Pony Working Document

Andrew Hirsch

September 22, 2012

1 Introduction

In Pony, we allow users to define extensions to a base language as transformations of that language. However, a user may be using more than one transformation at any particular time. If the user is attempting to apply transformations that conflict with each other, data may be overwritten by transformations in such a way that the transformations are no longer semantics-preserving; indeed, we may no longer create valid C99 code.

Most compilers will tell us when we are doing something wrong. Either they will give an error if given code that they cannot process, or they will give a warning if they recognize a pattern that is commonly a bug. We would like for Pony to detect such collisions for us, and give us a similar error.

To achieve this, we exploit the use of Data.Generics ("Scrap your boilerplate"-style coding) and the structure of programming languages as semantic trees.

2 Strong Collision

In strong collision, we detect possible collisions between two transformations. We do this by looking at the code of the transformations; it should only run on certain types of nodes. The central idea is that we create a type theory the abstract syntax tree of the (possibly extended) language the transformation runs on.

The existence of BNF suggests that this is possible: if we have a BNF formula as follows:

```
<expression> ::= <expression> + <term>
              | <expression> - <term>
              | <term>
```

Then there must be a datatype in the tree correspondingly:

```
data expression = Plus expression term
                | Minus expression term
                | Standalone term
```

It should be possible (through dynamic programming) to create a listing of all node types that can be reached from an `expression` node. Any transformation that works on an `expression` node also affects all nodes reachable from the `expression` node. Thus, we can have a type theory of nodes with a subtyping relation.

What allows for tighter expressiveness for strong collision is the idea of guards. The idea here is that if you have the following transformation:

```
trans :: Expr -> Expr
trans Plus e1 e2 = Minus e1 e2
trans Minus e1 e2 = Plus e1 e2
trans e@(Standalone _) = id e
```

We then *refine* the type of trans to

```
trans :: Expr -> Expr | forall e. e = (Plus e1 e2) \/ (Minus e1 e2)
```

For more complex transformations, this is the same as type inference for dependent types. A cursory search of the literature tells us that this may not be possible in the general case. (See: Dependent Types: Easy as PIE and Type Inference and Reconstruction for First Order Dependent Types.)

It appears from the above example that we don't have much added from guards. However, consider the following example:

```
data NumType = NumFloat Float
             | NumInt Int
             | NumLong Long

doIntLong :: NumType -> NumType | forall (NumInt n). n > 0
doIntLong f@(NumFloat _) = id f
doIntLong i@(NumInt n)
    | n > 0 = NumLong n
    | otherwise = id i
doIntLong l@(NumLong _) = id l

doIntFloat :: NumType -> NumType | forall (NumInt n). n <= 0
doIntFloat f@(NumFloat _) = id f
doIntFloat i@(NumInt n)
    | n <= 0 = NumFloat (fromInteger f)
    | otherwise = id i
doIntFloat l@(NumLong _) = id l
```

As you can see, these transformations (`doIntLong` and `doIntFloat`) will never collide. However, naively looking at the types they work on, even branching into the type to see what constructors it uses, would be too weak to decide this. But we see that is impossible to ever unify the refined types of `doIntFloat` and `doIntLong`. Thus, our system, with guards, can tell that these transformations will compose.

However, even if this is not decidable in the general case, it may be possible to find correct types, but not types that are as "tight" as possible, and thus we would be too conservative in our warnings. However, this would be acceptable in most cases.

3 Weak Collision

In weak collision, we detect only possible collisions between two transformations *on a particular piece of code.* This requires a bit more work, but allows for pony transformations to be used in more general cases.

To do weak collision, we use the Curry-Howard correspondence to get a logical version of the transformations' Haskell code. We can then unify these over the database of tree nodes, using standard techniques of logic programming.

This requires a library for getting logical forms of Haskell code. Currently, I am only aware of one library, and it only finds the propositional forms of types. Finding such a library, or a work-around, is paramount.

4 Better Weak Collision?

Because our version of weak collision only works on the node level, it is possible that two transformations might be rejected because they work on the same node, but do not actually overwrite each other. However, detecting this kind of "better weak collision" is difficult, and is probably left for another day.

5 Better Strong Collision?

The holy grail of this work would be to find an algorithm that would detect that two transformations are incompatible in **exactly** the cases where they may overwrite one another. However, this is an uncomputable problem (see: Rice's Theorem). It may very well be possible to create better versions of strong collision. However, this is difficult work, along the same lines of the work on halting (see: Microsoft Terminator).

6 Parsing

Parsing is a hard problem in Pony, harder than might be expected. Haskell is very commonly used for compilers and interpreters, so parsing can't be that hard of a problem, right?

This is correct in most senses. There are two very good common parsing tools for Haskell: Happy, a YACC-style parser, and Parsec, a parser combinator library using monads. However, Pony runs into something interesting: The Expression Problem. This states that in functional languages, it's hard to come up with new cases to a datatype, while in (Object-Oriented) imperative programming languages, it is difficult to add new functions to a datatype. (Interestingly, C seems to fall into the functional camp here: I don't know that that means anything, but it's interesting.)

Since Pony requires parsers that can be arbitrarily extended, we have to solve the expression problem, and this is *not* easy. We plan to use a technique from the functional pearl Data Types a la Carte, which allows for subtyping in Haskell. (Basically, it lifts Either to kind $* \to *$, with automatic injections and searches, but this is not important.) We will combine this with the Parsec library.

Unfortunately, Parsec is currently in its third iteration (Parsec 3), and has a very complicated structure. However, by moving to the first iteration of the Parsec library (Parsec 1), we may be able to take advantage of the simpler structure of the monad. (We do not currently use anything that is more powerful than the Parsec 1 monad, as far as I'm aware.)