Efficient Functional Unification and Substitution

Atze Dijkstra Arie Middelkoop S. Doaitse Swierstra

Department of Information and Computing Sciences
Universiteit Utrecht
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
{atze,ariem,doaitse}@cs.uu.nl

Abstract

Implementations of language processing systems often use unification and substitution to compute desired properties of input language fragments; for example when inferring a type for an expression. Purely functional implementations of unification and substitution usually directly correspond to the formal specification of language properties. Unfortunately the concise and understandable formulation comes with gross inefficiencies. A seond appoach is to focus on efficiency of implementation. However, efficient implementations of unification and substitution forgo pure functionality and rely on side effects. We present a third, 'best of both worlds', solution, which is both purely functional and efficient by simulating side effects functionally. We compare the three approaches side by side on implementation and performance.

1. Introduction

Although unification arises in many problem areas, for example in theorem proving systems and in Prolog implementations, our inspiration for this paper comes from its application in type checking and inferencing in a Haskell compiler (Dijkstra 2004; Dijkstra et al. 2007; Dijkstra 2005). In Haskell we may write, for example:

$$
\begin{aligned}
&\mathit{first}\ (a, b) = a\\
&x_1 = \mathit{first}\ 3\\
&x_2 = \mathit{first}\ (3, 4)\\
&x_3 = \mathit{first}\ ((3, 4), 5)
\end{aligned}
$$

For *first* we need to infer (or reconstruct) its type $\forall a\ b.(a, b) \rightarrow a$, whereas for x_1, x_2 and x_3 we need to check whether it is permitted to pass the given argument to *first*. Obviously this is not the case for x_1.

In implementations of type systems the reconstruction of yet unknown type information and the check whether known types match is usually done with the help of *unification* of types, the unification paradigm being one of many strategies to solve equations on types imposed by the formal specification of a type system. Types may contain type variables representing yet unknown type information; unification then either matches with returning possible new bindings for such type variables, referred to as *substitution*, or it fails with a type mismatch. For example, for the application of *first* to $(3, 4)$ in the definition of x_2 types $(\mathit{Int}, \mathit{Int})$ and (v_1, v_2) match with bindings for type variables v_1 and v_2; in the right hand side of the definition for x_1 the given argument type Int and expected argument type (v_1, v_2) do not match.

Formally, the unification problem is described as follows (see Knight (Knight 1989)). We define a *term*, denoted by $\{\, s, t \,\}$, to be constructed from *function symbol*s $\{\, f, g \,\}$ and *variable symbol*s $\{\, v, w \,\}$:

$$
\begin{aligned}
t &= f\ (t_1, .., t_n)\\
&\mid\ v
\end{aligned}
$$

Function symbols take a possibly empty sequence of arguments; functions without arguments act as *constant symbol*s.

A *substitution* is a mapping from variables to terms: $\{\, v_1 \mapsto t_1, .., v_n \mapsto t_n \,\}$. We wll use $\{\, \theta, \sigma, \vartheta \,\}$ to refer to substitutions. A substitution can be extended to a function from terms to terms via its application to terms, denoted by $\theta(t)$ or juxtaposition θt when it is clear that substitution application is meant. The term θt denotes the term in which each variable v_i in t in the domain of θ is replaced by $t_i = \theta(v_i)$:

$$
\begin{aligned}
\theta(f\ (t_1, .., t_n)) &= f\ (\theta t_1, .., \theta t_n)\\
\theta(v \qquad\quad\ ) &= t, \{\, v \mapsto t \,\} \in \theta\\
&= v, \mathit{otherwise}
\end{aligned}
$$

Substitutions can be composed: $\sigma\theta t$ denotes t after the application of θ followed by σ. The application to a substitution $\theta = \{\, v_i \mapsto t_i \,\}$ is defined as $\sigma\theta = \sigma \cup \{\, v_i \mapsto \sigma t_i \,\}$. Composition of substitutions is associative, but in general not commutative.

Two terms s and t are *unifiable* if there exists a substitution θ such that $\theta s = \theta t$. The substitution θ is then called the *unifier*, θt the *unification*. A unifier θ is called the *most general unifier* (MGU) if for any other unifier σ, there exists a substitution ϑ such that $\vartheta\theta = \sigma$. Two terms s and t may be *infinitely unifiable* if their unifier binds variables to infinitely long terms. In this paper we prevent this from happening.

The problem From the above definitions we already can see why a straightforward functional implementation will be inefficient. When we directly translate the definition of the substitution application θt to a corresponding function application in a purely functional language like Haskell, each such application will construct a copy s of t, differing only in the free v_i for which θ has a binding. Furthermore, whenever v_i occurs more than once in t, several copies of $\theta(v_i)$ will be present in s. This leads to duplication of work for a subsequent substitution, a situation which occurs when substitutions are composed. Substitution composition is done frequently; this then makes variable replacement in substitutions the culprit, and thus has to be avoided in more efficient implementations of the subsitution process.

A solution with side effects and its derived problems The growth of terms via duplicate copies of substituted variables can be avoided by never replacing variables. Instead we let variables act as pointers to a possible replacement term. This is easily accomplished in imperative languages, but is more difficult in purely functional ones because of the side effects involved: initially a variable will have no replacement bound to it, and when later a replacement is found for the variable the pointer is made to point to the term replacing the variable.

In a functional language like Haskell we achieve this by leaving the side effect free functional world: the IO monad (Haskells imperative environment) and $IORef$s (Haskells pointer mechanism) are then used. This is the approach taken in the GHC (Peyton Jones and Shields 2004; Team 2007) by the type inferencer, with the following consequences:

- Side effects infect: term reconstruction (type inferencing) and related functionality all have to be aware of side effects and loose the benefits of pure functions.
- Once updated, a variable is changed forever after. This, for example, complicates the use of backtracking mechanisms that may need to undo substitutions.

How much we suffer from these consequences depends on the necessities of the program using unification. We found ourselves in a situation where we were hindered by the lack of efficiency of the basic functional solution, and did not want to corrupt the cleanliness of our compiler implementation (Dijkstra 2004; Dijkstra et al. 2007; Dijkstra 2005). Furthermore we wanted the freedom to experiment with temporary assumptions about type variables, instead of fixing knowledge about such variables in one pass directly. So we designed a third solution which is both functionally transparant and efficient. We come back to our rationale and context of this paper in Section 7 after dealing with the technical content.

Our contribution: a solution without side effects A solution infecting an otherwise functional program with side effects can be avoided by simulating side effects purely functional. The essence of an efficient substitution mechanism is to share the binding of a variable instead of copying it. This can be implemented without relying on imperative constructs such as IO in Haskell. Our contribution thus is:

- Present our side effect free efficient functional unification and substitution.
- Compare our solution with the naive purely functional as well as the side effect solution. We look at both the implementation and performance.

Related work We are not aware of other published work describing a solution for unification and substitution similar to the one to be presented here; neither are we aware of side by side presentations with other solutions. The purely functional solution is frequently used in textbook examples (Jones 1999; Pierce 2002), whereas the one with side effects is used when efficiency is important, such as in production quality compilers (Peyton Jones and Shields 2004; Team 2007).

Much work has been done on unification, in fact so much that we only mention some entry points into existing literature, amongst which some surveys (Knight 1989; Baader and Snyder 2001; Gallier 1991) and seminal work by Robinson (Robinson 1965, 1971, 1976), Paterson and Wegman (Paterson and Wegman 1978), and Martelli and Montanari (Martelli and Montanari 1982).

Observable sharing (Claessen and Sands 1999) provides identity of values, allowing equality checking based on this identity. The low level implementation requires side effects, similar to the solution in this paper based on side effects.

The problem we encounter is a consequence of being purely functional. Hiding the problem and its solution can be done by offering unification as a language feature and building the implementation of unification into the language implementation, as done in Prolog and its implementations.

Outline of the remainder of this paper In Section 2 we proceed with the preliminaries for our work, in particular a mini system, formally described, and implemented using the three variants of unification and substitution. In Section 3 we present the purely functional implementation, in Section 4 the one with side effects, and in Section 5 our solution, which we baptise FUNCTIONAL SHARING. We look at performance results in Section 6, discuss in Section 7 and conclude in Section 8.

2. Preliminaries

The essence of the problem: purely functional versus side effects A function f is called *purely functional* (or simply *functional*) when for all invocations $f_1\ x$ and $f_2\ x$ of f parameterized with x, in all execution contexts and all execution orderings, $f_1\ x = f_2\ x$ holds. Given an execution order $f_1\ x_1; e; f_2\ x_2$ with $x_1 = x_2$, then e has a *side effect* when for the execution order $x_1; e; x_2$ the invocations have different results $f_1\ x \not\equiv f_2\ x$. In particular we are interested in computations resulting in terms t. We want t to be purely functional, that is, we want two uses t_1 and t_2 of t always to be equal: $t_1 = t_2$. Naively done this turns out to be inefficient (Section 3), so we forgo pure functionality and allow side effects in e to modify t, that is $t_1 \not\equiv t_2$ in the execution order $t_1; e; t_2$ (Section 4). Finally we recover purely functional behavior by parameterizing t with that part se of e which is responsible for the side effect (Section 5), so once again $t_1\ se = t_2\ se$ in the execution order $t_1\ se; e; t_2\ se$. The side effect of e is modelled explicitly by se instead of being implicit. A side effect means a different se. Different $se_1 \not\equiv se_2$ are passed explicitly as a parameter to functions using a term t, in particular t itself: $t\ se$. In this paper unification yields such t and se, where se is a substitution θ.

Experimental environment Our experimental environment consists of an implementation resembling structures found in many compilers. We thus mimic the actual runtime environment we are interested in, while keeping things as simple as possible. Figure 1 shows the rules for our system; it should be familiar to those acquainted with type systems. Since we want to focus on unification mechanisms without wandering off to type systems, our example system neutrally specifies which values Val are to be associated with a tree $Tree$.

A $Tree$ offers constructs for binding and using program identifiers, as well as constructing and deconstructing pairs of (ultimately) some constant. The concrete syntax is included in comment, the exclamation mark enforces strictness and can be ignored for the purpose of understanding:

```
data Tree                                -- concrete syntax:
  = Constant                             -- C
  | UseBind !String                      -- n
  | DefBind !String !Tree !Tree          -- bind n = x in y
  | Tuple   !Tree   !Tree                -- (x,y)
```

$$\boxed{\Gamma \vdash \mathit{Tree} : \mathit{Val}}$$

$$\frac{}{\Gamma \vdash C : c}\ \text{T.CONSTANT}_D \qquad \frac{(n \mapsto v) \in \Gamma}{\Gamma \vdash n : v}\ \text{T.USEBIND}_D$$

$$\frac{\begin{array}{c}\Gamma \vdash x : v \\ n \mapsto v, \Gamma \vdash y : w\end{array}}{\Gamma \vdash \mathbf{bind}\ n = x\ \mathbf{in}\ y : w}\ \text{T.DEFBIND}_D$$

$$\frac{\begin{array}{c}\Gamma \vdash x : v \\ \Gamma \vdash y : w\end{array}}{\Gamma \vdash (x, y) : (v, w)}\ \text{T.TUPLE}_D \qquad \frac{\Gamma \vdash x : (v, w)}{\Gamma \vdash \mathbf{fst}\ x : v}\ \text{T.FIRST}_D$$

$$\frac{\Gamma \vdash x : (v, w)}{\Gamma \vdash \mathbf{snd}\ x : w}\ \text{T.SECOND}_D$$

Figure 1. Rules for Val of Tree (D)

```
  | First  ! Tree        -- fst x
  | Second ! Tree        -- snd x
  deriving Show
```

The rules associate a *Val* with a *Tree*. Again, a *Val* is inspired by type systems, but for the purposes of this paper it is just some structure, complex enough to discuss unification and substitution. Therefore, in the remainder of this paper a *Val* is a term participating in unification and substitution.

```
data Val                 -- concrete syntax:
  = Pair ! Val ! Val     -- (v,w)
  | Const                -- c
  | Var  ! VarId
  | Err  ! String
  deriving Show
type VarId = Int
```

A *Val* has two alternatives in its structure which do not have a *Tree* constructor as counterpart: an construct *Var* for encoding variables as used in unification and substitution, and an construct *Err* for signalling errors.

Test examples For example, with the following tree:

$$\begin{array}{ll}\mathbf{bind}\ v_1 = \mathit{constant} & \mathbf{in} \\ \mathbf{bind}\ v_2 = (v_1, v_1) & \mathbf{in} \\ \mathbf{bind}\ v_3 = (\mathbf{snd}\ v_2, \mathbf{fst}\ v_2) & \mathbf{in}\ v_3\end{array}$$

the rules associate the value (c, c). This example is one of the test cases we use, where we also vary in the number of bindings similar to v_3. The value of the tree is always (c, c).

The second example we use for testing infers a *Val* of exponential size in terms of the number of bindings similar to v_4, yielding values $((c, c), ((c, c), (c, c)))$ and so forth for increasing numbers of similar bindings:

$$\begin{array}{ll}\mathbf{bind}\ v_1 = \mathit{constant} & \mathbf{in} \\ \mathbf{bind}\ v_2 = (v_1, v_1) & \mathbf{in} \\ \mathbf{bind}\ v_3 = (\mathbf{fst}\ v_2, v_2) & \mathbf{in} \\ \mathbf{bind}\ v_4 = (\mathbf{snd}\ v_3, (v_2, v_2)) & \mathbf{in}\ v_4\end{array}$$

The first example provide typical programming language input, with many small definitions, whereas the second example provides a worst case scenario. We label the tests respectively LINEAR and EXPONENTIAL.

$$\boxed{\theta_{\mathbf{in}}; \Gamma \vdash \mathit{Tree} : \mathit{Val} \rightsquigarrow \theta_{out}}$$

$$\frac{}{\theta; \Gamma \vdash C : c \rightsquigarrow \theta}\ \text{T.CONSTANT}_A$$

$$\frac{(n \mapsto v) \in \Gamma}{\theta; \Gamma \vdash n : \theta\ v \rightsquigarrow \theta}\ \text{T.USEBIND}_A$$

$$\frac{\begin{array}{c}\theta; \Gamma \vdash x : v \rightsquigarrow \theta_x \\ \theta_x; n \mapsto v, \Gamma \vdash y : w \rightsquigarrow \theta_y\end{array}}{\theta; \Gamma \vdash \mathbf{bind}\ n = x\ \mathbf{in}\ y : w \rightsquigarrow \theta_y}\ \text{T.DEFBIND}_A$$

$$\frac{\begin{array}{c}\theta; \Gamma \vdash x : v \rightsquigarrow \theta_x \\ \theta_x; \Gamma \vdash y : w \rightsquigarrow \theta_y\end{array}}{\theta; \Gamma \vdash (x, y) : (\theta_y v, w) \rightsquigarrow \theta_y}\ \text{T.TUPLE}_A$$

$$\frac{\begin{array}{c}\theta; \Gamma \vdash x : vw \rightsquigarrow \theta_x \\ v, w\ \text{fresh} \\ (v, w) \equiv vw \rightsquigarrow \theta_m\end{array}}{\theta; \Gamma \vdash \mathbf{fst}\ x : \theta_m \theta_x v \rightsquigarrow \theta_m \theta_x}\ \text{T.FIRST}_A$$

$$\frac{\begin{array}{c}\theta; \Gamma \vdash x : vw \rightsquigarrow \theta_x \\ v, w\ \text{fresh} \\ (v, w) \equiv vw \rightsquigarrow \theta_m\end{array}}{\theta; \Gamma \vdash \mathbf{snd}\ x : \theta_m \theta_x w \rightsquigarrow \theta_m \theta_x}\ \text{T.SECOND}_A$$

Figure 2. Rules for Val of Tree (A)

From declarative rules to an algorithm The rules in Figure 1 are declarative of nature, notationally indicated by the suffix D in the names of the rules. The rules in Figure 2 provide an algorithmic equivalent, indicated by the suffix A. The essential difference lies in rule T.FIRST (and rule T.SECOND) where the declarative variant simply states some restriction on a *Val*. In this case the argument of **fst** is constrained to have a *Val* of the form (v, w). This is typical of declarative rules: a restriction is just stated. The algorithmic variant however needs to computationally check the restriction and compute its constituents. The rules in Figure 2 do this in a way typical of algorithmic variants: the constraining structure (v, w) is unified with the structure to be checked. The constraining *Val* is built from variables guaranteed to be unique (called *fresh*), whereas the extraction is done by simply using the unique variables together with a substitution θ holding possible additional information about the variables.

The algorithmic version threads a substitution θ through its computation, while gathering information about the *Var*s participating in the construction of the *Val* associated with the root of the tree. The rules maintain the invariant that θ is already taken into account in resulting t's, that is $\theta t = t$, where t refers to the *Val* component of the conclusion.

A substitution θ is represented by a variable mapping *VMp*, mapping identifiers *VarId* of variables to terms *Val*:

```
newtype VMp = VMp (Map VarId Val)
```

We need the usual functions for constructing and querying, for which we only give the signatures:

```
emptyVM :: VMp
(|?)    :: VarId → VMp → Maybe Val   -- lookup
```

```
vmUnit  :: VarId → Val → VMp
vmUnion :: VMp   → VMp → VMp
```

The rules in Figure 2 thus specify a particular strategy to find a solution for all types represented by the metavariable occurrences of v, w in Figure 1, constrained by the declarative rules. Usually one would now prove soundness and completeness between these two sets of rules; we do not do so here as we are exploring the behavior of the substitution mechanism.

Contextual information Γ holding assumptions for program identifiers is encoded by an environment *Env*:

```
newtype Env = Env (Map String Val)
```

We omit definitions for functions on *Env* and assume their names are understandable enough to indicate their meaning.

Finally, in the following we restrict ourselves to first order unification, and do not allow infinite values.

3. Substitution by copying

We first discuss the purely functional reference implementation to which we compare the others. We present the overall computational structure on which we vary in the subsequent alternate implementations. We label this solution by FUNCTIONAL.

Figure 3 shows the implementation of the algorithmic rules (Figure 2). The rules strongly suggest a direction in which information flows over a tree, upward or synthesized for e.g. *Val*, downward or inherited for e.g. Γ, and chained for θ. We use a state monad to encode this flow:

```
data St = St{ stUniq :: !VarId
            , stEnv  :: !Env
            , stVMp  :: !VMp
            }
type Compute v = State St v
```

The *Compute* state monad threads the following three values through the computation:

- a counter used for creating fresh variables,
- an environment *Env* holding Γ,
- and a variable mapping *VMp* corresponding to both the inherited and synthesized substitution θ.

Strictly speaking the *Env* needs not be threaded, but we prefer to avoid the additional complexity of placing this part of the state into a reader monad and using the associated monad transformers.

Substituting In a *Val* substitutable variables may occur, and thus also in *Env*. Substitutabilty is expressed by the class *Substitutable*:

```
class Substitutable x where
  (|@) :: VMp → x → x
  ftv  :: x → Set VarId
```

The application θx of a substitution θ to some x is expressed by the function |@. The function *ftv* computes the free variables of a x.

Substitution over a *Val* is straightforwardly encoded as a recursive replacement:

```
instance Substitutable Val where
  s |@ v
    = sbs s v
    where sbs s (Pair v w) = Pair (sbs s v) (sbs s w)
          sbs s v@(Var i)  = case i |? s of
```

```
treeCompute :: Tree → Compute Val
treeCompute t =
  case t of
    Constant     → return Const
    UseBind n    →
      do st ← get
         case envLookup n (stEnv st) of
           Just v → return (stVMp st |@ v)
           _      → return (Err ("not found: " ++ show n))
    DefBind n x y →
      do v  ← treeCompute x
         st ← get
         let env = stEnv st
         put (st{ stEnv = envUnit n v `envUnion` env })
         w  ← treeCompute y
         st ← get
         put (st{ stEnv = env })
         return w
    Tuple x y →
      do v  ← treeCompute x
         w  ← treeCompute y
         st ← get
         return (Pair (stVMp st |@ v) w)
    First x      →
      do vw     ← treeCompute x
         [v, w] ← newVars 2
         valUnify (Pair v w) vw
         st     ← get
         return (stVMp st |@ v)
    Second x     →
      do vw     ← treeCompute x
         [v, w] ← newVars 2
         valUnify (Pair v w) vw
         st     ← get
         return (stVMp st |@ w)
```

Figure 3. Computation of *Val* over *Tree* in the FUNCTIONAL solution

```
                                Just v' → v'
                                _       → v
          sbs s v          = v
  ftv (Var i)      = Set.singleton i
  ftv (Pair v w)   = ftv v `Set.union` ftv w
  ftv _            = Set.empty
```

The composition of two substitutions, that is, substituting over a substitution itself means taking the union of two *VMp*s and ensuring that all *Val*s in the previous substitution are substituted over as well, the previous substitution being the second operand to |@:

```
instance Substitutable VMp where
  s |@ (VMp m) = s `vmUnion` VMp (Map.map (s |@) m)
  ftv  (VMp m) = Map.fold (λv fv → fv `Set.union` ftv v)
                          Set.empty m
```

Applying the |@ from this instance over and over again makes the update of a substitution with new bindings for variables a costly operation, and alone is responsible for a major part of the efficiency loss of this solution.

```
valUnify :: Val → Val → Compute Val
valUnify v1 v2
  = uni v1 v2 where
  uni   x@(Const) (Const)           = return x
  uni   x@(Var i) (Var j) | i == j  = return x
  uni   (Var i)   y                 = bindv i y
  uni   x         y@(Var _)         = uni y x
  uni   (Pair p q) (Pair r s)       =
    do pr    ← uni p r
       st1   ← get
       qs    ← uni (stVMp st1 |@ q) (stVMp st1 |@ s)
       st2   ← get
       return (Pair (stVMp st2 |@ pr) qs)
  uni   _         _                 = err "fail"
  bindv i v
    | Set.member i (ftv v)          = err "inf"
    | otherwise                     =
        do   st ← get
             put (st{stVMp = vmUnit i v |@ stVMp st})
             return v
  err x = return (Err x)
```

Figure 4. Val unification in the FUNCTIONAL solution

Value unification Unification tells us whether two values can be made syntactically equal, and a substitution tells us which variables in these values have to be bound to another value to make this happen. Figure 4 shows the code for *valUnify*, which unifies two *Val*s, thus implementing the operator $\equiv$ used by e.g. rule T.FIRST in Figure 2. Function *valUnify* applied to t and s yields the unification θt directly and the substitution θ via the state of *Compute*. A unification may also fail, which we simply signal by the *Err* alternative of *Val*.

We note that always returning the unification θt is convenient but strictly not necessary, as θ and t can also be combined outside *valUnify*. Now additional *Val*s are constructed, however, we could not observe an effect on performance (see Section 6 for further discussion). Encoding an error as part of *Val* is also a matter of convenience, and merely to show where errors arise; we do not report those errors and in our test cases no errors arise.

The function *valUnify* assumes that its *Val* parameters do not contain free variables bound by the substitution *stVMp* passed via the *Compute* state. Whenever a variable is encountered during the comparison of the two types being unified, it is bound to the other comparand. We prevent recursive bindings causing infinite values, like $v \mapsto (v, v)$, from occurring by performing the so called *occurs check* done in *bindv*, and by checking on the trivial unification of v with v.

Unification proceeds recursively over *Pair*s. We ensure the invariant that *Val*s passed for further comparison always have the most recent substitution already applied to them.

Fresh variables Besides the environment and the current substitution, *St* contains state for the generation of fresh variables. Function *newVar* increments the counter *stUniq* in the *Compute* state and returns *Var*s with unique *VarId*s:

```
newVar     :: Compute Val
newVar = do st ← get
            let fresh = stUniq st
            put (st{stUniq = fresh + 1})
            return (Var fresh)
```

```
main :: IO ()
main
  = do ((kind : '/' : dep) : (output : _) : variant : _)
         ← getArgs
       let t = case kind of
                 'a' → mkLinearTree (read dep)
                 'b' → mkExponentialTree (read dep)
                 _   → error [kind]
           (r, _) = runState (treeCompute t) emptySt
       when (output == 'p')
            (do putPPLn (pp t)
                putPPLn (pp r)
            )
       putStrLn (r `seq` "done")
```

Figure 5. Toplevel test environment

Function *newVars* conveniently returns a group of such variables:

```
newVars    :: Int → Compute [Val]
newVars n = sequence [newVar | _ ← [1 .. n]]
```

Computing a Val over a Tree All ingredients for Figure 3 come together in the alternative for e.g. rule T.FIRST:

```
First x →
  do vw     ← treeCompute x
     [v, w] ← newVars 2
     valUnify (Pair v w) vw
     st     ← get
     return (stVMp st |@ v)
```

We closely follow the algorithmic variant of the rule by recursing over the x component of **fst** x, allocating fresh variables, using these to match the value of x and returning its first component with the most recent substitution applied. We also slightly deviate from the rule by threading the full *Compute* state through *valUnify* instead of computing additional bindings only.

Finally, *treeCompute* is invoked by a toplevel test environment which first constructs a tree as specified by commandline arguments, then calls *treeCompute*, enforces a deep evaluation of the result and prints the result, also depending on commandline arguments. See Figure 5 for further details not explained here.

This completes our basic reference implementation, often used for its simplicity in explanations, but avoided in real world systems because of the time and memory spent in copying and substituting over the content pointed to by variables.

4. Substitution by sharing

We can avoid the copying of *Val*s during substitution in the previous solution by sharing the content bound to variables. Variables become pointers[1] in a directed acyclic graph (DAG) representation of *Val* instead of a tree representation as used by the FUNCTIONAL solution (Paterson and Wegman 1978). We use an *IORef* to encode such a pointer (Peyton Jones and Shields 2004), with utility functions like *newRef* for hiding its use. Note that *refRead* is not returning a *Compute* monad; a tricky point we come back to at the end of this section. We label this solution SHARING.

```
data Val               -- concrete syntax:
  = Pair ! Val ! Val   -- (v,w)
```

[1] We still need the *VarId* fields because of the computation of *ftv* returning a *Set*; *IORef* is not an instance of *Ord* required for *Set*.

```
valUnify :: Val → Val → Compute Val
valUnify v1 v2
  = uni v1 v2 where
  uni    x@(Const)   (Const)                = return x
  uni    x@(Var i _) (Var j _) | i == j     = return x
  uni    (Var _ r)   y         | isJust mbv = uni v y
    where mbv = refRead r
          v   = fromJust mbv
  uni    x@(Var _ _) y                      = bindv x y
  uni    x           y@(Var _ _)            = uni y x
  uni    (Pair p q)  (Pair r s)             =
    do pr    ← uni p r
       qs    ← uni q s
       return (Pair pr qs)
  uni    _           _                      = err "fail"
  bindv (Var i r) v
    | Set.member i (ftv v)                  = err "inf"
    | otherwise                             =
       do   refWrite r (Just v)
            return v
  err x = return (Err x)
```

Figure 6. Val unification in the SHARING solution

```
    | Const               -- c
    | Var   ! VarId ! Ref
    | Err   ! String
  deriving Show
type     RefContent = Maybe Val
newtype Ref        = Ref (IORef RefContent)
data St = St{ stUniq :: !VarId
            , stEnv  :: !Env
            }
type Compute v = StateT St IO v
newRef   :: Compute Ref
refRead  :: Ref → RefContent
refWrite :: Ref → RefContent → Compute ()
newRef = do r ← lift $ newIORef Nothing
            return (Ref r)
```

In essence, we now store the substitution which maps variables to values directly in a *Var*. Hence we do not need the *VMp* in the *Compute* state anymore. On the other hand, we need to combine the *State* monad with the *IO* monad because of the use of *IORef*. A fresh variable now also gets a fresh shared memory location *Ref*, initialized to hold nothing:

```
newVar    :: Compute Val
newVar = do st ← get
            let fresh = stUniq st
            put (st{ stUniq = fresh + 1})
            r  ← newRef
            return (Var fresh r)
```

Unification now has to be aware that variables are pointers: the SHARING solution is presented in Figure 6. Relative to the FUNCTIONAL solution we need to modify the following:

- When comparing a variable *Var* we no longer can assume that the variable is still unbound. Hence we need to inspect its *Ref* and use it for further comparison.
- Binding a variable in *bindv* now also involves updating the reference with the bound value.
- There is no *VMp* threaded through the *Compute* state, hence we need not maintain the invariant that it is always applied, for example when comparing *Pair*s. This is now guaranteed via the *Ref* mechanism.

The implementation of *treeCompute* becomes simpler, because we need not apply the *VMp* here either. As before, we highlight the *First* case branch for rule T.FIRST; also for the other alternatives the only difference with the FUNCTIONAL solution is the removal of the application of *VMp*.

```
First x →
  do vw     ← treeCompute x
     [v, w] ← newVars 2
     valUnify (Pair v w) vw
     return v
```

The substitution mechanism is completely hidden as a side effect throughout the *Compute* state.

Finally, when computing free variables one also has to be aware of *Ref*s. Since we no longer have a need for class *Substitutable* we define *ftv* as a separate function:

```
ftv :: Val → Set VarId
ftv (Var i r)   = case refRead r of
                    Just v → ftv v
                    _      → Set.singleton i
ftv (Pair v w) = ftv v `Set.union` ftv w
ftv _          = Set.empty
```

The price we have to pay for this solution is that we only may have at most one binding for a *Var*, the one stored in the *Var* itself. This is problematic if we want to have more than one binding during the computation, for example when we want to compute a tentative value and later backtrack on it (Dijkstra 2005; Dijkstra and Swierstra 2006a). We have lost the parameterizability of the binding by introducing side effects and giving up purely functional behavior of substitutions.

The use of *IORef* has other, more subtle, consequences typical of the use of monads. For the sake of clarity all implementations are kept as similar as possible, for example if we look in advance at Figure 7 alongside Figure 6 we can see the case for *Var* in *uni* uses |? in the next solution and *refRead* in the current solution. However, the implementation of *refRead* relies on *unsafePerformIO*:

```
refRead :: Ref → RefContent
refRead (Ref r) = unsafePerformIO $ readIORef r
```

Getting rid of *unsafePerformIO* is possible, the consequence is that we need to encode the function *uni* in *valUnify* differently because we cannot refer to the content of the *Ref* in the guard of the *Var* case anymore:

```
uni (Var _ r)       y
  do mbv ← refRead' r
     case mbv of
       Just v → uni v y
       _      → ?? wrong branch after all
refRead' :: Ref → RefContent
refRead' (Ref r) = readIORef r
```

In Haskell we have no way to backtrack on a case alternative after having committed to it, which is exactly what we must do after *Ref* inspection and finding out no binding exists for the variable.

```
valUnify :: Val → Val → Compute Val
valUnify v1 v2
  = do {st ← get; uni st v1 v2} where
  uni st x@(Const) (Const)                = return x
  uni st x@(Var i) (Var j) | i == j       = return x
  uni st (Var i)   y       | isJust mbv   = uni st v y
      where mbv = i |? stVMp st
            v   = fromJust mbv
  uni st (Var i)   y                      = bindv st i y
  uni st x         y@(Var _)              = uni st y x
  uni st (Pair p q) (Pair r s)            =
    do pr  ← uni st p r
       st2 ← get
       qs  ← uni st2 q s
       return (Pair pr qs)
  uni _ _          _                      = err "fail"
  bindv st i v                            =
    do put (st{stVMp = vmUnit i v |@ stVMp st})
       return v
  err x = return (Err x)
```

Figure 7. Val unification in the FUNCTIONAL SHARING solution

We find ourselves stuck between the desire to maintain clarity and the desire to avoid *unsafePerformIO*.

Finally, in similar spirit we attempted to use *STRef* and the *ST* monad in order to further simplify this FUNCTIONAL SHARING solution; we discuss in Section 7 why we did abandon this approach.

5. Substitution by functional shared memory

We regain purely functional behavior of the unification and substitution machinery by letting a *Var* itself –once again– be unaware of its content, and thus decouple it from the particular baked-in way *IORef*s implement the notion of pointers to memory content. Instead we implement our own dereferencing mechanism by combining *VMp*s from the FUNCTIONAL solution with the pointer based approach of the SHARING solution. We use the *Val* definition of the FUNCTIONAL solution, and adapt the *valUnify* function of the SHARING solution: instead of *IORef*s we create 'do it yourself' memory in the *VMp* as shown in Figure 7. The key difference with SHARING is that the dereferencing required for a variable now is implemented via a lookup in the threaded *stVMp*. The key commonality with SHARING is that we do not replace a variable; we do not apply the substution to a variable but only use the variable itself.

We now also can avoid the expensive copying because we follow pointers instead of accessing a copied value directly. The implementation of the *Substitutable VMp* instance no longer needs to update the 'previous' VMp, a subtle but most effective memory saving change:

```
instance Substitutable VMp where
  s |@ s2 = s `vmUnion` s2
```

Dereferencing and infinite values The consequence of derefencing via a table lookup is a performance loss because such a lookup is expensive compared to a plain memory dereference. Both *valUnify* and its use of *ftv* now require such table lookups. Our design choice is to avoid excessive dereferencing by not using *ftv* at all during unification, and consequently omitting the occurs check from unification. In turn this means that unification may return a substitution with cycles, and we have to deal with infinite values and the occurs check elsewhere, that is, all functions traversing a *Val* need to be aware that an infinite value may indirectly occur via a substitution.

For example, we need to check during application of a substitution to a *Val*. We adapt the application of a subsitution to a *Val* to implement the occurs check: we return an error whenever a substitution for a variable occurs twice, marked by its presence in the set of dereferenced variables *visited*, thus preventing the formation of cycles:

```
instance Substitutable Val where
  s |@ v
    = sbs Set.empty s v
    where sbs visited s (Pair v w) = Pair v' w'
            where v' = sbs visited s v
                  w' = sbs visited s w
          sbs visited s v@(Var i) =
            case i |? s of
              Just v'
                | Set.member i visited
                    → Err "inf"
                | otherwise
                    → sbs (Set.insert i visited) s v'
              _     → v
          sbs visited s v          = v
```

Actually, the necessity for such a check depends on the context in which unification and substitution are used. In this case we could have done without the check because a binding for a variable leading to an infinite value, like $v \mapsto (v, v)$, only arises when we would have had recursive references to bindings in the *Tree* language. Other languages of course have a need for the check; for example in Haskell the following leads to an infinite type for the argument of f, unless accompanied by an explicit type signature:

```
f x = f (x, x)
```

For *valUnify* we have to look harder for an example leading to infinite recursion of *valUnify*. This is because we only can recurse infinitely when two values unfold in parallel in the same way, for example when unifying v and w given bindings $v \mapsto (v, v)$ and $w \mapsto (w, w)$ or similar pairs of bindings with pairwise recursion. The following Haskell program gives rise to such a situation if it were not for binding group analysis which prevents the three definitions to be analysed together:

```
f x = f (x, x)
g x = g (x, x)
h x = (f x, g x)
```

The unification function *valUnify* now has to be adapted to check for variables which are already expanded, in the same way as done for |@ on *Val* above.

We come back to its effect on performance (by putting the occurs check back into *valUnify*) when discussing performance (Section 6).

Finally, for the result to be usable without being aware of a *VMp*, we apply the substitution outside *treeCompute*, in the toplevel test function. For example, our pretty printing is unaware of a *VMp*. Again, we come back to this because of its degrading effect on performance.

6. Performance results

We compared the three solutions, FUNCTIONAL, SHARING and FUNCTIONAL SHARING, by running two test trees, LINEAR and

test	depth	FUNCTIONAL sec	FUNCTIONAL Mb	SHARING sec	SHARING Mb	FUNCTIONAL SHARING sec	FUNCTIONAL SHARING Mb	FUNCTIONAL SHARING NO TOP SUBST sec	FUNCTIONAL SHARING NO TOP SUBST Mb	FUNCTIONAL SHARING OCCUR CHECK sec	FUNCTIONAL SHARING OCCUR CHECK Mb
LINEAR	500	0.70	70.7	0.06	1.8	0.03	2.7	0.03	2.7	0.50	2.9
	1100	3.90	376.2	0.29	3.2	0.07	4.7	0.08	4.7	3.03	5.2
	1600	8.07	629.5	0.60	4.9	0.13	6.4	0.13	6.4	7.39	7.4
EXPONENTIAL	20	0.04	4.5	0.01	1.3	0.01	2.1	0.01	1.3	0.01	1.3
	25	0.87	60.7	0.10	1.3	0.20	13.7	0.07	1.3	0.08	1.3
	28	5.61	436.7	0.56	1.3	1.34	107.7	0.37	1.3	0.42	1.3

Figure 8. Performance results

EXPONENTIAL, with various depths. Both tests are described in Section 2 and are characterized by manipulation of Vals, linear and exponential in the number of bindings introduced (which equals the depth of the tree) by the test $Tree$s. The results are shown in Figure 8. The FUNCTIONAL, SHARING and FUNCTIONAL SHARING variants are already described; the remaining variants are introduced and discussed hereafter as part of the performance analysis. The memory sizes in Figure 8 correspond to the maximum resident set size as reported by the Unix time command, and is because of the GHC garbage collection an overestimate of the actual memory requirements. However, it still gives an indication of the proportial memory use. Tests were run on a MacBook Pro 2.2Ghz Intel Core 2 Duo with 2GB memory, MacOS X 10.5.2, the programs compiled with GHC 6.8.2 without optimization flags. Each test was run twice, the results taken from the second run. Further runs did not give significant variation in the results.

We observe the following:

- On the linear test cases all but the FUNCTIONAL variant perform equally well, using small amounts of memory.
- On the exponential test case the SHARING variant runs best, the FUNCTIONAL variant worst, especially in terms of memory. The FUNCTIONAL SHARING variant sits in between. It turned out this was caused by the substitution still applied in the toplevel test function. Variant FUNCTIONAL SHARING NO TOP SUBST has this substitution removed and replaced by code forcing a deep evaluation over the Val and substitution jointly. The results are now similar to those of SHARING, even a bit faster.
- When tests are run with GHC optimization switched on, the absolute numbers drop, but only by a relative small factor of at most 1.5; the relative performance remains the same. We therefore did not include these numbers.
- Omitting the occurs check in FUNCTIONAL SHARING is worthwhile. Variant FUNCTIONAL SHARING OCCUR CHECK includes the occurs check relative to the fastest variant FUNCTIONAL SHARING NO TOP SUBST: it is significantly slower for the linear test. This is an apparent trade-off between efficiency and responsibility of doing the occurs check: encapsulated in unification or outside of unification. Carrying the 'occurs check' responsibility implies additional program complexity, but, in the light of variant FUNCTIONAL SHARING NO TOP SUBST, no loss of efficiency. We did not further experiment and measure this. In our real world use (Dijkstra et al. 2007) of our solution only a limited number of functions is aware of substitutions, yielding a sufficient gain in efficiency.
- We noted that $valUnify$ constructs a fresh copy for the resulting unification θt of t and s. Replacing such construction for FUNCTIONAL SHARING by a $Bool$ indicating success or failure did not improve performance; we therefore did not include performance numbers for this variation. However, it confirms that the copying involved in the substitution mechanism indeed is the performance bottleneck, and not the copying of terms occurring in $valUnify$.

7. Discussion

Implementation alternative: use of ST and $STRef$ In order to get rid of IO and $IORef$ in solution SHARING we did consider the use of ST and $STRef$ instead. ST may be seen as a more general IO; vice versa IO corresponds to a ST specialized to the $RealWorld$. This did not turn out very well because the use of our state St and the restrictions imposed upon any state by ST do not combine. Using the and ST means running it via $runST$, which in turn means hiding of state; we want it to be visible so we can use its content. This can be remedied by adding even more use of unsafe IO constructs or more clever monadic compositions by the use of monad transformers. Or we could place the full machinery in the ST monad, forgo the use of monad transformers, and put all state in a $STRef$; we did not explore this option, as we doubt it will bring additional benefit. In summary, our ST approaches defeat the purpose of getting rid of IO and achieving simplicity.

Context In the introduction we expressed the desire to get rid of IO and mechanisms with side effects. One could ask why we do want this because IO works well enough, isn't it? Our longterm goal is to be able to describe and implement languages aspect wise, with tools and mechanisms to build description and implementation compositionally from such aspects. Currently we achieve this by using attribute grammars (Baars et al. 2004) and a type rule domain specific language (Dijkstra and Swierstra 2006b), which allow us to specify aspectwise, with tools to construct working compilers (Dijkstra et al. 2007). This solution roughly corresponds to the use of monads for each aspect with monad transformers combining these (Jones 1995). The difference lies in the obligation of the use of monad transformers to specify their construction on the type level, and thereby fixing the ordering of use of state and computation of results. Both become difficult to do, if not impossible, when the number of basic monads, each of which corresponds to an independent implementation of a language aspect, increases and their interaction becomes more complex. Adding side effect to this mix limits –in our view– the practical applicability of monads for the implementation of complex languages.

The gist of these observations and the above experience with the ST monad is that we want to avoid monads and side effects in particular, in order to have better compositionality. Our solution FUNCTIONAL SHARING contributes to just that by separating the notion of value and what we get to know about it as part of a particular strategy of finding out more about such a value. Of course, some interaction cannot be avoided, a Val has Var alternative after all, but at least any knowledge about a Var is never irrevocably hardcoded in the Var: it is manipulated separately, thus allowing its compositional use.

8. Conclusion

Avoiding copying and the resulting memory allocation, and using sharing mechanisms instead, pays off. This is the overall conclusion which can be drawn. Furthermore, using a solution with $IORef$ based side effect can be avoided without performance penalties; there is no need to fall back on the IO monad to achieve acceptable levels of performance.

Our 'best of both worlds' solution has been implemented as part of EHC, the essential Haskell Compiler (Dijkstra 2004; Dijkstra et al. 2007); The programs discussed in this paper can also be found there as part of its experiments subdirectory. Because we have based our EHC implementation on attribute grammars, avoiding the dependency on IO and side effects, the efficient functional solution was critical to the success of the implementation of the type system in EHC.

References

Franz Baader and Wayne Snyder. *Unification Theory*, chapter 8, pages 447–531. Elsevier Science Publishers, 2001.

Arthur Baars, S. Doaitse Swierstra, and Andres Löh. Attribute Grammar System. `http://www.cs.uu.nl/groups/ST/Center/AttributeGrammarSystem`, 2004.

Koen Claessen and David Sands. Observable Sharing for Functional Circuit Description. In *Asian Computing Science Conference*, number 1742 in LNCS, pages 62–73, 1999.

Atze Dijkstra. EHC Web. `http://www.cs.uu.nl/groups/ST/Ehc/WebHome`, 2004.

Atze Dijkstra. *Stepping through Haskell*. PhD thesis, Utrecht University, Department of Information and Computing Sciences, 2005.

Atze Dijkstra and S. Doaitse Swierstra. Exploiting Type Annotations. Technical Report UU-CS-2006-051, Department of Computer Science, Utrecht University, 2006a.

Atze Dijkstra and S. Doaitse Swierstra. Ruler: Programming Type Rules. In *Functional and Logic Programming: 8th International Symposium, FLOPS 2006, Fuji-Susono, Japan, April 24-26, 2006*, number 3945 in LNCS, pages 30–46. Springer-Verlag, 2006b.

Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. The Structure of the Essential Haskell Compiler, or Coping with Compiler Complexity. In *Implementation of Functional Languages*, 2007.

Jean H. Gallier. Unification Procedures in Automated Deduction Methods Based on Matings: A Survey. Technical Report MS-CIS-91-76, University of Pennsylvania, Department of Computer and Information Science, 1991.

Mark P. Jones. Functional Programming with Overloading and Higher-Order Polymorphism. In *First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden*, number 925 in LNCS. Springer-Verlag, may 1995.

Mark P. Jones. Typing Haskell in Haskell. In *Haskell Workshop*, 1999.

Kevin Knight. Unification: a multidisciplinary survey. *ACM Computing Surveys*, 21(1):93–124, Mar 1989.

Alberto Martelli and Ugo Montanari. An Efficient Unification Algorithm. *ACM TOPLAS*, 4(2):258 – 282, April 1982.

M.S. Paterson and M.N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158–167, Apr 1978.

Simon Peyton Jones and Mark Shields. Practical type inference for arbitrary-rank types, 2004.

Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, Jan 1965.

J.A. Robinson. Computational logic: The unification computation. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, pages 63–72, 1971.

J.A. Robinson. Fast unification. In *Theorem Proving Workshop*, 1976.

GHC Team. GHC Developer Wiki. `http://hackage.haskell.org/trac/ghc`, 2007.