Narrowing: The Scope of this Project

Steven Libby

December 3, 2014

This project implements a narrowing strategy, specifically the Needed Narrowing strategy [1] for Term Rewriting. This implementation is correct and relatively efficient. It can be used to form a simple functional logic DSL inside of Haskell. The scope of this project is limited to implementing this strategy. There are many features in PAKCS that are extensions to the basic idea of Narrowing, and they are not covered. As such, the resulting DSL is not convenient, but it is functional. Further development of this system into a real language, or integration with Funlog, are possible future projects, although integration with Funlog would require modification of the Funlog parser.

1 Term

Terms are similar to the terms that we covered in the first order logic example, but there are a few additions that I've had to make. The first is that I've made a distinction between a constructor and a function. This is because the theory of narrowing, or at least the part I'm familiar with, is built around constructor based systems. In this case, a term will be in a normal form when it has only constructors and variables, or there are no more rewrite rules to apply. The first case is actually a subset of the second case, but the first case gives us a clear stopping point when narrowing.

```
module Term where
import Subst

type Name = String
```

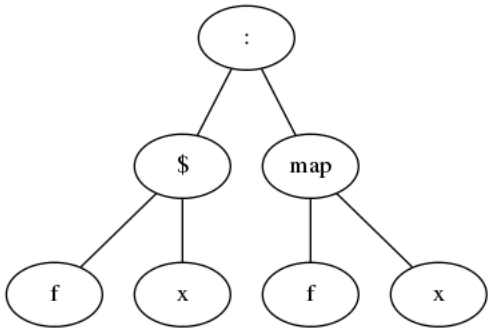
To get anywhere with narrowing we need to start with a term. A term t is inductively defined as either

- a constructor with 0 or more subterms
- a function with 0 or more subterms
- a variable

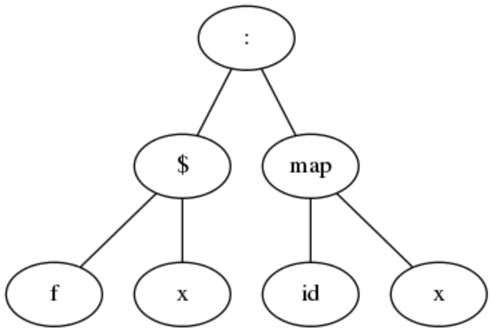
```
data Term v = Con Name [Term v]
            | Fun Name [Term v]
            | Var v deriving (Show, Eq, Ord)

type Pos = [Int]
```

There are a few important points about terms. Let t be a term. The arity of t is the number of direct subterms it has. In our case this is the length of the subterm list. A position p in t is a sequence of integers that determine a subterm of t. The subterm of t determined by p (written $t|_p$) is a path down the tree where we take the subtree at position p.

So if t is the tree of $fx : mapfxs$



and p is the path $[2, 1]$ Then $t|_p$ is f. Finally if t is a term, p a position and s a term then $t[s]_p$ is the result of replacing $t|_p$ with s. so $t[id]_p$ would result in.



A rewrite rule is a pair of terms where the left term is rooted with a function. We write $l \to r$ to refer to the rule. The idea is that if we have $l \to r$ and $t|p = l$ for some p then we could rewrite that to $t[r]_p$

The general problem of term rewriting is difficult. In fact in many systems equality of terms is undecidable. To get around this problem, we limit ourselves to left linear, non overlapping, constructor based systems. In practice this amounts to making sure that variables in the left hand side are distinct, and the left hand side of a rule can't unify with any subterm of the left hand side of another rule.

```
data Rule v    = Term v :=> Term v | E deriving (Show, Eq, Ord)
type Program = [Rule Name]
```

In order to create the structures needed to narrow a term we need three different equivalence relations. The first is the usual definition of equality given in Haskell. The second is equality up to variables. That means two terms are equal if renaming variables in the first term produces the second term. The third is equality of the function and constructor symbols. This is just used for grouping differnt rules together.

```
(~=) :: (Eq v) ⇒ Term v → Term v → Bool
(Con c ts) ~= (Con d us) = c ≡ d ∧ and (zipWith (~=) ts us)
(Fun f ts) ~= (Fun g us) = f ≡ g ∧ and (zipWith (~=) ts us)
(Var _) ~= (Var _) = True
_          ~= _          = False

(~~) :: (Eq v) ⇒ Term v → Term v → Bool
(Con c _) ~~ (Con d _) = c ≡ d
(Fun f _) ~~ (Fun g _) = f ≡ g
(Var v)    ~~ (Var w)    = v ≡ w
_             ~~ _             = False
```

The following function are all simple helper functions. Note that *arity* and ! > correspond to the notion of arity and position defined above.

```
isCon (Con _ _) = True
isCon _             = False

isFun (Fun _ _) = True
isFun _             = False

isVar (Var _) = True
isVar _          = False

x ! > [] = x
(Con _ cs) ! > (h : t) = (cs !! h) ! > t
(Fun _ fs) ! > (h : t) = (fs !! h) ! > t

left   (l :=> _) = l
right (_ :=> r) = r

arity                :: Term a → Int
arity (Var _)     = 0
arity (Fun _ ts) = length ts
arity (Con _ ts) = length ts
```

variables gets a list of variables occurring in the term. I know, shocking.

```
variables             :: Term v → [Term v]
variables (Var v)     = [Var v]
variables (Con c ts)  = [v | t ← ts, v ← variables t]
variables (Fun f ts)  = [v | t ← ts, v ← variables t]
```

Finally we come to variable substitution. We handle this using a monad, because the syntax is similar to how this is normally written. If t is a term and σ is a substitution then $t >>= (\sigma!) \equiv t\sigma$ is the application of σ to t. A common pattern is $(t! > p) >>= (\sigma!)$ and this is read as "take the subterm $t|_p$ and apply σ to it.

```
instance Monad Term where
  return v = Var v
  (Var v)     >>= s = s v
  (Fun f ts) >>= s = Fun f (map (>>=s) ts)
  (Con c ts) >>= s = Con c (map (>>=s) ts)
```

1.1 Substitution

The substitution is very similar to the one used in class for first order logic. The only difference is that I've changed the type from a function to a *Map*. This has two advantages. The first is that lookups in a *Map* are faster than a function, although given the inefficiencies in the *pdt* construction the value here is debatable. The second, and much more important, advantage is that a Map can be printed out. This was indispensable for debugging.

```
module Subst (Subst, emptySubst, (| =>), (|− >), (|/− >), (!), chain) where
import qualified Data.Map.Lazy as M

type Subst v m = M.Map v (m v)
```

We need a simple function for getting values out of our substitution. This is normally written as $\sigma(v)$, but we use the notation $\sigma!v$.

```
(!) :: (Ord v, Monad m) ⇒ Subst v m → v → m v
s ! v
  | M.member v s = sM. ! v
  | otherwise     = return v
```

The following functions are equivalent to the definitions in the first order logic example.

```
emptySubst :: Monad m ⇒ Subst v m
emptySubst = M.empty

(|− >)   :: (Ord v, Monad m) ⇒ v → m v → Subst v m
v |− > t = M.singleton v t

(| =>)      :: (Ord v, Monad m) ⇒ Subst v m → Subst v m → Subst v m
s1 | => s2 = M.union s1 s2

chain    :: (Ord v, Monad m) ⇒ [Subst v m] → Subst v m
chain ss = foldr (| =>) emptySubst ss

(|/− >)   :: (Ord v, Monad m) ⇒ v → Subst v m → Subst v m
v |/− > s = M.delete v s
```

2 unification

```
module Unify where
import Term
import Subst
```

This is nearly identical to the unification algorithm in the first order logic example. I actually did some of my own work I swear. Two terms t and t' unify if there exists a substitution σ such that $t\sigma = t'$.

```
occurs :: (Eq v)                ⇒ v → Term v → Maybe ()
occurs v t
   | any hasV (variables t) = Nothing
   | otherwise              = Just ()
     where hasV (Var u)     = v ≡ u

unify  :: (Show v, Ord v)                ⇒ Term v → Term v → Maybe (Subst v Term)
unify (Var v)     (Var u)      | u ≡ v = return emptySubst
unify (Var v)     y                    = occurs v y ≫ return (v |− > y)
unify x           (Var v)              = occurs v x ≫ return (v |− > x)
unify (Fun f ts) (Fun g ss)  | f ≡ g  = unifyLists ts ss
unify (Con c ts) (Con d ss) | c ≡ d  = unifyLists ts ss
unify x           y                    = Nothing

unifyLists :: (Show v, Ord v) ⇒ [Term v] → [Term v] → Maybe (Subst v Term)
unifyLists []        []       = Just emptySubst
unifyLists []        (x : xs) = Nothing
unifyLists (x : xs) [ ]        = Nothing
unifyLists (x : xs) (y : ys) = do s1 ← unify x y
                                   s2 ← unifyLists (map (≫=(s1!)) xs) (map (≫=(s1!)) ys)
                                   return (s2 | => s1)
```

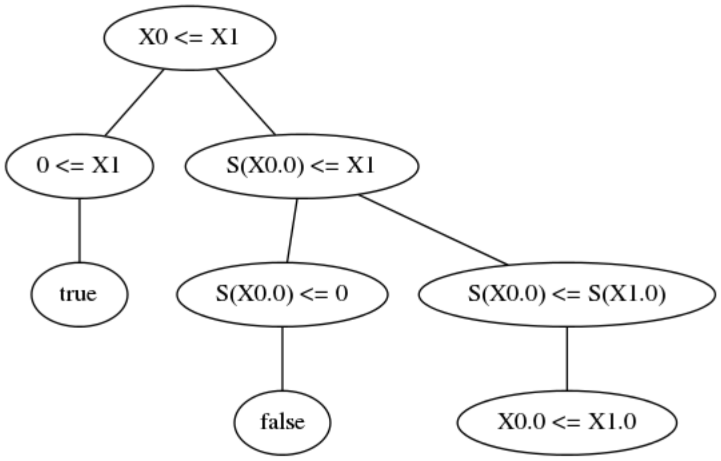
3 Definitional Trees

```
module PDT where
import Term
import Unify
import Subst
import Data.List
import Data.Char (isAlpha)
```

This is the first new idea introduced in this paper. At the high level, a Partial Definitional Tree (or PDT) is a tree for pattern matching rewrite rules. There are three types of patterns in PDTs: a *Branch* which represents matching part of a pattern, a *Rule* which represents matching enough of a pattern to apply a rewrite rule, and an *Error* which represents a failed pattern match. A Definitional Tree is a PDT where the root pattern has only variables. By convention a rule must be operation rooted, that is it must start with a function on the left.

For example consider the function $\leqslant$ defined by
$0 \leqslant_{\rightarrow} true$
$S(X1) \leqslant 0 \rightarrow false$
$S(X1) \leqslant S(X2) \rightarrow X1 \leqslant X2$

This would produce the following Definitional Tree



```
data PDT v = Branch (Term v) Pos [PDT v]
```

```
                | Rule  (Term v) Pos (Term v) (Term v)
                | Error (Term v) Pos deriving (Show)
```

In order to turn a set of rules into Definitional Trees, we group the rules by their outer function symbol, and then turn each group of rules into a Definitional Tree.

```
dt                             :: [Rule Name] → [PDT Name]
dt rs                          = map topdt (splitRules rs)
  where newPat (Fun f ts :=> _) = Fun f (take (length ts) (newVars "x" []))
        topdt r                = pdt (newPat (head r)) [0] r
```

To turn a group of rules with the same function symbol into a Definitional Tree, we need to slowly fill in the variables with constructors until we reach the pattern for a rule. The basic algorithm is simple. Start with a pattern with only variables (i.e. $X1 \leqslant X2$). We split the rules into three cases. Case 1: The pattern we are looking at matches a rule up to variable renaming. Then we just return the rule. Case 2: All of the rules have a variable at the position in the pattern we are looking at. Then we just move on to the next variable in our pattern and start over. Case 3: Some rules have a constructor in the position of our variable. In this case we split all of rules by the constructor in that position, and make a subtree out of each constructor.

If now rules match our patter then this is an error. This actually can't happen, because this implementation forces the system to be orthogonal (or we will discard rules until it is)

```
pdt          :: Term Name → Pos → [Rule Name] → PDT Name
pdt pat p [] = Error pat p
pdt pat p rs
  | ¬ (null rules) = renameRule (head rules)
  | null cons      = pdt pat [head p + 1] (vRules ++ tooShort)
  | otherwise      = Branch pat p $ branches ++ varBranch
  where branches  = map branch cons
        branch cs = pdt (sub cs) (p ++ [0]) cs
        rules     = [Rule pat (init p) l r | (l :=> r) ← rs, pat ˜ = l]
        vRules    = filter unifies rs
        tooShort  = filter ends rs
        cons      = groupBy (leftEq p) (filter differs rs)
        con l              = Con (name l) (take (arity l) (newVars v p))
        sub ((l:=> _): _)  = pat ≫= ((v |− > con (l ! > p))!)
        (Var v)            = pat ! > p
        exists l           = arity (l ! > (init p)) > last p
        ends (l :=> r)     = ¬ (exists l)
        unifies (l :=> r)  = exists l ∧ (l ! > p) ˜ = (pat ! > p)
        differs (l :=> r)  = exists l ∧ ¬ ((l ! > p) ˜ = (pat ! > p))
        varBranch          = if ¬ (null vRules) then [pdt pat [head p + 1] vRules] else []
```

The following functions are all helper functions for *dt* and *pdt*. *leftEq* checks if the left hand side of two rules have the same function/constructor symbol at position *p*. *splitRules* splits the rules based on their outer function symbol. *renameRule* renames a rule based on the variable names in the PDT. *newVars* makes an infinite list of new variables at position *p*. *name* just returns the outer name of a term.

```
leftEq :: (Eq v)                    ⇒ Pos → Rule v → Rule v → Bool
leftEq p (lx:=> _) (ly:=> _) = (lx ! > p) ~~ (ly ! > p)

splitRules :: (Ord v) ⇒ [Rule v] → [[Rule v]]
splitRules = map sort ∘ groupBy (leftEq [ ])

renameRule  :: (Show v, Ord v) ⇒ PDT v → PDT v
renameRule (Rule pat p l r)     = Rule pat p (l ≫= (s!)) (r ≫= (s!))
  where    (Just s)             = unify l pat

newVars        :: Name → Pos → [Term Name]
newVars n p    = [Var ((takeWhile isAlpha n) ++ (dots (p ++ [i]))) | i ← [0 . .]]
  where dots = concat ∘ intersperse "." ∘ map show

name :: Term Name → Name
name (Con c _) = c
name (Fun f _) = f
name (Var v)   = v
```

4 Narrowing

```
module Narrow where
import Term
import Subst
import Unify
import PDT
```

Finally we get to Narrowing. The idea here is that we are evaluating a term while unifying it. This is done by unifying our value with a PDT, and when we encounter a rewrite rule we replace that subterm. This is called a narrowing step. First we need to find the rule we want to use. This can be done by traversing the PDT and unifying with each branch until we hit a rule or an error. Since one term can be rewritten to multiple different terms we return a list of new terms together with substitutions.

```
type Step v = [(Pos, Rule v, Subst v Term)]
```

```
narrow :: (Show v, Ord v)      ⇒ Term v → PDT v → [PDT v] → Step v
narrow t (Rule pat p l r)    _ = [(p, l :=> r, u) | (Just u) ← [unify pat t]]
narrow t (Error pat p)       _ = [(p, E, u)       | (Just u) ← [unify pat t]]
narrow t (Branch pat p ts) dt
   | ¬ (null us)               = [(p′, r, s)            | (ti, u) ← us,
                                                          (p′, r, s) ← narrow t ti dt]
   | otherwise                 = [(p′, r, s | => tau) | (p′, r, s) ← narrow t t′ dt]
     where us          = [(ti, u) | ti ← ts, (Just u) ← [unify (pattern ti) t]]
           (Just tau) = unify pat t
           t′          = getTree ((t ! > p) ≫= (tau!)) dt

getTree :: Term v → [PDT v] → PDT v
getTree t [] = Error t []
getTree t@(Fun f _) (dt@(Branch (Fun g _) _ _) : dts)
   | f ≡ g       = dt
   | otherwise = getTree t dts

pattern (Rule pat _ _ _)  = pat
pattern (Error pat _)      = pat
pattern (Branch pat _ _) = pat
```

Now that we can complete a narrowing step, we need to narrow a term to a normal form. We can do this by narrowing each term to head normal form.

```
nf :: (Show v, Ord v) ⇒ [PDT v] → Term v → [(Term v, Subst v Term)]
nf dts (Con c ts)      = [(Con c (map fst tsu), chain (map snd tsu))
                          | tsu ← sequence (map (nf dts) ts)]
nf dts f@(Fun _ _)    = [(f″, u | => u′) | (f′, u) ← hnf dts f, (f″, u′) ← nf dts f′]
nf dts v@(Var _)      = [(v, emptySubst)]

hnf :: (Show v, Ord v) ⇒ [PDT v] → Term v → [(Term v, Subst v Term)]
hnf _    c@(Con _ _)    = [(c, emptySubst)]
hnf dts f@(Fun _ _)    = [(f′, u′ | => u) | (p, l :=> r, u) ← narrow f (getTree f dts) dts, (f′, u′) ← hnf d
hnf _    v@(Var _)       = [(v, emptySubst)]
```

5 examples that will be fixed later

```
import Term
import Subst
import Unify
import PDT
import Narrow
```

6 Example

The following is a simple example of a rewrite system where equations can be solved using the Narrowing strategy.

$$
\begin{array}{l}
z = \mathit{Con}\ \texttt{"0"}\ [\,] \\
s\ x = \mathit{Con}\ \texttt{"S"}\ [x] \\
x = \mathit{Var}\ \texttt{"X"} \\
y = \mathit{Var}\ \texttt{"Y"} \\
t = \mathit{Con}\ \texttt{"true"}\ [\,] \\
f = \mathit{Con}\ \texttt{"false"}\ [\,] \\
x <-= y = \mathit{Fun}\ \texttt{"<="}\ [x, y] \\
\mathit{half}\ x = \mathit{Fun}\ \texttt{"half"}\ [x] \\
x \mathbin{/\!\backslash} y = \mathit{Fun}\ \texttt{"/\\\\"}\ [x, y] \\
x =:= y = \mathit{Fun}\ \texttt{"=:="}\ [x, y]
\end{array}
$$

$$
\begin{array}{lll}
\mathit{rules} = [& (z <-= x) & :=> t, \\
 & (s\ x <-= z) & :=> f, \\
 & (s\ x <-= s\ y) & :=> (x <-= y), \\
 & (\mathit{half}\ z) & :=> z, \\
 & (\mathit{half}\ (s\ z)) & :=> z, \\
 & (\mathit{half}\ (s\ (s\ x))) & :=> (\mathit{half}\ x), \\
 & t \mathbin{/\!\backslash} x & :=> x, \\
 & f \mathbin{/\!\backslash} x & :=> f, \\
 & (t =:= t) & :=> t, \\
 & (f =:= f) & :=> t, \\
 & (z =:= z) & :=> t, \\
 & ((s\ x) =:= (s\ y)) & :=> (x =:= y)]
\end{array}
$$

References

[1] Antoy, Sergio *A Needed Narrowing Strategy* 1994, POPL 94.