Generic Unification via Two-Level Types and Parameterized Modules

[Functional Pearl]

Tim Sheard
Pacific Software Research Center
Oregon Graduate Institute
sheard@cse.ogi.edu

ABSTRACT

As a functional pearl, we describe an efficient, modularized implementation of unification using the state of mutable reference cells to encode substitutions. We abstract our algorithms along two dimensions, first abstracting away from the structure of the terms to be unified, and second over the monad in which the mutable state is encapsulated.

We choose this example to illustrate two important techniques that we believe many functional programmers would find useful. The first of these is the definition of recursive `data` types using two levels: a structure defining level, and a recursive knot-tying level. The second is the use of rank-2 polymorphism inside Haskell's record types to implement a form of type parameterized modules.

Keywords

Generic programs, unification, parameterized modules

1. INTRODUCTION

This pearl describes the modularization of a whole class of algorithms that compare two instances of the same data structure. This class contains algorithms for *unification*, *matching*, and *equality* amongst others. Efficient implementations of many of these algorithms rely on mutable state. We show how to modularize these algorithms using Haskell. Our modularization is performed along two dimensions, abstracting over the monad in which the mutable state resides, and the structure of the terms being compared.

The pearl demonstrates two different techniques of interest to serious functional programmers: generic programming in standard Haskell (without using any extensions), and type-parameterized modules (using higher-order kinds, and the rank-2 polymorphism extension). Generic programming is the construction of a single algorithm that works over multiple data structures. This allows the programmer to write an algorithm once and to reuse it at many different types. A type-parameterized module is a collection of (possibly polymorphic) algorithms or functions parameterized by a set of types. This allows a single module to be reused, simply by instantiating it at different types.

The work reported here has been influenced strongly by two papers. *Basic Polymorphic Type Checking*[1] by Luca Cardelli, and *Using Parameterized Signatures to Express Modular Structure*[10] by Mark Jones.

Luca Cardelli's paper describes how to implement Hindley-Milner type inference for an ML-like language. The algorithm uses destructive update to achieve an efficient implementation of unification over terms representing types. This kind of unification algorithm is extremely versatile and useful. The author of this paper has based literally dozens of other implementations on it, unifying datatypes representing many different kinds of terms.

Mark Jones' papers describes how to implement a module system by using parameterized signatures. The idea is to define functor-like operators, as defined by ML's module system[2], that use parameterization over types rather than sharing constraints to express sharing[3]. The recent addition of rank-2 polymorphism to the Hugs Haskell interpreter, and the GHC compiler, allows the encoding of modules as first class objects. Our experience with this encoding provides strong evidence that type-parameterized modules really work.

2. UNIFICATION

Unification of x and y is usually defined as finding a substitution σ such that $\sigma\, x = \sigma\, y$. The terms x and y may contain variables, and a substitution is a partial function from variables to terms (often represented as a list of pairs).

An efficient implementation of unification relies on representing variables as pointers to terms. A substitution in this case is represented by the global state of the pointers. If a variable points to null, it is said to be unbound. If the pointer is not null, then the variable is bound to the term it points to. This can be implemented in Haskell by using the state monad (`ST`)[12]. In what follows we assume the reader has a rudimentary knowledge of Haskell, unification and the `ST` monad. Some additional features of Haskell, perhaps unknown to some readers can be found in Figure 2.

In Figure 1 we give a basic transcription of Cardelli's unification algorithm into Haskell. The goal of this paper is to abstract details from this implementation, and to modularize it into several orthogonal components. Grasping the details of this concrete instance of the algorithm, will make

ICFP'01, September 3-5, 2001, Florence, Italy.

```
type Ptr a = STRef a (Maybe (TypeExp a))

data TypeExp a
 = MutVar (Ptr a)
 | GenVar Int
 | OperType String [ TypeExp a ]


prune :: TypeExp a -> ST a (TypeExp a)
prune t =
 case t of
   MutVar r ->
     do { m <- readSTRef r
        ; case m of
            Nothing -> return t
            Just t2 ->
              do { t' <- prune t2
                 ; writeSTRef r (Just t')
                 ; return t'}}
   other -> return t


occursInType :: Ptr a -> TypeExp a -> ST a Bool
occursInType r t =
  do { t' <- prune t
     ; case t' of
         MutVar r2 -> return(r==r2)
         GenVar n -> return False
         OperType nm ts ->
            do { bs <- mapM (occursInType r) ts
               ; return(or bs)
               }
     }
```

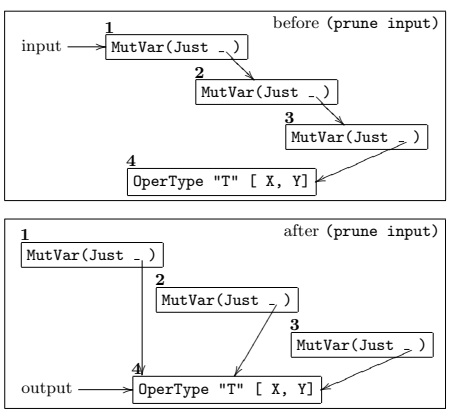
```
unifyType :: TypeExp a -> TypeExp a -> ST a ()
unifyType t1 t2 =
  do { t1' <- prune t1
     ; t2' <- prune t2
     ; case (t1',t2') of
        (MutVar r1, MutVar r2) ->
          if r1==r2
             then return ()
             else writeSTRef r1 (Just t2')
        (MutVar r1, _) ->
          do { b <- occursInType r1 t2'
             ; if b then error "occurs in"
                    else writeSTRef r1 (Just t2') }
        (_,MutVar _) -> unifyType t2' t1'
        (GenVar n,GenVar m) ->
          if n==m then return() else error "different genvars"
        (OperType n1 ts1,OperType n2 ts2) ->
          if n1==n2
             then unifyArgs ts1 ts2
             else error "different constructors"
        (_,_) -> error "different types"
     }
 where unifyArgs (x:xs) (y:ys) =
         do { unifyType x y; unifyArgs xs ys }
       unifyArgs [] [] = return ()
       unifyArgs _ _ = error "different lengths"

instantiate :: [TypeExp a] -> TypeExp a -> TypeExp a
instantiate ts x =
  case x of
    MutVar _ -> x
    OperType nm xs -> OperType nm (map (instantiate ts) xs)
    GenVar n -> ts !! n
```

Figure 1: Basic unification modelled after the algorithm described in Luca Cardelli's *Basic Polymorphic Type Checking*. The algorithm is transcribed from Modula-2 into Haskell.

it easier to understand the abstract version we will produce. We enumerate the important parts of Figure 1.

- In the `ST` monad, every function that accesses a mutable variable (to either read, write, or create a new one) has a range of type `(ST a x)`. We say that such functions have a monadic type. Each computation that access a mutable variable takes place in some thread. It is instructive to think of the type variable `a` in the type signatures of the state mutating functions as representing this thread. Thus functions whose signature contains several parameters with the same type variable `a`, must manipulate these objects in the same thread.
- A `TypeExp` is either a variable (`MutVar`), a generic type (`GenVar`, which is used for template variables, see the last bullet of this enumeration), or a type constructor applied to a list of types (`OperType`). For example `Int` is represented by `(OperType "Int" [])` and `[ x ]` by `(OperType "[]" [ x ])`.
- A pointer to a `(TypeExp a)` is a reference cell in the ST monad that holds an object of type `Maybe(TypeExp a)`. `Nothing` represents the null pointer, and `Just x` represents a pointer to `x`.
- Objects of type `(TypeExp a)` often contain long chains of `MutVar`'s pointing to other `MutVar`'s. The function `prune` follows such a chain, side-effecting each of the pointers in the chain to point to the element at the bottom of the chain. The value returned by `prune` is this last element. This is sometimes called path compression, and is illustrated below.



```
-- Monads in general
mapM :: Monad a => (b -> a c) -> [b] -> a [c]
mapM_ :: Monad a => (b -> a c) -> [b] -> a ()

-- References in the ST monad
readSTRef :: STRef a b -> ST a b
writeSTRef :: STRef a b -> b -> ST a ()
newSTRef :: a -> ST b (STRef b a)

-- Haskell's Record Syntax
-- definition
data R = R { one :: Int, two :: Bool }
-- field selection
one(R {one = 5, two = False})  --> 5
-- partial construction
R { one = 5 } --> R {one =5, two = undefined}
-- field updating
x { one = 4 }  --> R{one = 4, two = two x}
```

Figure 2: Types of Haskell library functions dealing with monads in general, references and the ST monad, and uses of Haskell record syntax.

- The function `occursInType` determines if a pointer appears somewhere inside another `TypeExp`. Note that pointers may be changed by the call to `prune`. This is why `occursInType` has a monadic type, indicating that it might mutate state.

- The function `unifyType` first calls `prune` on its two arguments to eliminate any chains of `MutVar`'s. The resulting objects (`t1'` and `t2'`) may still be `MutVar`'s, but if they are, then they are guaranteed to be null pointers (i.e. be references to `Nothing`). A simultaneous case analysis of the two resulting objects succeeds if both have the same top-level constructor, or if at least one is a variable.

  If they are both the same variable, then there is nothing to do. If two variables are matched, but are different variables, make the first variable point to the second. This is how chains of variables are created. If one is a variable, and the other is not, check if the variable occurs in the other. If it does, this is an error. If not, then make the variable point at the other object.

  If they both have the same top-level constructor then recursively analyze the substructures. This is the purpose of the local function `unifyArgs`.

- The function `instantiate` behaves like a substitution function, replacing every `(GenVar n)` with the `n`'th element of the substitution list `ts`. The function `instantiate` is the generic interface to templates. A template is a `GT` object containing no `MutVar`s. Templates are used when one wants to unify one `GT` term with many other `GT` terms. Once a `MutVar` becomes bound, it is difficult to unbind it without jeopardizing the correctness of the unification algorithm. If it is necessary to unify a single term many times, then a template structure must be used. A template structure contains template variables (`GenVar`) in place of `MutVar` variables. Each time the template is to be used, it is instantiated by making a copy of the template, replacing each template-variable with *fresh* type variables. The `instantiate` function provides this capability. The technique of using `Int` to represent template variables, and a list of `TypeExp` as their instantiations derives from the paper *Typing Haskell in Haskell*[11].

Unification using the state of updatable reference cells to encode substitutions is hard to get right. On reason is the algorithmic details of chasing and overwriting pointers. Pointer chasing is necessary to ensure the invariant that a pointer, and and the thing it points to, are in some sense semantically equivalent. Every function that manipulates `TypeExp`s must maintain this invariant. One way to accomplish this is to use a "copying" algorithm that removes these excess pointers as other computations are performed. Overwriting a minimal number of pointers ensures that the time behavior of the algorithm remains tractable, and adds to the complexity. Because overwriting is a stateful operation, everything must be in some appropriate monad.

As the datatype representing terms becomes more complex, the control structure of the algorithm becomes more sophisticated. Terms with complex sub-structure require sophisticated control to make recursive calls on subterms properly, and to combine the results of the recursive calls to build a suitable overall result.

It would be nice to separate the pointer issues from the control issues; get each one done right, and then combine them. The pointer chasing could be abstracted over different monads, and reused with multiple data structures representing different kinds of terms. The control structure could be reused when defining additional algorithms over terms other than unification. After all *matching* (where only one of the terms can have variables), and *equality testing* have control structure remarkably similar to unification.

In the rest of this paper we explain how this can be done in Haskell.

3. MODULAR DATA

Separating the pointer algorithms from the algorithms that deal with the structure of terms, requires splitting the datatype `TypeExp` into several datatypes. This separates the *structure of terms* and the *use of variables* into two different datatypes. All kinds of terms which support unification will have variables, but their structure will vary according to what the terms are meant to represent. We separate terms into two levels. The first level incorporates the two kinds of variables encoded in the constructors `MutVar` and `GenVar`. The second level incorporates the role played by the constructor `OperType`. It abstracts the structure of all the other constructor functions of terms. The *recursive* structure of terms will be split between the two levels.

We will make this more clear by applying our technique to the same kinds of terms we used in Figure 1. We split the definition of `TypeExp` into the two components `S` (for the *Structure* of terms) and `GT` (for *Generic Term*).

```
-- Structure operator, hence "S"
data S x = OperType String [x]

-- Generic Terms, hence "GT"
-- "s" abstracts over structure, "r" over references
data GT s r
```

```
  = S (s (GT s r))
  | MutVar (r (Maybe (GT s r)))
  | GenVar Int

type TypeExp a = GT S (STRef a)
```

We call `S` the structure operator because it captures the structure of the terms we are manipulating. Note how it is parameterized by `x` which appears in places where recursive calls to `TypeExp` were placed in the original definition. This is the first half of capturing the recursive structure of terms. The structure of `TypeExp` is quite simple, so `S` only has one constructor (`OperType`), but we will soon see examples where the structure of terms is much richer.

The `GT` datatype incorporates the role of variables in terms: template variables (`GenVar`), and normal variables (`MutVar`). It abstracts over the rest of the structure of terms using the parameter `s`. It is in the type of the `S` constructor function that `GT` captures the second half of the recursive structure of `TypeExp`. Note how a recursive call `(GT s r)` is passed to the parameterized structure operator `s`.

It is important to note that the parameter `s` to `GT` is a type constructor, not a type. The parameter `r` is also a type constructor. It abstracts over the type constructor constructing mutable references. The type constructor `GT` is recursive in both the `MutVar` and `S` constructor functions, forwarding the recursion through the type constructors `s` and `r`.

The new version of `TypeExp` is an instantiation of `GT`, choosing for its two parameters the structure operator `S`, and the `STRef` type constructor for references from the `ST` monad.

Values of type `TypeExp` are constructed in two levels. An outer level consisting of one of the constructors of `GT`: `S`, `MutVar`, or `GenVar`; and in the case of the constructor function `S` an inner level consisting of the constructor function of the type constructor `S`: (`OperType`) (in general the type constructor `S` may have many constructor functions). We illustrate this in the table below.

New, two level examples	Old, one level examples
`GenVar 4`	`GenVar 4`
`S(OperType "Bool" [])`	`OperType "Bool" []`
`S(OperType "[]"` `  [S(OperType "Int" [])])`	`OperType "[]"` `  [OperType "Int" []]`

This pattern of prefixing every constructor of `S` with `S` to construct a `TypeExp` is captured by employing a convention that defines a new function for every constructor of `S`. These functions have the same name as the constructor, except their initial upper case letter is made lower case.

```
operType s ts = S(OperType s ts)
```

The benefit of employing this convention is that the programmer need not continually employ the type constructor `S` every time a term is constructed.

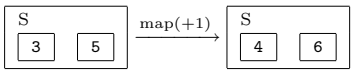
3.1 Abstract Operations

How are functions over objects of type `(GT s r)` written if `s` and `r` are unknown? One way to do this is to assume that `s` and `r` are instances of some special classes that enumerate their general operations. For `s` we have found the following class useful.

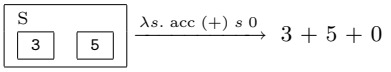
```
class Sclass s where
  mapS   :: (x -> y) -> s x -> s y
  accS   :: (x -> y -> y) -> s x -> y -> y
  seqS   :: Monad m => s(m x) -> m(s x)
  matchS :: s x -> s x -> Maybe[(x,x)]\
```

One way to understand these functions is to imagine objects of type `(s x)` as "boxes" labeled `S`, with compartments of type `x`.

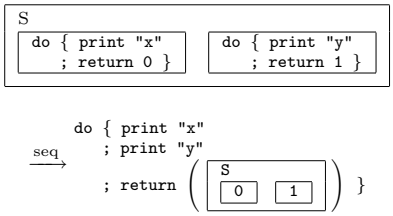
- The operation `mapS` applies a function of type `(x -> y)` to each compartment, producing a box labeled `s` with compartments of type `y`.



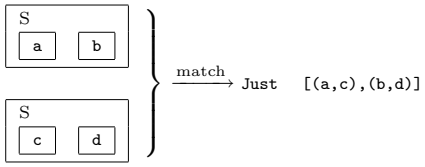
- The operation `accS` accumulates a "sum" by repeatedly applying a binary "addition" function to each of the compartments and the previous "subtotal".



- The operation `seqS` produces a single `m`-effecting computation that produces an object of type `s x` from a box labeled `s` where the compartments are each filled by smaller `m`-effecting computations, each of which produces an object of type `x`. It does this by ordering all the sub-computations into one large computation.



- Finally, `matchS` compares the top level constructors of two `s` boxes. If the constructors match (then both boxes contain the same kind of "compartments") it returns `Just` applied to a list of pairs, pairing corresponding compartments from each box. If the constructors do not match then it returns `Nothing`.



When `s` is the `S` datatype from our `TypeExp` example, we can use the following instance.

```
instance Sclass S where
  mapS f (OperType s xs) = OperType s (map f xs)
  accS acc (OperType s xs) ans = foldr acc ans xs
  seqS (OperType s xs) =
     do {xs' <- order xs; return(OperType s xs')}
    where order [] = return []
          order (x:xs) = do { x' <- x
                            ; xs' <- order xs
                            ; return (x':xs') }
  matchS (OperType s xs) (OperType t ys) =
    if s==t && (length xs)==(length ys)
       then Just(zip xs  ys)
       else Nothing
```

3.2 Using the abstract operations

It is now possible to write the `instantiate` function without knowing the structure of `s` at all.

```
instantiate ::
  Sclass s => [GT s r] -> GT s r -> GT s r
instantiate ts x =
  case x of
    S y -> S(mapS (instantiate ts) y)
    MutVar _ -> x
    GenVar n -> ts !! n
```

In the first clause of the case expression, the variable `y` has type `(S (TypeExp a))`. The `mapS` function applies recursive calls of `instantiate` to each of the compartments, obtaining another object of type `(S (TypeExp a))` that is wrapped by `S` into the final answer with type `(TypeExp a)`. It is possible to write many functions over `(GT s r)` without actually knowing the exact structure of its `s` substructure. The type of `instantiate`, `(Sclass s => [GT s r] -> GT s r -> GT s r)`, makes precise the requirement that not any structure operator will do. Only those `s` that are a member of the class `Sclass`.

To write the functions `occursInType` and `unifyType` additional structure must be known about the type constructor `r`. This can be captured using two additional classes, that abstract over the operations on references.

```
class EqR r where
  sameVarR :: r x -> r x -> Bool

class Monad m => Rclass r m where
  writeVarR :: r x -> x -> m()
  readVarR  :: r x -> m x
  newVarR   ::  x -> m(r x)
```

The class `EqR` captures that references must be comparable for equality. The class `Rclass` aggregates the operations that create, read, and write references into a single class with a common multiparameter constraint. `Rclass` is multiparameter, because it relates two type constructors, `r` the reference type constructor, and `m` the monad in which it operates.

The function `prune` over `GT s r` objects can be defined in exactly the same manner as it was over `TypeExp` objects. The definition found in Figure 1 is identical to the new definition, so we omit it here. This is possible because `prune` returns a default action on all non-`MutVar` objects, so the changes in the structure of terms is not noticed. Of course it has a different type:

```
prune :: (Rclass r m) => GT s r -> m (GT s r)
```

With these classes in place the function `occursInType` can be written in a generic manner.

```
occursInType :: (Rclass r m, Sclass s, EqR r) =>
          r (Maybe (GT s r)) -> GT s r -> m Bool
occursInType r t =
  do { t' <- prune t
     ; case t' of
         MutVar r2 -> return(sameVarR r r2)
         GenVar n -> return False
         S x ->
           do { bs <- seqS(mapS (occursInType r) x)
              ; return(accS (||) bs False) } }
```

To determine if reference `r` occurs in term `t` prune away any leading chain of `MutVar`'s. If the result of the prune is `MutVar r2` then `r` occurs in `t` only if `r` and `r2` are the same reference. If the result is an `S` structure, `x`, then use the generic operators. First apply `occursInType` to all the compartments in `x`. The operation `mapS` performs this task. This produces an `S` structure filled with monadic computations, each of which returns a `Bool`. Applying the `seqS` operation to the `S` structure produces a larger computation returning an `S` structure filled with booleans. This structure is bound to `bs`. All the `Bool`s inside `bs`'s compartments can then be logically or-ed together (using `accS`) to produce the final result.

The function `unifyType` can be made general in a similar manner. Replace each call to reference equality with one to `sameVarR`, `readSTRef` with `readVarR`, `writeSTRef` with `writeVarR`, and `newSTRef` with `newVarR`. Finally, replace the case clause that compares two `OperType` constructors with some generic code written in terms of the operations of the `Sclass` class. The important difference are summarized here:

```
unifyType :: (Rclass r m, Sclass s, EqR r) =>
             (GT s r, GT s r) -> m ()
unifyType (t1,t2) =
  do { t1' <- prune t1
     ; t2' <- prune t2
     ; case (t1',t2') of
       . . .
        (S x, S y) ->
          case matchS x y of
            Nothing -> error "different constructors"
            Just pairs -> mapM_ unifyType pairs
       . . .
     }
```

If we are unifying two `S` structures, determine if their top level constructors are the same using `matchS`. If they don't match (`Nothing` is returned) then unification fails. If they do match, then `pairs` is a list of corresponding subcomponents. Recursively, unify each pair and succeed only if all are successful. The function `mapM_ :: Monad m => (b -> m c) -> [b] -> m ()` is part of the standard monad library and performs exactly the task necessary.

3.3 Comments about modularity and classes

The solution above makes heavy use of classes. So heavy, in fact, that there are serious questions about its scalability. The type of `unifyType` is cluttered with three class constraints: `(Rclass r m, Sclass s, EqR r) => (GT s r, GT`

`s r) -> m ()`. And, this type doesn't explicitly mention the implicit constraint `Monad m` that is implied by `Rclass r m`. When a function's type is prefixed by too many class constraints it becomes hard for the programmer to grasp the full meaning of the type.

Having many class constraints also has implications for program maintainability. As a means of documentation, it is standard practice for Haskell programmers to annotate functions with their types, even though their types could be inferred. Such annotations become hard to maintain if the number of class constraints gets large.

We have built richer implementations than the one described here. These implementations introduce recoverable errors, add a return type for `unifyType` other than `()`, and add a notion of generalization (a way of creating templates from terms with unbound variables). Each of these extensions adds an additional class or two. The types of our functions become so constrained with these additional classes that they become unreadable. The types also become so general that type inference often fails to assign a unique type to each function, requiring explicit typing annotations.

What is needed is one big "class-like" thing that abstracts over all the types `s`, `r` and `m`, and all their operations at once. Something like:

```
aBetterKindOfClass RSclass r m s where
  sameVarR  :: r x -> r x -> Bool
  writeVarR :: r x -> x -> m()
  readVarR  :: r x -> m x
  newVarR   :: x -> m(r x)
  seqS      :: s(m x) -> m(s x)
  mapS      :: (x -> y) -> s x -> s y
  accS      :: (x -> y -> y) -> s x -> y -> y
  matchS    :: s x -> s x -> Maybe[(x,x)]
```

Unfortunately, in Haskell such a class is impossible to define. The desire for type inference requires that every method in every class mention all of the abstracted variables; in this example, all of `r`, `m`, and `s`, which is just not the case. The recent suggestion[9] of using functional dependencies to partially alleviate this restriction is still not strong enough for the example above.

Doing without class constraint inference, and requiring explicit type annotations at every overloaded function, could alleviate this problem. But class constraint inference is so useful in other contexts, that this is hardly a credible solution. What is needed is something along the lines suggested by Mark Jones in his paper *Using Parameterized Signatures to Express Modular Structure*[10].

Fortunately, we need not wait for a new version of Haskell with such a module system built in. We can build this functionality ourselves using an existing extension to Haskell: rank-2 polymorphism.

Without rank-2 polymorphism, universal quantification must be completely "outside" all other types. For example in the type of `(++) :: forall a . [a] -> [a] -> [a]`, the `forall` quantifies the whole type. With rank-2 polymorphism we can place the quantifier "inside" other type constructors. For example: `runST :: forall a . (forall b. ST b a) -> a`. Here the outermost quantifier `(forall a)`, quantifies the whole expression, but the `(forall b)` quantifies only the back end of the arrow type. Rank-2 polymorphism admits functions, like `runST`, that require (and make use of) polymorphic parameters. Of course the type of such functions must be explicitly declared by the programmer[13], as they cannot be inferred. These type declarations are necessary since type inference of rank-2 polymorphic types is in general undecidable.

This extension also applies to constructor functions inside of `data` declarations. They can be given polymorphic components. Using this technique we can express the structure we require.

```
data RSclass s r m = RSclass
  { sameVarRS  :: forall x. r x -> r x -> Bool
  , writeVarRS :: forall x. r x -> x -> m()
  , readVarRS  :: forall x. r x -> m x
  , newVarRS   :: forall x. x -> m(r x)
  , seqRS :: forall x. s(m x) -> m(s x)
  , mapRS :: forall x y. (x -> y) -> s x -> s y
  , accRS::forall x y. (x -> y -> y) -> s x -> y -> y
  , matchRS :: forall x. s x -> s x -> Maybe[(x,x)]
  , errorRS :: forall x . String -> m x
  }
```

The `RSclass` datatype definition plays the role of a type-parameterized signature. It specifies an aggregation of (possibly polymorphic) functions parameterized by a set of types. An object of type `RSclass` plays the role of a module. Every such object will specialize the type parameters to some concrete types. A function from `RSclass` to some other type-parameterized signature plays the role of an ML-style functor.

In `RSclass` we have added an `errorRS` field to the operations we have previously discussed. In Figure 1 the use of Haskell's `error` function makes aborting the program the only possible response to errors. The `errorRS` field allows the possibility that generic operations, such as `unifyType` can handle errors in some more graceful manner.

Under this scheme, an instance declaration corresponds to a value of type `RSclass`, instantiated at real types for `s`, `r`, and `m`.

```
rs :: RSclass S (STRef a) (ST a)
rs = RSclass
  { sameVarRS = (==) :: STRef a b -> STRef a b -> Bool
  , writeVarRS = writeSTRef
  , readVarRS = readSTRef
  , newVarRS = newSTRef
  , seqRS = seqS
  , mapRS = mapS
  , accRS = accS
  , matchRS = matchS
  , errorRS = error
  }
```

Here the "`STRef`" functions are the library functions whose types were given in Figure 2, and `seqS`, `mapS`, `accS`, and `matchS` are defined exactly as they were in the `Sclass` instance for `S` in Section 3.1. We have instantiated `errorRS` as `error` but other "instances" could use a different error function.

The collection of functions we wish to generate in a generic manner can also be aggregated into a `data` structure.

```
data GTstruct s r m =
  B { unifyGT :: GT s r -> GT s r -> m ()
    , occursGT :: Ptr s r -> GT s r -> m Bool
    , instanGT :: [GT s r] -> GT s r -> m(GT s r)
    , pruneGT :: GT s r -> m(GT s r)
    }
```

Here the record fields of the `GTstruct` are populated with generic versions of the functions `UnifyType`, `OccursInType`, `instantiate`, and `prune`.

The aggregate `GTstruct` structure can be generated by a function with type `Monad m => RSclass s r m -> GTstruct s r m`. This function can be written once, and applied to different `RSclass` objects to generate unification structures for many different term types. It can also be instantiated on different monads. We call such a function `makeUnify`, and a skeleton for it is given below:

```
makeUnify ::
   Monad m => RSclass s r m -> GTstruct s r m
makeUnify lib =
  B { freshGT = freshVar
    . . .
    }
  where freshVar = do { r <- newVarRS lib Nothing
                      ; return (MutVar r) }
    . . .
```

Note the explicit parameter `lib` of type `RSclass` to `makeUnify`, and the explicit application of the field selector `newVarRS` to `lib`. Using the class system, rather than encoding our own type parameterized modules, allows this parameterization and selection to be implicit, but allows only a single instance at each type, and admits all the disadvantages enumerated in Section 3.3. In Appendix A we have included the the complete code for `makeUnify`. In the appendix we have also enriched the `GTstruct` to include operations for matching, equality, and several other generic operations as well.

4. A RICHER MONAD

In this section we illustrate how a different monad can be used to instantiate the `RSclass` structure. A obvious choice might be to use the `IO` monad, since it too supports references. Instead we show how to extend the monad in a different direction.

Many programs must recover from failed unification. We can accommodate this in several ways. One way, that we do not illustrate here, is to design generic unification algorithms with type: `term -> term -> M ans`, where `ans` is some type other than `()`. Thus unification can return an interesting result, that indicates what happened. Types like `Bool` , `[ Failure ]`, or `Maybe ErrorMessage` spring to mind. We have done this, and it requires adding an `answer` type to the type parameters of `RSclass` and adding several new operations to the class, as well as making slight modifications to `makeUnify`.

A more interesting way to accommodate this is to design a richer monad that models failure, and to instantiate `makeUnify` with an instance that uses this richer monad instead of the `ST a` monad.

```
data Error = E String
newtype EM a x = EM (ST a (Either Error x))

instance Monad (EM a) where
  return x = EM(return(Right x))
  (>>=) (EM z) g =
    EM(do { c <- z
          ; case c of
              Left e -> return(Left e)
              Right x -> let EM w = g x in w
          })

runEM :: (forall b. EM b a) -> Either Error a
runEM x = let EM z = x in runST z
```

The `(EM a)` monad is a simple extension to the `(ST a)` monad. The possibility of failure is accommodated by returning `Either` an error (`Left x`) or a normal answer (`Right x`). The bind `(>>=)` operator of the monad propagates failure in its first argument without evaluating its second argument. We lift the `runST` operation to the `EM` monad with the function `runEM`. We use the `newtype` declaration, rather than the `data` declaration, when defining `EM` to avoid the extra level of indirection that an actual constructor function `EM` would introduce. When using a `newtype` the constructor function is "virtual" and is only used to supply type inference annotations.

We add two new operations to the `(EM a)` monad, `raise`, and `handle`. The operation `raise` introduces an error, and `handle` catches an error, and then begins a new computation.

```
raise :: String -> EM a x
raise s = EM(return(Left (E s)))

handle :: (EM a x) -> (EM a x) -> (EM a x)
handle (EM x) (EM y) =
  EM (do { x' <- x
         ; case x' of
            Left _ -> y
            Right x -> return(Right x)
         })
```

4.1 Using the richer monad

The `RSclass` aggregates nine separate operations. We can define a partially defined `RSclass` object, where only the 5 reference and monad operations have been filled in. The laziness of Haskell makes this possible. Later in the program source we can construct many other instances of `RSclass` objects, that fill in the holes with the missing operations.

```
readVar r =
  EM(do { a <- readSTRef r; return(Right a) })
writeVar r x =
  EM(do { a <- writeSTRef r x; return(Right ()) })
newVar x =
  EM(do { r <- newSTRef x; return(Right r) })

emPartial :: RSclass s (STRef a) (EM a)
emPartial = RSclass
  { sameVarRS = (==)::STRef a b -> STRef a b -> Bool
  , writeVarRS = writeVar
  , readVarRS = readVar
  , newVarRS = newVar
  , errorRS = raise
  }
```

Note that `emPartial` is still polymorphic in its `s` parameter. We can use the field updating syntax of Haskell to make more complete copies of `emPartial` at many different instantiations of `s`. We do this in the definition of `commandClass` in Section 5 below.

5. A RICHER TERM-STRUCTURE.

In this section we illustrate instantiating our general framework on a type for representing terms. We consider a type of terms representing commands in a simple imperative language. We build the `S` structure operators for this type, instantiate our general framework with the `(EM a)` monad of the previous section and illustrate the use of the framework to build a simple transformation system over commands.

```
data C x
  = If Exp x x       --  if e then s2 else s1
  | While Exp x      --  while e do s
  | Begin x x        --  { s1 ; s2 }
  | Skip             --  {}
  | Assign Var Exp   --  x := e

-- the lowercase constructor convention

ifc e x y = S(If e x y)
while e x = S(While e x)
begin x y = S(Begin x y)
skip = S Skip
assign v e = S(Assign v e)

type Command a = GT C (STRef a)
```

In Appendix B we give the definitions of the `S` structure operators `seqC`, `mapC`, `accC`, and `matchC`. These are easy to define, and in fact, could be generated automatically given the right tools[4, 5]. To build an `RSclass` instance using the `(EM a)` monad reference operators, we simply use the record update syntax on the partially defined `RSclass` value `emPartial` defined in the previous section.

```
commandClass :: RSclass C (STRef a) (EM a)
commandClass =
  emPartial { seqRS = seqS
            , mapRS = mapS
            , accRS = accS
            , matchRS = matchS
            }
```

Our transformation system will implement simple rewrites over terms. For example, rewrites like `(if True then x else y)` $\longrightarrow$ `x`. To build the machinery needed, generic versions of matching, instantiation, and the creation of fresh variables will be needed. At this point we can conjure up working examples of these and several other generic operators, simply by using the generic `makeUnify` function from Appendix A.

```
B{ matchGT = match
 , instanGT = instan
 , freshGT = fresh
 , tofixGT = toFix
 , fromfixGT = fromFix
 } = makeUnify commandClass
```

Two things to note here. First, when using a record pattern on the left-hand-side of a binding, it is the variables to the right of the equals (`=`) that are being defined. In this example, those variables being bound are `match`, `instan`, `fresh`, etc. Second, several of the field names in this example are derived from the definition in Appendix A, rather than the one in the running text.

A transformation system is defined by rewrite rules, and an engine that applies those rules. A simple system for the command language can be built succinctly using our framework. First, define the structure of rules:

```
data Rule a = R Int (Command a) (Command a)
```

A rule is an `Int` and a pair of `(Command a)` objects. The `Int` represents the number of template variables in the rule, and the pair of `(Command a)` objects represent the left- and right-hand sides of the rewrite rule it encodes. We use this representation to simplify the implementation. The `(Command a)` objects have template variables where the rule may match any sub-command. We give some example rules below.

```
(w,x,y,z) =(GenVar 0,GenVar 1,GenVar 2,GenVar 3)

r1,r2,r3,r4 :: Rule a
r1 = R 3 (begin w (begin x y)) (begin (begin w x) y)
r2 = R 2 (ifc True w x) w
r3 = R 2 (ifc False w x) x
r4 = R 1 (while False x) skip
```

To apply a rule we instantiate the template variables with real variables, apply the matching procedure to the left-hand side of the rule and the term being transformed. If the match succeeds, then the instantiated right-hand side contains the result of the match.

```
rewrite (R n lhs rhs) x =
  handle (do { freshvs <-
                 sequence(take n (repeat fresh))
             ; lhs' <- instan freshvs lhs
             ; match lhs' x
             ; instan freshvs rhs
             })
         (return x)
```

If the matching fails, then the whole `(do ... )` fails. The failure is captured by `handle`, and the original term is returned unchanged.

6. ESCAPING THE MONAD

Monads (`M`) with mutable references have the unfortunate problem that there is no simple way to transform a computation of type `(M x)` into a value of type `x`. This restriction is imposed because it ensures that no reference escapes its scope.

It first appears, that because generic types (`GT s r`) have references inside them, that once inside the monad there is no hope of ever escaping the monad to produce "pure" values. If the monad is built on top of the `IO` monad this will always be the case, but if the monad is built on top of the `ST` monad (like the `EM` monad) this need not be the case. Computations with type `(ST a x)` that are completely polymorphic in the thread type variable `a` can be converted into values of type `x` using the function `runST`.

Fortunately many terms are completely polymorphic in the state thread variable `a`. All top-level program constants

with type `(GT C (STRef a))`, and larger constants derived from them, are always polymorphic in the state thread, `a`. Witness the type of `r1 :: Rule a`.

An important role of the `GenVar` constructor, and the `instanGT` generic function is to allow the construction of templates. Templates, because they contain no `MutVar` constructors, are always completely polymorphic. Templates can be used to create non-polymorphic instances (with `MutVar` constructors) by the use of the pattern illustrated in the function `rewrite` in the previous section. Create a list of fresh type variables and instantiate the template using the function `instan`.

Polymorphic terms can also be constructed algorithmically, by a parser for example, if the algorithms never use the `MutVar` constructor. A term completely polymorphic in its thread variable can be extracted from the monad using `runST` in the `ST` monad or its equivalent (such as `runEM`) in other monads. In order to do this we need a form of terms that does not mention the thread variable.

If we know a generic term has no `MutVar` or `GenVar` constructors we can turn it into a type with similar structure. The `Fix` type constructor, like the `GT` type constructor, takes a type constructor as argument, but has a single constructor function, `Fix`. It plays the same role as the `S` constructor function.

```
newtype Fix s = Fix (s (Fix s))
```

Conversion between the two types can be made generic as well. In Appendix A, generic functions have been added to the `GTstruct` aggregate structure.

```
tofixGT :: GT s r -> m(Fix s)

fromfixGT :: Fix s -> GT s r
```

We can now illustrate a complete program.

```
try1 :: (forall a . GT C (STRef a))
           -> Either Error (Fix C)
try1 x = runEM(do { x1 <- rewrite r1 x; toFix x1})

transform :: Fix C -> IO()
transform x =
  let x' = fromFix x
  in case try1 x' of
      Left (E s) -> print $ "Fail: " ++s
      Right y -> print(show x++" =\n"++show y++"\n")
```

7. DISCUSSION

We have shown how to abstract the class of algorithms that includes unification, matching and equality, over both the structure of terms, and over the monad in which the computations occur. We used two techniques that we believe are generally useful and should be in the repertoire of every functional programmer.

The first is the use of two-level algebraic data-types, and utilization of generic operators such as `map`, `seq`, `acc`, and `match`. This allows the construction of generic algorithms in Haskell without the use of any language extensions. It also allows a brevity of expression in non-generic algorithms. We have not demonstrated such use in this paper for lack of space.

The second is the use of rank-2 polymorphism to encode user-defined parameterized modules. Such modules allow a level of abstraction not possible using Haskell's class system since they allow arbitrary "overlapping" instances. Such flexibility comes at the cost of a few explicit type annotations.

Our exploration of this new paradigm has not been without its setbacks. Using advanced features often pushes the limits of a paradigm or a language implementation. Three issues are worth discussion.

Mutual recursion. To generalize two level datatypes to mutually recursive structures one needs to parameterize each structure operator over all of the recursive components, both direct and indirect. We can illustrate this using a simple language for Haskell-like expressions and declarations. Below `E` is the structure operator for expressions, and `D` is the structure operator for declarations. Both `E` and `D` have parameters `e` and `d` which are used where recursive sub-components of type expression or declaration would normally be used.

```
type N = String          -- names
data E d e
  = Var N                -- x
  | Const Integer        -- 5
  | App e e              -- f x
  | Abs [N] e            -- \ x1 x2 -> e
  | Let [d] e            -- let x=e1 in e2
data D d e
  = Fun N [([N],e,[d])]  -- f p1 p2 = b where ds
  | Val N e [d]          -- p = e where ds

newtype Exp = E (E Decl Exp)
newtype Decl = D (D Decl Exp)
```

Since the structure operators (`E` and `D`) have more than one parameter, the generic operators `map` and `seq` must be generalized as well. For example:

```
mapD :: (a->b) -> (c->d) -> D a c -> D b d
seqD :: Monad m => D (m a) (m b) -> m(D a b)
```

Two level types are not restricted to lazy languages like Haskell. We have used them in ML as well. In fact we have found them to scale quite well to even very large, highly mutually-recursive datatype declarations. We use two level types to represent all the datatypes in the MetaML[1] implementation. We found them both easy to use, and advantageous in the genericity they supplied.

The following two problems were encountered using the Hugs interpreter in our initial research. Each has simple work-arounds. Further investigation as shown that neither occurs when using GHC.

Pattern matching polymorphic records. Using `data` types with polymorphic components as the input to functors (like `makeUnify`) worked well, but when we tried using pattern matching to bind the results of functor application, we stretched the limits of the Hugs implementation. For example consider the functor-like function `makeSeqmap` that takes a `(RSclass s r m)` as input and produces a `(T1 s m)` object as output:

[1]See `http://www.cse.ogi.edu/PacSoft/projects/metaml/index.html`

```
data T1 s m =
  T1 {seqmapGT :: forall x y .
        (x->m y) -> (s x) -> m(s y) }

makeSeqmap :: RSclass s r m -> T1 s m
makeSeqmap lib = T1 { seqmapGT = seqmap }
  where seqmap f x = seqRS lib(mapRS lib f x)
```

Everything works fine until we try and use the pattern matching feature of records to produce an actual instance of a `(T1 s m)` object with a component called `seqmap`. This top-level definition is not allowed:

```
T1 { seqmapGT = seqmap } = makeSeqmap rs
```

because of restrictions on the use of rank-2 polymorphism in pattern matching in Hugs. A work around for this is not to use pattern matching, but use the field selection mechanism instead.

```
lib = makeSeqmap rs
seqmap = seqmapGT lib
```

Update syntax of polymorphic records. In Section 5 we specialized the `emPartial` structure by updating its (undefined) fields for `seqRS`, `mapRS`, `accRS`, and `matchRS`. Unfortunately, in the Hugs interpreter, the record update syntax is not implemented for records that contain explicitly-typed polymorphic components. Fortunately there is a work around for this as well. We can replace the elegant:

```
commandClass :: RSclass C (STRef a) (EM a)
commandClass =
  emPartial { seqRS = seqS
            , mapRS = mapS
            , accRS = accS
            , matchRS = matchS
            }
```

with the equivalent, but much less elegant:

```
addSpart (RSclass sv wv rv nv e s m a mtch) =
  (RSclass sv wv rv nv e seqC mapC accC matchC)

commandClass :: RSclass C (STRef a) (EM a)
commandClass = addSpart emPartial
```

8. CONCLUSION

Despite these minor complications, the techniques discussed here show great promise in writing high-level, generic, reusable programs in Haskell (with minor extensions). The two techniques demonstrated here: two level syntax with generic operators, and type-parameterized `data` types with rank-2 polymorphic components, are both useful and complementary ideas that every functional programmer should be aware of.

The second idea could be made easier to use by incorporating it into a high-level module system based upon parameterized signatures as suggested by Mark Jones. Such a system should include the facility to define parameterized modules. They would serve the same purpose as the `makeUnify` function of this paper. A good design for parameterized modules would allow the names of components in the input module to be used directly, rather than relying on the selector function mechanism as was done in the definition of `makeUnify` in Appendix A. A well designed module system would also alleviate the last two problems discussed in the previous section.

9. HISTORY

The author was introduced to the use of two-level types by Erik Meijer in the fall of 1996, in a talk in which he used them to define catamorphisms, and other uniform control structures in Haskell.

```
data Fix f = Fix (f (Fix f))

fmap :: Functor f => (a->b) -> f a -> f b

cata :: Functor f => (f a -> a) -> Fix f -> a
cata phi (Fix x) = phi (fmap (cata phi) x)
```

This started a long period of experimentation with these ideas as a mechanism to write programs that did not depend upon the structure of the datatype they operated on. Ultimately, the use of uniform control structures, like `cata`, turned out to be too inflexible, but the generic operators, like `fmap` above, and `acc`, `match`, and `seq` turned out to be just the right stuff.

Most of these generic operators originate in the work of Johan Jeuring and his colleagues on *polytypic programming*[6, 8]. The one exception is `seq :: Monad m => s(m a) -> m(s a)`, which the author likes to believe he independently discovered (though it's probably been around for years). Generic unification is one of the prime examples[7] of polytypic programming, but the efficient algorithm employing mutable references has not been described.

The type parameterized modules idea originated from a discussion with Mark Jones about examples illustrating multiparameter type classes. Mark suggested abstracting the operations on stateful references away from the actual reference type constructor and monad type constructor.

```
class Monad m => Mutable r m where
  read  :: r x -> m x
  write :: r x -> x -> m ()
  new   :: x -> m(r x)
```

Frustration trying to retrofit some existing examples into this framework led to the techniques presented.

10. ACKNOWLEDGMENTS

The work described here was supported by NSF Grant CDA-9703218, the M.J. Murdock Charitable Trust and the Department of Defense.

The author would also like to thank Andy Gill, Andy Moran, Mark Jones, Emir Pasalic, and Simon Peyton Jones for discussions about these ideas, Frank Taylor for LaTex wizardry, and the entire *Advanced Functional Programming* class, at the Oregon Graduate Institute, in the winter of 2001, who suffered through endless versions of the implementation.

11. REFERENCES

[1] L. Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8(2):147–172, Apr. 1987.

[2] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In ACM, editor, *Conference record of POPL '94, 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: Portland, Oregon, January 17–21, 1994*,

pages 123–137, New York, NY, USA, 1994. ACM Press.

[3] R. Harper and B. C. Pierce. Advanced module systems (invited talk): a guide for the perplexed. *ACM SIGPLAN Notices*, 35(9):130–130, Sept. 2000.

[4] R. Hinze. Memo functions, polytypically! In J. Jeuring, editor, *Proceedings 2nd Workshop on Generic Programming, WGP'2000, Ponte de Lima, Portugal, 6 July 2000*, Tech. Report UU-CS-2000-19, Dept. of Computer Science, Utrecht Univ., pages 17–32. June 2000.

[5] R. Hinze. A new approach to generic functional programming. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POLP-00)*, pages 119–132, N.Y., Jan. 19–21 2000. ACM Press.

[6] P. Jansson and J. Jeuring. PolyP — a polytypic programming language extension. In ACM, editor, *Conference record of POPL '97, the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the symposium, Paris, France, 15–17 January 1997*, pages 470–482, New York, NY, USA, 1997. ACM Press.

[7] P. Jansson and J. Jeuring. Functional pearl: Polytypic unification. *Journal of Functional Programming*, 8(5):527–536, Sept. 1998.

[8] J. Jeuring and P. Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Tutorial Text 2nd Int. School on Advanced Functional Programming, Olympia, WA, USA, 26–30 Aug 1996*, volume 1129 of *Lecture Notes in Computer Science*, pages 68–114. Springer-Verlag, Berlin, 1996.

[9] M. Jones. Type classes and functional dependencies. In *Proceedings of the 9th European Symposium on Programming, ESOP 2000*, volume LNCS 1782. Springer-Verlag, March 2000.

[10] M. P. Jones. Using parameterized signatures to express modular structure. In 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, pages 68–78, St. Petersburg Beach, Florida, 21–24 Jan. 1996.

[11] M. P. Jones. Typing haskell in haskell. In *Proceedings of the 1999 Haskell Workshop*, pages 68–78, Paris, France, 21–24 Oct. 1999. Published in Technical Report UU-CS-1999-28, Department of Computer Science, University of Utrecht.

[12] J. Launchbury and S. Peyton-Jones. Lazy functional state threads. In *PLDI'94: Programming Language Design and Implementation, Orlando, Florida*, pages 24–35, New York, June 1994. ACM Press.

[13] M. Odersky and K. Läufer. Putting type annotations to work. In ACM, editor, *Conference record of POPL '96, 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: St. Petersburg Beach, Florida, 21–24 January 1996*, pages 54–67, New York, NY, USA, 1996. ACM Press.

APPENDIX

A. MAKEUNIFY

```
data GTstruct s r m =
  B { unifyGT :: GT s r -> GT s r -> m ()
    , matchGT :: GT s r -> GT s r -> m()
    , equalGT :: GT s r -> GT s r -> Bool
    , freshGT :: m (GT s r)
    , occursGT :: Ptr s r -> GT s r -> m Bool
    , colGT :: GT s r -> m(GT s r)
    , pruneGT :: GT s r -> m(GT s r)
    , instanGT :: [GT s r] -> GT s r -> m(GT s r)
    , tofixGT :: GT s r -> m(Fix s)
    , fromfixGT :: Fix s -> GT s r
    }
makeUnify :: Monad m => RSclass s r m -> GTstruct s r m
makeUnify lib =
  B { unifyGT = unify
    , matchGT = match
    , equalGT = equal
    , freshGT = freshVar
    , occursGT = occursIn
    , colGT = col
    , pruneGT = prune
    , instanGT = inst
    , tofixGT = toFix
    , fromfixGT = fromFix
    }
  where
    -- first some common patterns
    seqmap f x = seqRS lib(mapRS lib f x)
    write r x = writeVarRS lib r x
    freshVar = do { r <- newVarRS lib Nothing
                  ; return (MutVar r) }
    prune (typ @ (MutVar ref)) =
      do { m <- readVarRS lib ref
         ; case m of
             Just t ->
               do { newt <- prune t
                  ; write ref (Just newt)
                  ; return newt }
             Nothing -> return typ}
    prune x = return x
    col x =
      do { x' <- prune x
         ; case x' of
            (S y) ->
              do { t <- (seqmap  col y)
                 ; return (S t)}
            (MutVar r) -> return(MutVar r)
            (GenVar n) -> return(GenVar n)}
    occursIn v t =
      do { t2 <- prune t
         ; case t2 of
             S w ->
               do { s <- (seqmap (occursIn v) w)
                  ; return(accRS lib (||) s False)
                  }
             MutVar z -> return(sameVarRS lib v z)
             GenVar n    -> return False }
    varBind r1 t2 =
              do { b <- occursIn r1 t2
                 ; if b
```

```
                then errorRS lib "OccursErr"
          else write r1 (Just t2) }
unify tA tB =
  do { t1 <- prune tA
     ; t2 <- prune tB
     ; case (t1,t2) of
        (MutVar r1,MutVar r2) ->
          if sameVarRS lib r1 r2
             then return ()
             else write r1 (Just t2)
        (MutVar r1,_) -> varBind r1 t2
        (_,MutVar r2) -> varBind r2 t1
        (GenVar n,GenVar m) ->
          if n==m
             then return ()
             else errorRS lib "Gen error"
        (S x,S y) ->
          case matchRS lib x y of
            Nothing -> errorRS lib "ShapeErr"
            Just pairs ->
              mapM_ (uncurry unify) pairs
        (_,_) -> errorRS lib "ShapeErr"
     }
match tA tB =
  do { t1 <- prune tA
     ; t2 <- prune tB
     ; case (t1,t2) of
         (MutVar r1,_) ->
           write r1 (Just t2)
         (GenVar n,GenVar m) ->
           if n==m
              then return ()
              else errorRS lib "Gen error"
         (S x,S y) ->
            case matchRS lib x y of
              Nothing -> errorRS lib "ShapeErr"
              Just pairs ->
                mapM_ (uncurry match) pairs
         (_,_) -> errorRS lib "ShapeErr"
     }
equal x y =
  case (x,y) of
    (MutVar r1,MutVar r2) ->
      sameVarRS lib r1 r2
    (GenVar n,GenVar m) -> m==n
    (S x,S y) ->
      case matchRS lib x y of
        Nothing -> False
        Just pairs -> all (uncurry equal) pairs
    (_,_) -> False
inst sub x =
  do { x' <- prune x
     ; case x' of
         MutVar r -> return(MutVar r)
         GenVar n -> col (sub !! n)
         S x ->
           do { x' <- (seqmap (inst sub) x)
              ; return (S x')
     }        }
fromFix (Fix x) = S(mapRS lib fromFix x)
toFix x =
  do { x' <- prune x
     ; case x of
         MutVar r -> errorRS lib "No vars"
         GenVar m -> errorRS lib "No generic vars"
         S y -> do { y' <- seqmap toFix y
                   ; return(Fix y')
                   }}
```

B. COMMAND LANGUAGE EXAMPLE

```
type Var = String
type Exp = Bool

 data C x
   = If Exp x x       --  if e then s2 else s1
   | While Exp x      --  while e do s
   | Begin x x        --  { s1 ; s2 }
   | Skip             --  {}
   | Assign Var Exp   --  x := e

-- the lowercase constructor convention
ifc e x y = S(If e x y)
while e x = S(While e x)
begin x y = S(Begin x y)
skip = S Skip
assign v e = S(Assign v e)

type Command a = GT C (STRef a)

---- The S structure operators

mapC f (If e x y) = If e (f x) (f y)
mapC f (While e x) = While e (f x)
mapC f ( Begin x y) =  Begin (f x) (f y)
mapC f Skip = Skip
mapC f (Assign v e) = Assign v e

accC acc (If e x y) ans = acc x (acc y ans)
accC acc (While e x) ans = acc x ans
accC acc ( Begin x y) ans = acc x (acc y ans)
accC acc Skip ans = ans
accC acc (Assign v e) ans = ans

seqC (If e x y) =
  do { x' <- x; y' <- y; return (If e x' y')}
seqC (While e x) =
  do { x' <- x; return(While e x')}
seqC ( Begin x y) =
  do { x' <- x; y' <- y; return( Begin x' y') }
seqC Skip = return Skip
seqC (Assign v e) = return(Assign v e)

matchC (If e w x) (If f y z) =
  if f==e then Just[(w,y),(x,z)]
          else Nothing
matchC (While e w) (While f y) =
  if f==e then Just[(w,y)]
          else Nothing
matchC (Begin w x) (Begin y z) = Just[(w,y),(x,z)]
matchC Skip Skip = Just[]
matchC (Assign v e) (Assign u f) =
  if v==u && e==f
     then Just []
     else Nothing
matchC _ _ = Nothing
```