J. Functional Programming **8** (5): 1–10, September 1998. Printed in the United Kingdom

FUNCTIONAL PEARLS

Polytypic Unification

Patrik Jansson and Johan Jeuring

Chalmers University, S-412 96 Göteborg, Sweden.

Utrecht University, Utrecht, The Netherlands

E-mail: `..patrikj..@cs..chalmers..se, johanj@cs.uu.nl`

Abstract

Unification, or two-way pattern matching, is the process of solving an equation involving two first-order terms with variables. Unification is used in type inference in many programming languages and in the execution of logic programs. This means that unification algorithms have to be written over and over again for different term types. Many other functions also make sense for a large class of datatypes; examples are pretty printers, equality checks, maps etc. They can be defined by induction on the structure of user-defined datatypes. Implementations of these functions for different datatypes are closely related to the structure of the datatypes. We call such functions polytypic. This paper describes a unification algorithm parametrised on the type of the terms and shows how to use polytypism to obtain a unification algorithm that works for all regular term types.

1 Introduction

In simple pattern matching, a pattern (a string containing wild cards) is matched with a normal string to determine if the string is an instance of the pattern. This can be generalised in at least two directions; we can allow the second string to contain wild cards too, thus making the matching symmetric, or we can allow more complicated terms than strings. By combining these two generalisations we obtain unification. A unification algorithm tries to find a *most general unifier* (*mgu*) of two terms. The most general unifier of two terms is the smallest substitution of terms for variables such that the substituted terms become equal.* Use of unification is widespread; it is used in type inference algorithms, rewriting systems, compilers, etc. (Knight, 1989).

Descriptions of unification algorithms normally deal with a general datatype of terms, containing variables and applications of constructors to terms, but each real implementation uses one specific instance of terms and a specialised version of the algorithm for this term type. This paper describes a functional unification program that works for all regular term types. The program is an example of a *polytypic function* (Jeuring, 1995).

* If two first order terms are unifiable, their *mgu* is unique (Robinson, 1965).

Function `length :: List a -> Int`, which counts the number of occurrences of `a`'s in a list, and the similar function `numOfElems :: Tree a -> Int`, which counts the number of occurrences of `a`'s in a tree are both instances of a more general function `size :: d a -> Int`. Function `size` is not only polymorphic in `a`, but also in the type constructor `d`. In the same way we can generalise the function `map :: (a -> b) -> List a -> List b` into a function `pmap :: (a -> b) -> d a -> d b`, so that it too works for trees and other similar datatypes. We call such functions polytypic functions. For an introduction to the basic ideas of polytypic functions see (Jeuring & Jansson, 1996) and for a more theoretical treatment of polytypism (Bird *et al.*, 1996) and (de Moor, 1994).

In this paper we show that

- by parametrising the unification algorithm by the datatype for terms, we can separate the core of the algorithm from the parts depending on the specific datatype, and
- by abstracting away from the type constructor dependence in the datatype dependent part we obtain a polytypic program.

Thus we have *one* implementation of unification that works for many *different* term types leaving the specialisation to the compiler.

The core of the unification algorithm is written in Haskell and the polytypic part is written in the Haskell extension PolyP (Jansson & Jeuring, 1997). The full code is available from `http://www.cs.chalmers.se/~patrikj/unify/`.

2 Unification

In this section we will specify and implement a functional unification algorithm. We start with an example. Consider the unification of the two terms $\boldsymbol{f}(x, \boldsymbol{f}(\boldsymbol{a}, \boldsymbol{b}))$ and $\boldsymbol{f}(\boldsymbol{g}(y, \boldsymbol{a}), y)$, where x and y are variables and $\boldsymbol{f}$, $\boldsymbol{g}$, $\boldsymbol{a}$ and $\boldsymbol{b}$ are constants. Since both terms have an $\boldsymbol{f}$ on the outermost level, these expressions can be unified if x can be unified with $\boldsymbol{g}(y, \boldsymbol{a})$, and $\boldsymbol{f}(\boldsymbol{a}, \boldsymbol{b})$ can be unified with y. As these two pairs of terms are unified by the substitution $\sigma = \{x \mapsto \boldsymbol{g}(y, \boldsymbol{a}), y \mapsto \boldsymbol{f}(\boldsymbol{a}, \boldsymbol{b})\}$, the original pair of terms is also unified by applying the substitution σ, yielding the unified term $\boldsymbol{f}(\boldsymbol{g}(\boldsymbol{f}(\boldsymbol{a}, \boldsymbol{b}), \boldsymbol{a}), \boldsymbol{f}(\boldsymbol{a}, \boldsymbol{b}))$.

2.1 Terms

In the unification literature, a term is usually defined as either a variable or an application of a constructor to zero or more terms. (*Var* is a set of variables and *Con* is a set of constructor constants.)

$$T \ ::= \ v \mid \boldsymbol{c}(T_1, \ldots, T_{arity(\boldsymbol{c})}), \quad v \in \mathit{Var}, \quad \boldsymbol{c} \in \mathit{Con}$$

We instead focus on the three properties of the type of terms we need to define unification. We need to know

- the children (immediate subterms) of a term and how to update them;

```
class Children t   where   children      ::   t → [t]
                           mapChildren   ::   (t → t) → t → t
class VarCheck t   where   varCheck      ::   t → Maybe Var
class TopEq t      where   topEq         ::   t → t → Bool

class (Children t, VarCheck t, TopEq t) ⇒ Term t

class Subst s      where   idSubst       ::   s t
                           modBind       ::   (Var,t) → s t → s t
                           lookupIn      ::   s t → Var → Maybe t
```

Fig. 1. Terms and substitutions

```
data T = V Var | App Con [T]

instance Children T where children        (V v)        = []
                          children        (App c ts)   = ts
                          mapChildren f   (V v)        = V v
                          mapChildren f   (App c ts)   = App c (map f ts)

instance VarCheck T where varCheck        (V v)        = Just v
                          varCheck        _            = Nothing

instance TopEq T    where topEq (App c l) (App c' l') = c == c' &&
                                                      length l == length l'
                          topEq (V v)     (V w)        = v == w
                          topEq _         _            = False
```

Fig. 2. T is an instance of `Term`.

- whether or not a term is a variable, and if it is, which variable;
- when two terms are top-level equal. (Think of 'top-level equal' terms as terms with 'equal outermost constructors'.)

We define one type class for each of these properties and define the class of terms to be the intersection of these three classes (see figure 1). As an example, the instances for the type T above are given in figure 2.

2.2 Substitutions

A substitution is a mapping from variables to terms leaving all but a finite number of variables unchanged. We define a class† of substitutions parametrised on the type of terms by the three class members `idSubst`, `modBind` and `lookupIn`. The value `idSubst` represents the identity substitution, the call `modBind (v,t) s` modifies the substitution `s` to bind `v` to `t` (leaving the bindings for other variables unchanged) and `lookupIn s v` looks up the variable `v` in the substitution `s` (giving `Nothing` if the variable is not bound in `s`). Using `lookupIn` a substitution can be viewed as a

† This class is not essential for the results in the paper, we could just as well have used some specific type constructor for substitutions instead.

function from *variables* to terms. To use substitutions as functions from *terms* to terms we define `appSubst`:

```
appSubst :: (Subst s, Term t) => s t -> t -> t
appSubst s t = case varCheck t of
                 Nothing -> mapChildren (appSubst s) t
                 Just v  -> case lookupIn s v of
                              Nothing -> t
                              Just t' -> appSubst s t'
```

When calling `appSubst s t`, the substitution `s` is applied to all variables in the term `t` and also recursively to all variables in the substituted terms.

A unifier of two terms is a substitution that makes the terms equal. A substitution σ is at least as general as a substitution σ' if and only if σ' can be factored by σ, i.e. if there exists a substitution ρ such that $\sigma' = \rho \circ \sigma$, where we treat substitutions as functions. (In Haskell notation this means that `s` is at least as general as `s'` iff there exists `r` such that `appSubst s' = appSubst r . appSubst s`.)

We want to define a function that given two terms finds the most general substitution that unifies the terms or, if the terms are not unifiable, reports this.

2.3 The unification algorithm

Function `unify` takes two terms, and returns their most general unifier. It is implemented in terms of `unify'`, which updates a current substitution that is passed around as an extra argument. The unification algorithm starts with the identity substitution, traverses the terms and tries to update the substitution (as little as possible) while solving the constraints found. If this succeeds the resulting substitution is a most general unifier of the terms. The algorithm distinguishes three cases depending on whether or not the terms are variables.

- If neither term is a variable we have two sub-cases; either the constructors of the terms are different (that is the terms are not top level equal) and the unification fails, or the constructors are equal and we unify all the children pairwise.
- If both terms are variables and the variables are equal we succeed without changing the substitution. (If the variables are not equal the case below matches.)
- If one of the terms is a variable we try to add to the substitution the binding of this variable to the other term. This succeeds if the variable does not occur in the term and if the new binding of the variable can be unified with the old binding (in the current substitution).

A straightforward implementation of this description gives the code in figure 3 using the auxiliary functions in figure 4. We use some functions and types from the Haskell prelude: functions `return` and `(>>=)` for the monad operations, the type `Maybe a` and the function `maybe` for error handling and later the type `Either` and the function `either` for disjoint sums.
To use this unification algorithm on some term type `T` we must make `T` an instance of the class `Term` by defining the four functions `children`, `mapChildren`, `varCheck`

```
unify  :: (Term t,Subst s) => t -> t -> Maybe (s t)
unify' :: (Term t,Subst s) => t -> t -> s t -> Maybe (s t)

unify  tx ty = unify' tx ty idSubst
unify' tx ty = uni (varCheck tx,varCheck ty) where
  uni (Nothing,Nothing) | topEq tx ty = uniTerms tx ty
                        | otherwise   = err
  uni (Just i, Just j ) | i == j      = ok
  uni (Just i, _      )               = i |-> ty
  uni ( _,     Just j )               = j |-> tx

  uniTerms x y = threadList (zipWith unify' (children x) (children y))

(|->) :: (Term t, Subst s) => Var -> t -> s t -> Maybe (s t)
(i |-> t) s = if occursCheck i s t then err s
              else case lookupIn s i of
                     Nothing -> ok (modBind (i,t) s)
                     Just t' -> mapMaybe (modBind (i,t)) (unify' t t' s)
```

Fig. 3. The core of the unification algorithm

```
vars   :: (Children t,VarCheck t) => t -> [Var]
vars t =  [ v | Just v <- map varCheck (subTerms t)]

subTerms :: Children t => t -> [t]
subTerms t = t : concat (map subTerms (children t))

occursCheck :: (Term t,Subst s) => Var -> s t -> t -> Bool
occursCheck i s t = i `elem` reachlist (vars t)
  where
    reachlist l = l ++ concat (map reachable l)
    reachable v = reachlist (maybe [] vars (lookupIn s v))

threadList :: Monad m => [a -> m a] -> a -> m a
threadList = foldr (@@) return

(@@) :: Monad m => (a -> m b) -> (c -> m a) -> (c -> m b)
(f @@ g) x = g x >>= f

mapMaybe :: (a -> b) -> Maybe a -> Maybe b
mapMaybe f = maybe Nothing (Just . f)

ok, err  :: a -> Maybe a
ok       = Just
err      = const Nothing
```

Fig. 4. Auxiliary functions in the unification algorithm

and `topEq`. Traditionally these instances would be handwritten for the type `T` and when we need unification on a different type we would need new instances. But we can do better than that; by making these functions polytypic we get one description that automatically generates instances for all term types.

3 Polytypic unification

A polytypic function is a function parametrised on type constructors. Polytypic functions are defined either by induction on the structure of user-defined datatypes or defined in terms of other polytypic (and non-polytypic) functions. We express polytypic functions in the Haskell extension PolyP (Jansson & Jeuring, 1997).

To make the unification algorithm polytypic we define `children`, `mapChildren`, `topEq` and `varCheck` for all term types `d a`, i.e. we express them as polytypic functions. In the following subsections we will describe the polytypic functions we need for unification and how they are expressed in PolyP.

3.1 Polytypic notation

To abstract away from the specific type constructors we view all datatypes as fixpoints of functors, and extend the type language to include functor building blocks. A category theoretic functor is a structure preserving mapping between two categories, see for example (Pierce, 1991). In this paper we take a more concrete view of functors and use them only to model the structure of datatypes.

Functors are built up from the constants `Par` for the parameter, `Rec` for recursive occurrences of the datatype and `Const t` for constant types (`Int`, `Bool` etc.) used in the datatype definition, `Empty` and the combinator `*` for products, `+` for alternatives and `@` for type application. See figure 5 for some examples.

With a recursive datatype $d\ a$ as a fixpoint, `inn` and `out` are the fold and unfold isomorphisms showing $d\ a \cong F_d\ a\ (d\ a)$.

```
inn  ::  d a ← (FunctorOf d) a (d a)
out  ::  d a → (FunctorOf d) a (d a)
```

To construct values of type `d a` we use `inn`, which effectively combines all the constructors of the datatype in one function, and conversely, to deconstruct values

```
data List a    = Nil          | Cons  a  (List a)
-- FunctorOf List = Empty     +      Par *  Rec
data Tree a   = Null          | Node (Tree a)   a   (Tree a)
-- FunctorOf Tree = Empty     +         Rec  * Par *  Rec
data Type c   = TyVar  Var    | Apply  c  (List (Type c))
-- FunctorOf Type = Const Var +       Par*(List @ Rec)
```

Fig. 5. Datatypes and functors.

of type `d a` we use `out` instead of pattern matching. PolyP defines `inn` and `out` for all regular‡ datatypes.

As we deal with recursive datatypes with one type parameter, the corresponding functors are bifunctors. The bifunctors built from the constructors above map two types to a type. We call the part of a bifunctor that maps two functions to a function `fmap`. The operators `(-*-)` and `(-+-)` are the "`fmaps`" for the pair type constructor `(,)` and the sum type constructor `Either`:

```
(-*-) :: (a -> c) -> (b -> d) -> (a,b) -> (c,d)
(f -*- g) (x,y) = (f x , g y)

(-+-) :: (a -> c) -> (b -> d) -> Either a b -> Either c d
(f -+- g) = either (Left . f) (Right . g)
```

The definition of `fmap` by induction over the structure of functors is given in figure 6. (The subscripts indicating the type are included for readability and are not part of the definition.) Using `fmap` we can define `pmap`, a polytypic variant of the Haskell

```
polytypic fmap_f :: (a -> c) -> (b -> d) -> f a b -> f c d
    = \p r -> case f of
                  g + h     ->  (fmap_g p r) -+- (fmap_h p r)
                  g * h     ->  (fmap_g p r) -*- (fmap_h p r)
                  Empty     ->  id
                  Par       ->  p
                  Rec       ->  r
                  d @ g     ->  pmap_d (fmap_g p r)
                  Const t   ->  id
```

Fig. 6. The polytypic `fmap` function.

function `map :: Functor d => (a -> b) -> d a -> d b`. Just like `map`, function `pmap` applies its function argument to all elements in a structure without affecting the shape of the structure. But unlike `map`, whose instances for different datatypes have to be written by hand, instances for `pmap` are automatically generated by PolyP.

```
pmap :: Regular d => (a -> b) -> d a -> d b
pmap f = inn . fmap f (pmap f) . out

instance Regular d => Functor d where  map = pmap
```

The function `mapChildren` is also expressed in terms of `fmap`.

3.2 Functions `children` *and* `mapChildren`

Function `children :: Children t => t -> [t]` returns the immediate subterms of a term. We find these subterms by unfolding the term one level, using `out`,

‡ A datatype `d a` is regular if it contains no function spaces, and if the argument of the type constructor `d` is the same on the left- and right-hand side of its definition.

mapping the parameters to empty lists and the subterms to singletons using `fmap` and flattening the result to a list using `fflatten`:

```
instance Regular d => Children (d a) where
  children      = fflatten . fmap nil singleton . out
  mapChildren f = inn      . fmap id  f         . out

nil x = []
singleton x = [x]
```

Function `fflatten :: f [a] [a] -> [a]` takes a value `v` of type `f [a] [a]`, and returns the concatenation of all the lists (of type `[a]`) occurring at the top level in `v`. The polytypic definition of `fflatten` is given in figure 7. As an example, we

```
polytypic fflatten_f :: f [a] [a] -> [a]
    = case f of
             g + h     ->   either fflatten_g fflatten_h
             g * h     ->   \(x,y) -> fflatten_g x ++ fflatten_h y
             Empty     ->   nil
             Par       ->   id
             Rec       ->   id
             d @ g     ->   concat . flatten_d . pmap_d fflatten_g
             Const t   ->   nil

flatten :: Regular d => d a -> [a]
flatten = fflatten . fmap singleton flatten . out
```

Fig. 7. The polytypic `fflatten` function.

unfold the definition of `fflatten` when used on the type `List a` (remember that `FunctorOf List` = `Empty+Par*Rec`):

```
fflatten_{Empty+Par*Rec}
  = either fflatten_{Empty} fflatten_{Par*Rec}
  = either nil (\(x,y) -> fflatten_{Par} x ++ fflatten_{Rec} y)
  = either nil (\(x,y) -> id x ++ id y)
  = either nil (uncurry (++))
```

3.3 Function `topEq`

Function `topEq :: TopEq t => t -> t -> Bool` compares the top level of two terms for equality. It is defined in terms of the polytypic equality functions `fequal` and `pequal` (see figure 8). The first argument to `fequal` compares parameters for equality, the second argument (that compares the subterms) is constantly true (to get top level equality) and the third and fourth arguments are the two (unfolded) terms to be compared:

```
instance (Regular d, Eq a) => TopEq (d a) where
  topEq t t' = fequal (==) (\_ _ -> True) (out t) (out t')
```

```
polytypic fequal_f :: (a->b->Bool) -> (c->d->Bool) -> f a c -> f b d -> Bool
    = \p r -> case f of
                g + h     ->   sumequal  (fequal_g p r) (fequal_h p r)
                g * h     ->   prodequal (fequal_g p r) (fequal_h p r)
                Empty     ->   \_ _ -> True
                Par       ->   p
                Rec       ->   r
                d @ g     ->   pequal_d (fequal_g p r)
                Const t   ->   (==)

pequal :: (a->b->Bool) -> d a -> d b -> Bool
pequal eq x y = fequal eq (pequal eq) (out x) (out y)

sumequal :: (a->b->Bool) -> (c->d->Bool) -> Either a c -> Either b d ->Bool
sumequal f g (Left  x) (Left  v) = f x v
sumequal f g (Right y) (Right w) = g y w
sumequal f g _         _         = False

prodequal :: (a->b->Bool) -> (c->d->Bool) -> (a,c) -> (b,d) -> Bool
prodequal f g (x,y) (v,w) = f x v && g y w
```

Fig. 8. The polytypic `fequal` function.

The polytypic function `pequal` is useful in its own right as we can use it to define Haskell's "derived" equality function for all regular datatypes:

```
instance (Regular d, Eq a) => Eq (d a) where
  x == y  =  pequal (==) x y
```

3.4 ***Function*** `varCheck`

Function `varCheck :: VarCheck t => t -> Maybe Var` checks whether or not a term is a variable. A polytypic `varCheck` must recognise the datatype constructor that represents variables, using only information about the structure of the datatype. We have for simplicity chosen to represent variables by the first constructor in the datatype, which should have one parameter of type `Var`.

```
instance Regular d => VarCheck (d a) where
  varCheck = fvarCheck . out

polytypic fvarCheck :: f a b -> Maybe Var
  = case f of
      g + h      ->  either fvarCheck' err
      Const Int  ->  ok
      g          ->  err
```

3.5 Summary

We have now made all regular datatypes instances of the class `Term`. Thus, by combining the unification algorithm from section 2 with the polytypic instance declarations from this section, we obtain a unification algorithm that works for all regular term types.

Acknowledgements

We are grateful to Richard Bird, Doaitse Swierstra, Patrik Berglund and Oege de Moor for their comments on previous versions of this paper.

References

Bird, Richard, de Moor, Oege, & Hoogendijk, Paul. (1996). Generic functional programming with types and relations. *Journal of Functional Programming*, **6**(1), 1–28.

Jansson, P., & Jeuring, J. (1997). PolyP - a polytypic programming language extension. *Pages 470–482 of: POPL'97*. ACM Press.

Jeuring, J. (1995). Polytypic pattern matching. *Pages 238–248 of: FPCA'95*. ACM Press.

Jeuring, J., & Jansson, P. (1996). Polytypic programming. *Pages 68–114 of: AFP'96*. LNCS, vol. 1129. Springer-Verlag.

Knight, K. (1989). Unification: A multidisciplinary survey. *Computing surveys*, **21**(1), 93–124.

de Moor, O. (1994). Categories, relations and dynamic programming. *Mathematical Structures in Computer Science*, **4**, 33–69.

Pierce, B.C. (1991). *Basic category theory for computer scientists*. Foundations of Computing. The MIT Press.

Robinson, J.A. (1965). A machine-oriented logic based on the resolution principle. *Journal of the ACM*, **12**, 23–41.