Normalizing Equality Constraints

Iavor S. Diatchki

Introduction

This is a literate Haskell script, which provides an implementation of a simple constraint solver, similar to the one used by GHC. Our purpose is to understand GHC's reasoning about equality, as such we do not consider the part of the constraint solver dealing with solving type-classes.

The implementation uses only standard Haskell libraries:

```
{-# LANGUAGE TypeSynonymInstances, FlexibleInstances #-}
module ConstraintSolver where

import           Data.List(mapAccumL)
import           Data.Maybe(fromMaybe)
import           Data.Either(partitionEithers)
import           Data.Map(Map)
import qualified Data.Map as Map
import           Control.Monad(msum,guard)
```

Basic Types

The constraint solver uses the following datatype to represent Haskell types:

```
data Type = TVar TVar          -- ^ Type variables
          | TCon TCon          -- ^ Type constructor
          | TApp Type Type     -- ^ Data-type application
          | TFun TFun [Type]   -- ^ Fully applied type function
            deriving Eq
```

This is quite similar to GHC's representation of types, except that we don't worry about universal quantifiers and qualified types. The types `TVar`, `TCon`, and `TFun` are used to represent the names of type variables, type constructors, and type functions, respectively:

```haskell
type TVar = String
type TCon = String
type TFun = String
```

Here we use `String` for all of these types, but any other type would do. All we need is a notion of equality, and an ordering for the names of type variables.

Simple Types

Throughout most of the implementation we will work with types that do not contain any type function applications. We call such types *simple*, and we use a type synonym to make it explicit when we are working with simple types:

```haskell
-- | A type that does not contain `TFun`
type SimpleType = Type
```

Type Equations

Two specify equations between types we use the following type:

```haskell
-- | An equation between full or simple types.
data Eqn t = t :=: t
```

We use `Eqn Type` for equations between arbitrary types, and `Eqn SimpleType` for equations between simple types.

Of special interest are equations of the form `F ts ~ t`, where `F` is a type function, `ts` is a list of simple types, and `t` is a simple type. While such equations may already be represented as `TFun f ts :=: t`, they are common enough that we define a custom type to capture this pattern:

```haskell
-- | An equation of the form: @F ts ~ t@
data TFunEqn    = TFunEqn TFun [SimpleType] SimpleType
```

The Work Queue

The work queue is the part of the constraint solver, which keeps track of constraints that are waiting to be processed by the main algorithm. The work queue is defined as follows:

```haskell
-- | The work queue
data WorkQueue = WorkQueue
  { freshTVars :: [TVar]     -- ^ fresh type variables (infinite)
  , simpleEqns :: [ Eqn SimpleType ]  -- ^ @s ~ t@
  , tfunEqns   :: [ TFunEqn ]         -- ^ @F ts ~ t@
  }
```

The following functions are used to manipulate the work queue by adding and removing equations, and also generating fresh type variables:

```haskell
-- | An empty work queue.
-- The seed is used to generate fresh type variables.
emptyWorkQueue :: String -> WorkQueue
emptyWorkQueue seed = WorkQueue { freshTVars = map toVar [ 0 .. ]
                                , simpleEqns = []
                                , tfunEqns   = []
                                }
  where toVar x = seed ++ show (x :: Integer)

-- | Add a new simple equation to the work queue.
addSimpleEqn :: Eqn SimpleType -> WorkQueue -> WorkQueue
addSimpleEqn eqn q = q { simpleEqns = eqn : simpleEqns q }

-- | Remove one of the simple equations from the work queue.
getSimpleEqn :: WorkQueue -> Maybe (WorkQueue, Eqn SimpleType)
getSimpleEqn q = case simpleEqns q of
                   e : es -> return (q { simpleEqns = es }, e)
                   []     -> Nothing

-- | Add a type-function equation to the work queue.
addTFunEqn :: TFunEqn -> WorkQueue -> WorkQueue
addTFunEqn eqn q = q { tfunEqns = eqn : tfunEqns q }

-- | Remove a type-function equation from the work queue.
getTFunEqn :: WorkQueue -> Maybe (WorkQueue, TFunEqn)
getTFunEqn q = case tfunEqns q of
                 e : es -> return (q { tfunEqns = es }, e)
                 []     -> Nothing

-- | Make up a fresh type variable.
newTVar :: WorkQueue -> (WorkQueue, TVar)
newTVar q = case freshTVars q of
              x : xs -> (q { freshTVars = xs }, x)
              []     -> error "[bug] Out of fresh variables."
```

Conversion To Simple Types

Given an arbitrary type, we may convert it to a simple type by providing explicit names for terms that are applications of type functions. For example, if `F` is a type function, then we may think of the type `(F Int, F Char)`, as `(a,b)` with the additional equations `(F Int ~ a, F Char ~ b)`.

This process is formalized with the function `toSimpleType`:

```
-- | Name all occurrences of type functions.
toSimpleType :: WorkQueue -> Type -> (WorkQueue, SimpleType)
toSimpleType s ty =
  case ty of
    TVar _      -> (s, ty)
    TCon _      -> (s, ty)
    TApp t1 t2  -> let (s1,t1') = toSimpleType s  t1
                       (s2,t2') = toSimpleType s1 t2
                   in (s2, TApp t1' t2')
    TFun f ts ->
      let (s1, xs)  = mapAccumL toSimpleType s ts
          (s2, x)   = newTVar s1
          newT      = TVar x
      in  (addTFunEqn (TFunEqn f xs newT) s2, newT)
```

Equation generated in the process of simplifying a type are added to the work queue. Once we have `toSimpleType` it is easy to add arbitrary equations to the work queue:

```
-- | Add an arbitrary equation to the work queue.
addEqn :: Eqn Type -> WorkQueue -> WorkQueue
addEqn (t1 :=: t2) s =
  let (s1, t1') = toSimpleType s  t1
      (s2, t2') = toSimpleType s1 t2
  in addSimpleEqn (t1' :=: t2') s2
```

The State of Constraint Solver

The state of the constraint solver has two parts: the *work queue*, which contains constraints that need to be processed, and the *inert set*, which contains the constraints that have been processed:

```
data State  = State { inerts :: Inerts, canEqs :: WorkQueue }
```

The equations in the inert set are normalized, so that they do not contain redundant information: equations between simple types are reduced to a substitution, and unsolved equations between functions are fully evaluated, and correspond to distinct (incomparable) function invocations. Also, the substitution is fully applied to the inert function equations.

```
data Inerts = Inerts { inertSubst :: Subst
                     , inertFuns  :: [TFunEqn]
                     }
```

Next are a few convenient functions for manipulating the state:

```
-- | Modify the work queue.
updWorkQ :: State -> (WorkQueue -> WorkQueue) -> State
updWorkQ s f = s { canEqs = f (canEqs s) }

-- | Extract the next simple equation, if any.
nextSimpleEqn :: State -> Maybe (State, Eqn SimpleType)
nextSimpleEqn s = do (q1,e) <- getSimpleEqn (canEqs s)
                     return (s { canEqs = q1 }, e)

-- | Extract the next functions equation, if any.
nextTFunEqn :: State -> Maybe (State, TFunEqn)
nextTFunEqn s = do (q1,e) <- getTFunEqn (canEqs s)
                   return (s { canEqs = q1 }, e)

-- | Add an inert function equation to the state.
addInert :: TFunEqn -> State -> State
addInert i s = s { inerts = is { inertFuns = i : inertFuns is } }
  where is = inerts s
```

Now, we may define a function that will compute the initial state of the solver from a collection of equations:

```
prepare :: String -> [Eqn Type] -> State
prepare seed eqns =
  State { inerts = Inerts { inertSubst = Map.empty
                          , inertFuns  = []
                          }
        , canEqs = foldr addEqn (emptyWorkQueue seed) eqns
        }
```

The function `prepare` converts the equations into constraints on simple types and adds them to the work queue.

For the rest of the document, we consider how to move constraints from the work queue into the inert set.

Substitutions

A substitution captures a mapping from type variables to simple types:

```
-- | A substitution
type Subst = Map TVar SimpleType
```

A substitution may be applied to a type, to replace the corresponding type variables with the types in the substitution. It is convenient to apply substitutions not just to types, but also to data-structures containing types (e.g., a list of types). To avoid a proliferation of function names we use a type class, which specifies exactly which types support substitution application:

```
class ApSubst t where
  -- | Apply a substitution, returning 'Nothing' if
  -- the substitution did not change anything.
  apSubstMb :: Subst -> t -> Maybe t
```

The method in the class applies a substitution to a structure and returns `Just newStructure`, if the substitution instantiated at least one variables in the structure, and `Nothing` otherwise.

Sometimes, we just need to apply a substitution and we don't need to know if anything was instantiated, which is why we define `apSubst`:

```
-- | Apply a substitution to some structure.
apSubst :: ApSubst t => Subst -> t -> t
apSubst su t = fromMaybe t (apSubstMb su t)
```

We also need to compose substitutions:

```
-- | Composed two substitutions.
composeSubst :: Subst -> Subst -> Subst
composeSubst g f = Map.union (apSubst g <$> f) g
```

Substitutions are composed in the same order as functions are composed in Haskell—the second argument is the first substitution, and the first argument is the second substitution.

If we know how to apply a substitution to two different objects, then we may apply a substitution to a pair of the objects. The pair is affected by the substitution if *either* of the objects is affected:

```haskell
instance (ApSubst a, ApSubst b) => ApSubst (a,b) where
  apSubstMb su (a,b)
    | Just a' <- mb1 = return (a', fromMaybe b mb2)
    | Just b' <- mb2 = return (a, b')
    | otherwise      = Nothing
    where
    mb1 = apSubstMb su a
    mb2 = apSubstMb su b
```

Once we know how to apply a substitution to two objects, it is easy to generalize to a list of values:

```haskell
instance ApSubst a => ApSubst [a] where
  apSubstMb _ []         = Nothing
  apSubstMb su (x : xs) = uncurry (:) <$> apSubstMb su (x,xs)
```

Perhaps the most interesting instance is applying a substitution to a type:

```haskell
instance ApSubst Type where
  apSubstMb su ty =
    case ty of
      TVar x     -> Map.lookup x su
      TApp t1 t2 -> uncurry TApp <$> apSubstMb su (t1,t2)
      TCon _     -> Nothing
      TFun f ts  -> TFun f <$> apSubstMb su ts
```

Using the above instances we also define how to apply substitutions to equations, function equations, and the work queue:

```haskell
instance ApSubst TFunEqn where
  apSubstMb su (TFunEqn tf ts t) =
    uncurry (TFunEqn tf) <$> apSubstMb su (ts,t)

instance ApSubst t => ApSubst (Eqn t) where
  apSubstMb su (t1 :=: t2) =
    uncurry (:=:) <$> apSubstMb su (t1,t2)

instance ApSubst WorkQueue where
  apSubstMb su q =
    do (es,fs) <- apSubstMb su (simpleEqns q, tfunEqns q)
       return q { simpleEqns = es, tfunEqns = fs }
```

Solving Simple Equations

An equation between simple types is satisfiable if we can compute the most general unifier (MGU) of the two types. If the MGU does not exist, then the equation is equivalent to `False`.

```
-- | Compute the most general unifier of two types.
mgu :: SimpleType -> SimpleType -> Maybe Subst
mgu (TVar x) ty = bindVar x ty
mgu ty (TVar x) = bindVar x ty
mgu (TApp t1 t2) (TApp s1 s2) =
  do su1 <- mgu t1 s1
     su2 <- mgu (apSubst su1 t2) (apSubst su1 s2)
     return (composeSubst su2 su1)
mgu (TCon c) (TCon d) | c == d = return Map.empty
mgu _ _ = Nothing
```

The first two cases of the definition describe what happens if one of the types is a variable:

```
-- | Bind a variable to a type.
bindVar :: TVar -> SimpleType -> Maybe Subst
bindVar x (TVar y) | x == y = return Map.empty
bindVar x t | occurs t  = Nothing
            | otherwise = return (Map.singleton x t)
  where
  occurs ty = case ty of
                TVar y    -> x == y
                TCon _    -> False
                TApp a b  -> occurs a || occurs b
                TFun _ _  -> error "[bug] Not a simple type"
```

In this case, we check that the variable does not occur in the other type, as if it did a solution would require an infinite type, and there are no such types in Haskell.

Definitions of Type Functions

In this section we describe how we represent definitions for type functions. Type functions are defined via equations schemes, which correspond to type family instances in Haskell: the left-hand side of a definitional equation for a type function has a list of patterns, and the right-hand side is a type. If some concrete types match the patterns on the left-hand side, then the function "evaluates" to the right-hand side:

```haskell
-- | A type "pattern" used to define equations for type functions.
type TPat     = SimpleType
data TFunDef  = TFunDef TFun [TPat] Type
```

To check if a particular type matches a type pattern we use *matching*. Matching is a little like one-side unification:

```haskell
-- | Check if a type matches a type pattern.
-- If successful
match :: TPat -> SimpleType -> Maybe Subst
match (TApp tp1 tp2) (TApp t1 t2) = matches [tp1,tp2] [t1,t2]
match (TCon c) (TCon d) | c == d  = return Map.empty
match (TVar a) ty                 = return (Map.singleton a ty)
match _ _                         = Nothing
```

Note that the variables bound by the substitution computed by matching are the variables of the type pattern. These are conceptually different from the variables in the type, even if they might appear syntactically equal. This substitution is only ever applied to the right-hand side of a definition.

Type functions may have multiple parameters, so we need to match multiple patterns. Indeed, even to complete a single match, we may need to match multiple sub-patterns. The function `matches` shows how to match multiple patterns consistently:

```haskell
-- | Match a list of patterns with a list of simple types.
matches :: [TPat] -> [SimpleType] -> Maybe Subst
matches (tpat : tpats) (ty : tys) =
  do inst1 <- match   tpat  ty
     inst2 <- matches tpats tys
     let common = Map.intersectionWith (==) inst1 inst2
     guard (and common) -- Same on overlap
     return (Map.union inst1 inst2)
matches [] [] = return Map.empty
matches _ _   = Nothing
```

The general idea is simple: to match multiple patters, we match the patterns individually, but we make sure that if a single pattern variable is bound by multiple matches, then all the bindings are equal. We need this check because type patterns may be non-linear (i.e., the same variable might appear multiple times in a pattern).

Once we know how to match patterns, it is simple to check if a particular function application can be reduced using a particular equation, and it is simple to generalize to a collection of equations:

```
-- | See if a particular function equation applies.
useDefn :: TFunDef -> TFun -> [SimpleType] -> Maybe Type
useDefn (TFunDef tf pats ty) tf' tys =
  do guard (tf == tf')
     inst <- matches pats tys
     return (apSubst inst ty)

-- | Try to reduce one step, using the given definitions.
lookupDef :: [TFunDef] -> TFun -> [SimpleType] -> Maybe Type
lookupDef allDefs tf args =
  msum [ useDefn def tf args | def <- allDefs ]
```

Solving Type Function Equations

Next we consider how to normalize equations involving type functions. Before we add such an equation to the inert set we need to check that 1) the equation cannot be reduced using a definition, and 2) the equation is not equivalent to another equation, which is already present in the inert set.

We start by trying to evaluate the function:

```
-- | Try to resolve a type-function constraint using a definition.
interactWithDef :: [TFunDef] -> TFunEqn -> State -> Maybe State
interactWithDef defs (TFunEqn tf args res) s =
  do t <- lookupDef defs tf args
     return
       $ updWorkQ s $ \q -> let (q',t') = toSimpleType q t
                            in addSimpleEqn (res :=: t') q'
```

The function `interactWithDef` tries to reduce the left-hand side of the equation one step, using a definition. If this succeeds, then we add a new equation, stating that the reduced function equals to the right-hand side of the equation.

If evaluation fails, then we still need to check that the new equation is truly different form other equations in the inert set. So, if we are working with an equation `F ts ~ t` and the inert set already contains `F ts ~ s`, then instead of adding the function equation, we should simply add `t ~ s`. This is encoded as follows:

```
-- | Try to resolve a type-function constraint using an inert.
interactWithInert :: TFunEqn -> TFunEqn -> State -> Maybe State
interactWithInert (TFunEqn tf args res) (TFunEqn tf1 args1 res1) s =
  do guard (tf == tf1 && args == args1)
     return $ updWorkQ s $ addSimpleEqn (res :=: res1)
```

Finally, an equation is considered "known" if it can be resolved either by evaluation, or by another inert:

```
-- | Is this equation redundant?
-- Returns 'Nothing' if the equation is not redundant,
-- and so should be added to the inert set.
isKnownFun :: [TFunDef] -> TFunEqn -> State -> Maybe State
isKnownFun defs tf s =
  msum ( interactWithDef defs tf s
       : [ interactWithInert tf i s | i <- inertFuns (inerts s) ]
       )
```

The Constraint Solver

Now we are ready to put everything together, by showing how the constraint solver orchestrates the normalization of equations. First, we define a single step of the solver, which may result in one of the following outcomes:

```
data Result = Done Inerts
            | Continue State
            | Inconsistent
```

When the solver performs its last step, it returns `Done`. This means that the work queue became empty, and so the inert set contain a normalized form of the initial equations.

It is possible the while normalizing the equations, the solver notices that they are inconsistent. In that case, it will return `Inconsistent`.

Finally, while it is working, the solver returns `Continue` with the new state.

A Single Step

A single step will take a constraint from the work queue and simplify it. If the constraint can't be fully simplified than it will be added to the inert set, and the process will continue.

```
-- | Process one constraint from the work queue.
makeStep :: [TFunDef] -> State -> Result
makeStep defs s =
  case nextSimpleEqn s of

    Just (s1, t1 :=: t2) ->
```

```
      case mgu t1 t2 of
        Nothing -> Inconsistent
        Just su -> Continue (apSubstState su s1)

    Nothing ->
      case nextTFunEqn s of
        Nothing       -> Done (inerts s)
        Just (s1, tf) ->
          case isKnownFun defs tf s1 of
            Just s2 -> Continue s2
            Nothing -> Continue (addInert tf s1)
```

Note that we process simple equations before function equations. The reason for this is the when we solve simple equations, we may learn new information about variables, which in turn might help us simplify function equations. Thus, whenever a substitution defines a variable, we have to reconsider previously inert functions, because it is possible that now they may be simplified further. This process is formalized in `apSubstState`:

```
-- | Apply a substitution to the state.
apSubstState :: Subst -> State -> State
apSubstState su s =
  State { inerts = newInerts
        , canEqs = foldr addTFunEqn (apSubst su (canEqs s)) kicked
        }
  where
  oldInerts = inerts s
  newInerts = Inerts { inertSubst = composeSubst su (inertSubst oldInerts)
                     , inertFuns  = stay
                     }

  (stay,kicked) = partitionEithers (map check (inertFuns oldInerts))

  check a       = case apSubstMb su a of
                    Just a' -> Right a'
                    _       -> Left a
```

The Main Loop

The main loops repeated applies the stepping function until there is no more work to be done, or it detects an inconsistency:

```
-- | Step until finished.
makeSteps :: [TFunDef] -> State -> Maybe Inerts
```

```
makeSteps defs s =
  case makeStep defs s of
    Done s1       -> Just s1
    Continue s1   -> makeSteps defs s1
    Inconsistent  -> Nothing

-- | Normalize a collection of equations.
normalize :: String -> [TFunDef] -> [Eqn Type] -> Maybe Inerts
normalize seed defs = makeSteps defs . prepare seed
```