended to handle user defined data types.
- The AST is then converted into a small functional language called STG. The only expressions allowed in STG are function applications, constructors, let expressions, and case expressions.
- STG is then compiled into C code. This is the most complicated part of the process, and the paper by Peyton Jones goes into great detail about how this translation occurs. The advantage of generating C is that it is low level and portable, and obviously much easier than generating assembly.
- The C code can then be compiled into an executable.

Another thing I need to do is implement a method for performing input and output. There are a couple ways to do this in a purely functional language, but the two I am considering are

---

[1] http://research.microsoft.com/apps/pubs/default.aspx?id=67083

[2] http://catamorph.de/documents/AlgorithmW.pdf presents one version of the algorithm.

monads, which is what Haskell uses, and dialogues, which represent a program as a function taking an infinite list of requests and returning an infinite list of responses.

Since this project could continue indefinitely, I intend to do three credits worth of work on it. Here is an outline of the deadlines I'm hoping to meet:

- Add support for algebraic data types to existing simple compiler (February 3rd)
- Implement the type checker (February 10th)
- Translation from FloProgram to STG (February 17th)
- Translation from STG to C (March 16th)
- Add support for updating thunks (March 30th)
- Implement IO system (April 13th)
- Implement garbage collection (April 20th)
- Improve editor (as time permits)
- Improve error messages (as time permits)