Deep Strictness: Milestone 1

Kenny, Hengchu

November 3, 2017

At the first phase of the project, we have investigated the objects in our abstract domain for characterizing strictness of a subset of GHC Core.

In particular, we have restricted our scope to functions that take a single argument, and produces a single argument. However, we place no constraint on the types of the input and output. They can be terms of arbitrary user-defined algebraic data types as input.

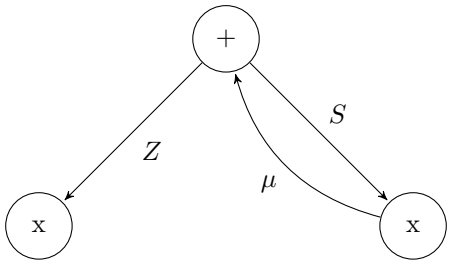
An example of such a function is `half`, which divides a natural number by 2.

```
-- |The data type of natural numbers
data Nat = Z     -- ^ Zero
         | S Nat -- ^ Successor of a natural number

half :: Nat -> Nat
half n =
  case! n of
    Z    -> Z
    S n' -> case! n' of
              Z     -> Z
              S n'' -> S (half n'')
```

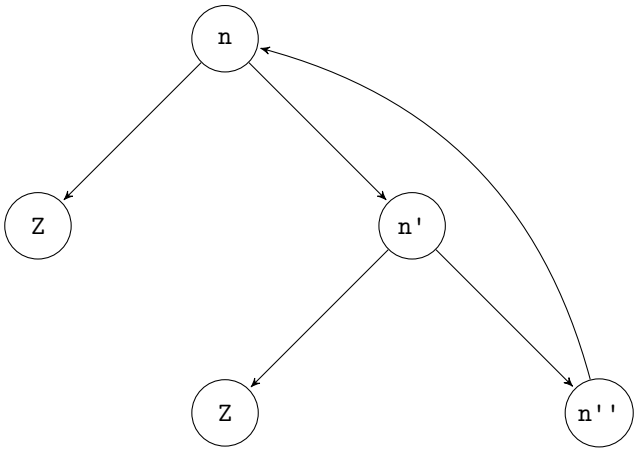
The `half` function pattern matches on the input number, and returns 0 if the input is either 0 or 1, and if the input is 2 successor constructors applied to some other natural number, then it recursively invokes itself on that natural number, and finally applies a constructor to the result of the recursive computation. We will use `half` as the running example in the following discussion.

Our first observation is that we can use a graph to represent the structure of an algebraic data type, with the internal nodes correspond to choices of constructors and projection of arguments applied to constructors. Concretely, the `Nat` type is represented by this graph:



The `+` node represents the choice between constructor `Z` and constructor `S` in a natural number, where the μ edge represents the recursive structure of `Nat` — the field of the successor constructor is also a natural number.

Having the type of `Nat` represented as a graph is a first step towards exploring representations of strictness in this format. We also need a control flow graph of the function under the analysis. In this case, the half function can be represented by the following CFG:



The CFG can be interpreted as the following: the function `half` branches on the variable n, and returns if n is `Z`, or it branches on the predecessor n' of n, and terminates if n' is `Z`, otherwise it recurses on the predecessor n'' of n'.

Now, we can connect the nodes on the type `Nat` and the nodes on the CFG of `half` to approximate the strictness behavior.

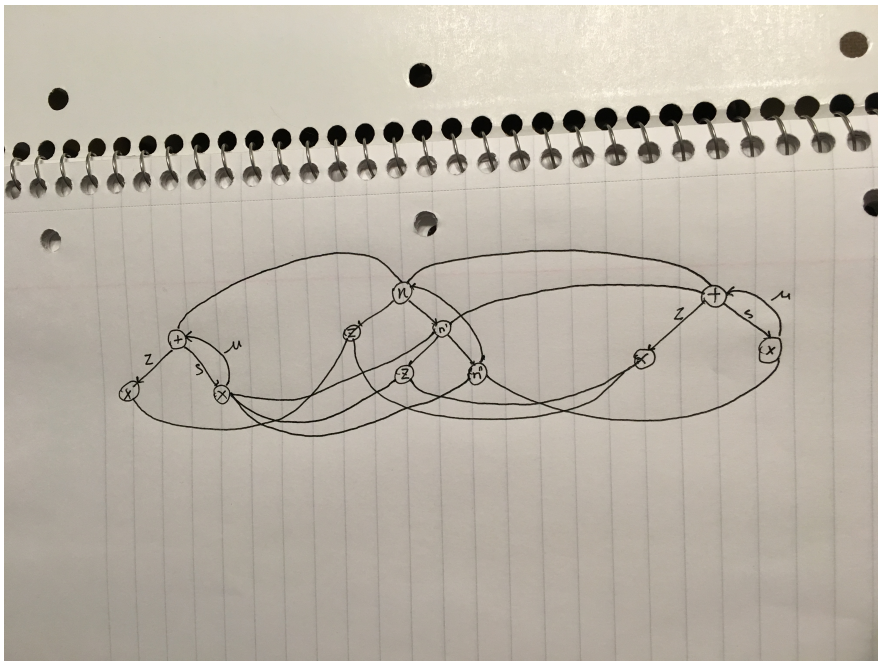


Figure 1: Strictness graph of `half`

Here, the graph on the righthand side propagates demand of the output through the control flow graph, back to the input on the lefthand side. The way to read this graph is

1. `half` produces nothing at nodes n and n'. Hence, they're connected to the top node on the RHS, which has not produced any constructors yet.
2. `half` produces a `Z` constructor when n turns out to be zero. Hence, it is connected to the `x` node following the `Z` branch on the RHS.
3. `half` also produces `Z` when n' turns out to be zero. Hence the connection between the lower `Z` node and the left `x` node following the `Z` branch on the RHS.

4. `half` produces a `S` constructor when n' turns out to be the successor of some number n''. Hence, there is a connection between n'' and the `x` node following the `S` branch.

5. We also connect all the pattern matching points to where the sums occur on the LHS. Since we pattern match on n, which corresponds to the top level `+` node on the LHS, there is an edge between them. Similarly, there is an edge between `x` following the `S` branch to the node n'. Note that the same `x` is also connected to n'', here is where we lose some precision on exactly where n'' occurs in the structure of the input. However, the precision can be improved by unrolling the LHS's recursion a few times.

The graph of `half` captures a sound approximation of the function's strictness behavior. We conjecture that there is in fact a class of functions whose strictness behavior can be soundly approximated like this.

Also, conceptually, the graph above describes a Finite State Transducer, which is a variant of Non-deterministic Finite State Automata that has an input tape and an output tape, and the transition graph has the same restrictions as an NFA. In this case, the input tape holds the constructors that one demands of the output, and the output tape outputs the constructors one *may* have forced on the input. This seems to suggest some form of regularness of the functions that can be characterized by such graphs.

One example that cannot be represented by graphs like this is a degenerate `swap` function that infinitely changes the position of its two arguments.

```
badSwap :: (a, a) -> a
badSwap p =
  case! p of
    (x, y) -> let p' = (y, x) in
              badSwap p'
```

The graph is not capable of capturing the change of position of the first and second elements in the tuple in a sound way, and we're currently considering solutions to this problem. One could potentially unroll the recursion in `badSwap` until the order of arguments is restored, but such methods can quickly break down in situations where the number of unrolling required is not known statically. Another solution is to develop a method of distinguishing such "misbehaving" functions from the functions whose strictness behavior can be captured by a graph, and our analysis will simply refuse to give a graph for these functions.