Pony Design Document

Andrew K. Hirsch

October 3, 2012

1 Introduction

Imagine you are a soldier in, say, the American Revolution. You have just been granted a rifle, and you want to customize it to fit your particular style. You have two options: you can add a longer barrel, or you can add a bayonet. The longer barrel allows you to take more accurate shots, and the bayonet allows you to fight at a short distance.

This sort of customization of tools is common in many fields. Programmers even have it to some degree; many heavily customize the environment that they code in. But what every programmer deals with on a daily basis is the language that they code in, which is, in many cases, completely non-customizable. Pony wants to give this sort of power to a programmer in their language, so that programmers can use the tools that they are most comfortable with.

However, if you add a bayonet to your gun with a longer barrel, then the longer barrel will keep the bayonet from doing what you want. This is an important point with customization: two customizations may interfere with each other. In this case, it is impossible to use both of them at the same time. To do so would be logically impossible. This does not change when we are talking about programming languages rather than weapons.

Pony attempts to detect when a programmer is attempting to use such customizations. This is actually impossible in the general case; however, we attempt to do so for as many cases as possible. This is a significant theoretical contribution, and is not at all trivial.

Pony is written in Haskell, which allows us to prove properties. Haskell forces us to write in a purely functional style. In order to allow for these customizations, we must write a program that can take a customized language as input. However, nobody has written such a program in a purely functional style before, at least not that is published in the literature. Writing such a program is also a technical challenge and contribution of Pony.

2 Core Requirements

- Allow users to use programming language constructs in C.
  - Allow users to use Objects, including class-based and prototype-based.
  - Allow users to use Lists, such as one might find in LISP or Haskell.
- Allow users to use these constructs in embedded systems.
  - By compiling to ANSI C, this comes naturally.
- Allow users to develop new programming languages, and compile them to C.
  - By allowing users arbitrary control of syntax and semantic transformation, it is possible to use Pony to write a compiler for i.e. Javascript.

3 List of Functions

Because Pony is in its second development phase, there are three pieces of functionality to focus on:

- Collision detection for semantic transformations
  - When a transformation's action may work on the action of another transformation, the user should be alerted to this possibility.

  - When a transformation's action will work on the action of another transformation on a particular piece of code, Pony should recognize this as an error.

- Languages for writing extensions and semantic transformations
  - Pony will have to parse these languages, so that it can feed them to interpreters or translators.
  - Pony will have to interpret these languages, using them in its processes, or translate them into a language it can use.

- A parsing function (Written using Parsec)
  - We need to be able to parse arbitrary formal languages, so that Pony can work with extended languages that do not follow C syntax rules.
  - We need to write a function for extending a pre-written parser, so that transformations can be written without specifying the entire language.

4 Use Cases

- A user wants to write C code with a new construct.
  - If the construct has already been written as a transformation, then the user can simply use that transformation.
  - Otherwise, the user must define a transformation before writing with it.

- A user wants to write a new language and Compile to C
  - The parser should no longer accept C code at this point.
  - The user must specify the syntax of the language as well as how it is transformed into C code.

- A user wants to write a new extension and publish it for others.

- Some extensions (i.e. Objects) may be standard.
- This means that transformations must be generalized and publishable!

5 Data Structures

Since Pony is written in Haskell, we eschew C-style data structures for the most part, preferring type-theoretic data structures such as sum types and records. However, we will also be making extensive use of linked lists, a FP standard, and trees, a compiler standard. Pony will also use monads and other category-theoretic data structures. Most of these are baked in either to Haskell or the pre-existing Pony code.

6 Key Algorithms

The main algorithm that is going to be introduced will be unification, which will be used for collision detection. Also important will be monad composition and transformation, combinator libraries, and Data Types a la Carte-style extensibility. The GHC API or its derivatives might come into play, but will hopefully not.

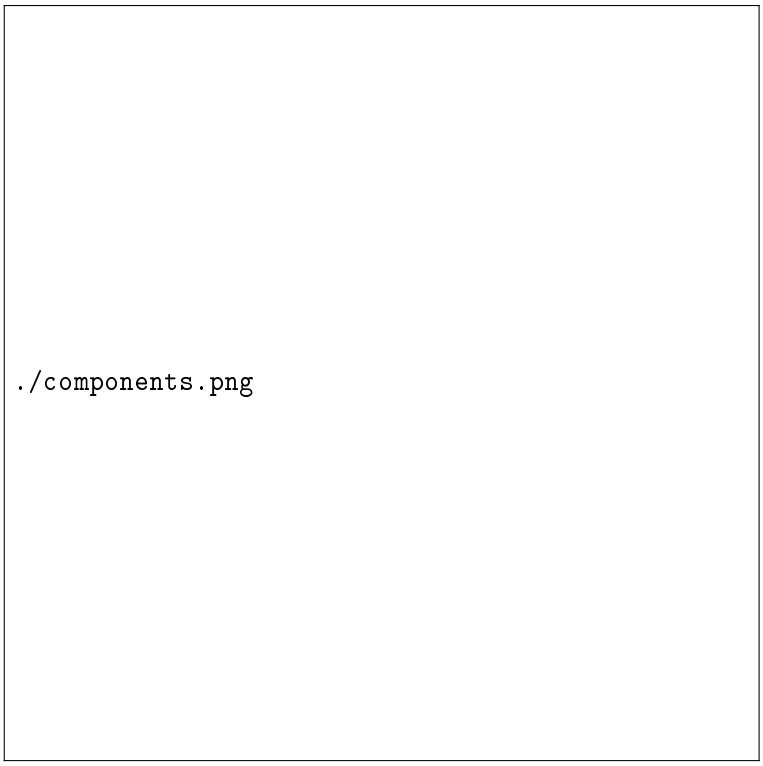

Figure 1: Pony Modules