Pony Design Document

Andrew K. Hirsch

September 11, 2012

1 Introduction

Pony is a tool designed to make life easier for developers.

The philosophy of Pony is simple: use the tools that make your life easier. Rather than forcing developers into using a specific line of tools, Pony allows developers the flexibility to define new tools, and use those new tools when developing. This means that developers can be more comfortable, more productive, more accurate, and have more fun when doing their work.

1.1 What Pony Does

Pony is a compiler; it is actually that simple. However, Pony allows for a user to define a *transformation* of the base language. In other words, Pony allows its users to *extend* the compiler to compile a broader class of languages. Pony then translates source code written in such an *extended language* back into the original language – ANSI C99.

This point may be confusing, because most compilers translate into a binary format, or some assembly language that is in one-to-one correspondence with a binary format, or near enough. However, by compiling Pony to C, we allow Pony to be used on any range of architectures, not just a specific one. In particular, by chaining Pony with another compiler, such as the GNU C Compiler (GCC), a developer can write code in an extended language and have it run on x86, MIPS, ARM, and other popular architectures.

Importantly, Pony also insists on correctness. Pony will detect transformations that will not work together, and when transformations may not work together on a certain piece of code. We call these problems “collisions”, and verified collision detections is an important part 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 <u>will</u> 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. Also used will be 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.

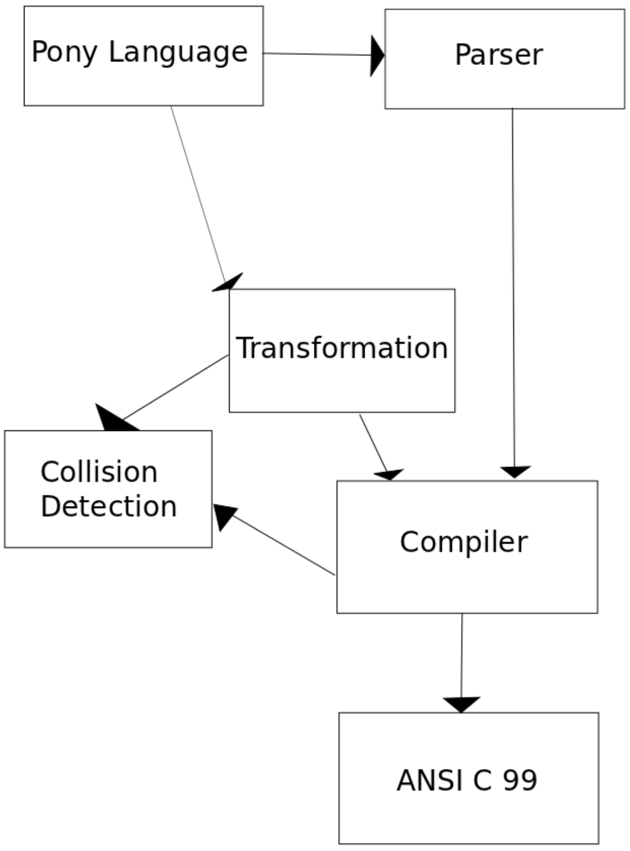

Figure 1: Pony Modules