Practical Type Inference for Polymorphic Recursion: an Implementation in Haskell

Cristiano Vasconcellos[1] **, Lucília Figueiredo**[2] **, Carlos Camarão**[3]

[1]Departamento de Informática,
Pontifícia Universidade Católica do Paraná, Curitiba, PR.

[2]Departamento de Computação,
Universidade Federal de Ouro Preto, Ouro Preto, MG.

[3]Departamento de Ciência da Computação,
Universidade Federal de Minas Gerais, Belo Horizonte, MG.

`damiani@ppgia.pucpr.br,lucilia@dcc.ufmg.br,camarao@dcc.ufmg.br`

Abstract. *This paper describes a practical type inference algorithm for typing polymorphic and possibly mutually recursive definitions, using Haskell to provide a high-level implementation of the algorithm.*

Keywords: Programming Languages, Type Inference, Polymorphic Recursion

1. Introduction

In general, polymorphic recursion occurs in functions defined over *nested* (also called *non-uniform* or *non-regular*) data types, i.e. data types whose definitions include a recursive component that is not identical to the type being defined. Several examples of such data types and interesting functions that operate on them have been presented [Connelly and Morris, 1995, Okasaki, 1997, Okasaki, 1998, Bird and Meertens, 1998].

Languages with support for parametric polymorphism [Milner, 1978, Damas and Milner, 1982], such as Haskell [Jones et al., 1998, Thompson, 1999] and SML [Milner et al., 1989], use one of the following two approaches for the treatment of recursive definitions.

The first approach imposes a restriction on recursive definitions, either considering recursive definitions to be monomorphic (as in e.g. SML), or allowing polymorphic recursion only when the programmer explicitly annotates the polymorphic types (as in e.g. Haskell). In this approach, the language processor front-end separates the definitions occurring in a program into non-mutually recursive binding groups, by examining the program's call graph, and performs a topological sort of these binding groups for determining an order in which to type the definitions.

The second approach allows polymorphic recursive definitions without a priori imposing restrictions, but the type inference algorithm uses in this case a user-configurable iteration limit for stopping the type inference process and rejecting input programs when this limit is exceeded (as in e.g. Mercury [Henderson et al., 2003]).

The obligation on programmers to annotate types of polymorphic recursive definitions goes against the intent of type inference, which is to give programmers the freedom to choose to make or not type annotations, depending on whether they think it is appropriate or not. Furthermore, it has the opposite effect of usual type annotations (in languages with support for type inference and parametric polymorphism), which is to specify a type that is an instance of the inferred most general type.[1]

In [Henglein, 1993, Kfoury et al., 1993] the equivalence of typability in the Milner-Mycroft calculus — that extends the Damas-Milner calculus with polymorphic recursion — with semi-unifiability was proved. In this paper we do not intend to discuss the question of decidability of the semi-unification problem, which is now being questioned in another paper [Figueiredo and Camarão, 2002]. Our aim here is to present a type inference algorithm — called `MMo` — that has been used with success by us and is expected to behave quite well and be useful in practice. In all examples we have seen in the literature (including all examples given in this paper), one iteration (unification) has been enough to infer the types of expressions involving polymorphic recursion (with the exception of contrived examples presented in this paper specifically provided as worst case examples of time complexity for the type inference algorithm). As occurs in the case of type inference for core ML (see e.g. [Kanellakis and Mitchell, 1989, Mairson, 1990, Kanellakis, Mairson and Mitchell, 1991, Mitchell, 1996]), there exist cases where the time complexity of `MMo` is exponential on the size of the input program, but these examples do not occur in practice. Both in the case of core ML and in the case of polymorphic recursion, exponentiality requires the existence of so-called *big types*: a big type of an expression e is a type that has a number of type variables that is exponentially larger than the size of e. Big types do not represent useful abstractions. For small types, the time complexity of the algorithm is polynomial (cf. [Henglein, 1993, Theorem 8, page 285]. This explains why the type inference algorithm for ML behaves well in practice, and is exactly the reason for our expectation on the behavior of `MMo`. An example where a big type occurs because of polymorphic recursion (and not because of let-bindings) is presented at the end of this paper (Section 7.).

For simplicity, in this paper we present an algorithm that imposes a restriction on the set of typable programs with polymorphic recursion: polymorphic recursive definitions may occur only at the outermost level (in other words, an inner let-binding may not introduce polymorphic recursive definitions). The elimination of this restriction is not straightforward, but also not very significant in principle, requiring some extra ("bookkeeping") work that would deviate from our main interests.

Our work explores the idea, suggested by Jim [Jim, 1996], that *principal typing* is the key for efficiently solving the problem of type inference for mutually recursive definitions. The basic ingredient used in `MMo` is the computation of principal typings for any given typing problem, instead of simply the principal type. This is achieved

[1]An undesirable consequence, in Haskell, of the possibility of making a type more general by means of a type annotation is that the insertion of a type annotation in a program may change its semantics. An interesting example illustrating this situation has been posted in the Haskell mailing list by Lennart Augustsson [Lennart Augustsson, 2001].

essentially by allowing inferred typing contexts (in a formulation of the algorithm in terms of a type system, this would mean typing contexts that occur in the right-hand side of typing formulas) to have more than one assumption for the same variable.

We present a stepwise description of `MMo` in Haskell. This is based on Mark Jones's type inference algorithm for Haskell [Jones, 1999] — although all the main ideas constitute an adaptation of those in [Figueiredo and Camarão, 2002] towards obtaining a practical type inference algorithm — intended to provide a palatable description for readers, as well as for language designers and developers, familiar with Haskell.

We assume in the sequel a basic knowledge of Haskell and of the process of type inference [Mitchell, 1996]. Our prototype implementation includes also a (monadic) parser (based on Parsec [Leijen, 2003]) that supports a large subset of Haskell (e.g. the Haskell layout rule is not yet implemented). It is available, together with examples (including all examples given in this paper) at `http://www.dcc.ufmg.br/~camarao/MMo/MMo.tar.gz`.

2. Motivation

Recently, several applications of the use of nested data types have been presented. One example is a data type that represents perfectly balanced binary trees, proposed by Chris Okasaki [Okasaki, 1998], which can be declared as follows:

```
data Seq a = Nil | Cons a (Seq (a,a))
```

In this example the recursive component *Seq*(a,a) is different from the type *Seq a* being defined, characterizing this as a nested data type. Often, nested data types support algorithms more efficient than corresponding uniform versions. For example, function *len* below calculates the number of elements in a data structure of type *Seq a* of length n in time $O(log\, n)$.

```
-- len ::  Seq a -> Int
len Nil            = 0
len (Cons x s)     = 1 + 2 * (len s)

pair f (x,y)       = (f x, f y)

-- mapseq :: (a -> b) -> Seq a -> Seq b
mapseq f Nil       = Nil
mapseq f (Cons x xs) = Cons (f x) (mapseq (pair f) xs)
```

Figure 1: An example of a nested data type

Function *len* uses polymorphic recursion, since it receives as parameter a value of type *Seq a* and returns an integer, but calls itself with values of type *Seq*$(a,\ a)$. The *mapseq* function is also polymorphic recursive. Functions *len* and *mapseq* can be declared in Haskell 98 as long as their of types are explicitly annotated.

Another example that illustrates the potential significance of the use of nested data types has been presented in [Bird and Meertens, 1998], where a nested data type is used to represent expressions of the λ-calculus in the notation of *De Bruijn levels*.

In De Bruijn's notation, bound variables are represented by natural numbers. Number n represents the variable of the λ-abstraction that is nested *n* times inside other λ-abstractions; for example, $\lambda.\,\lambda.\,0(11)$ represents the lambda term $\lambda x.\,\lambda y.\,x(y\,y)$.

To manipulate terms containing free variables, a (so-called) *named context* is necessary, assigning De Bruijn numbers to free variables [Pierce, 2002]. For example: in context $\Gamma = \{w \mapsto 0, x \mapsto 1, y \mapsto 2\}$, $x(w\,y)$ is represented by $1(0\,2)$ and $\lambda x.\,\lambda y.\,x(w\,y)$ is represented by $\lambda.\,\lambda.\,0(2\,1)$, where free variable *w* is represented by its index added to the number of nested lambda abstractions inside which it occurs.

```
data Bind a = Zero | Succ a
data Term a = Var a | App (Term a, Term a) | Abs (Term (Bind a))

-- lift                 :: (Term a, a) -> Term (Bind a)
lift (Var y, x)         = if x == y then (Var Zero) else (Var (Succ y))
lift (App (u, v), x)    = App (lift (u, x), lift (v, x))
lift (Abs t, x)         = Abs (lift (t, Succ x))

abstract (t, x)         = Abs (lift (t, x))
reduce (Abs s, t)       = subst (s, t)

-- subst                :: (Term (Bind a), Term a) -> Term a
subst (Var Zero, t)     = t
subst (Var (Succ x), t) = Var x
subst (App (u, v), t)   = App (subst (u, t), subst (v, t))
subst (Abs s,t)         = Abs (subst (s, term Succ t))

-- term                 :: (a -> b) -> (Term a, Term b)
term f (Var x)          = Var (f x)
term f (App (u, v))     = App (term f u, term f v)
term f (Abs t)          = Abs (term (bind f) t)

bind f Zero             = Zero
bind f (Succ x)         = Succ (f x)
```

Figure 2: A nested data representation of De Bruijn levels

Elements of *Term a* can be free variables, applications or abstractions. The variable of the outermost λ-abstraction is represented by *Var Zero*, of the next λ-abstraction $Var(Succ\,Zero)$, and so on. Free variables are represented by $Var(Succ^n\ a)$, where n is equal to the number of nested λ-abstractions. With this representation, there is no need for using a named context to represent free variables.

Term a is a nested data type because the recursive component $Term(Bind\,a)$ that appears in its definition is different from the type being defined. Using function *abstract* defined on Figure 2, we can obtain the representation of $\lambda x.\,\lambda y.\,x(w\,y)$. That is,

```
abstract (abstract (App (Var 'x', App (Var 'w', Var 'y')), 'y'), 'x')
```

is equal to

```
Abs (Abs (App (Var Zero, App (Var (Succ (Succ 'w')), Var (Succ Zero)))))
```

Function *reduce* implements (λ-calculus) β-reduction. The other functions in Figure 2 are used in the implementation of *abstract* and *reduce*: *term* maps f over a *term*, *bind* maps f over elements of *Bind*, *subst* is used in *reduce* to update (decrease) levels of variables when leaving a lambda abstraction, and *lift* updates levels of variables of a term t for use in the representation $\lambda x.\, t$.

Functions *lift*, *subst* and *term* are also polymorphic recursive. They can be declared in Haskell 98, as long as their types are explicitly annotated. It'd be necessary that *a* and *Bind a* be declared as equality types.

3. Types

We start the description of the implementation by defining how types are represented. For simple types, we have (where *Id* is a synonym for *String*):

```
data Type = TVar Tyvar | TCon Tycon | TGen Int
          | TAp Type Type deriving Eq
data Tyvar = Tyvar Id [Int] deriving Eq
data Tycon = Tycon Id deriving Eq
```

The use of a list of integers in type variables is explained in Section 6. The following definitions illustrate the representation of predefined types:

```
tInt    = TCon (Tycon "Int");     tChar   = TCon (Tycon "Char");
tList   = TCon (Tycon "[]");      tString = TAp tList tChar;
tTuple2 = TCon (Tycon "(,)");     tArrow  = TCon (Tycon "(->)");
```

TGen Int is used for representing quantified type variables. This representation is appropriate, because quantified type variables are then easily not modified by substitutions (see next section) and because type equality need not consider equivalence up to renaming of quantified type variables (since they are integer numbers generated in a given order).

Types can also be quantified types (also called *type schemes*):

```
data Scheme = Forall Type deriving Eq

quantify       ::[Tyvar] -> Type -> Scheme
quantify vs t  = Forall (apply s t)
      where vs'  = [ v | v <- tv t, v `elem` vs ]
            s    = zip vs' (map TGen [0..])

toScheme       ::Type -> Scheme
toScheme t     = Forall t
```

4. Substitutions

Substitutions are *finite* mappings from type variables into simple types. Their finite domain makes a list of pairs a suitable representation, since the operation of computing the domain of a substitution can then be easily implemented (in contrast to the situation of representing substitutions by a functional type).

```
type Subst      = [(Tyvar, Type)]
domain          = map fst
nullSubst       = []
(+->)           ::  Tyvar -> Type -> Subst
a +-> t         = [(a, t)]
(@@)            ::  Subst -> Subst -> Subst
s1 @@ s2        = [ (u, apply s1 t) | (u,t) <- s2 ] ++ s1
```

Straightforward definitions are given for the identity (null) substitution, a "maps-to" operator (`+->`), and composition of substitutions (`@@`). The latter uses list concatenation, taking into account that an application of a substitution, represented by a list, considers only the first occurrence of a type variable in this list.

Functions *apply* and *tv*, for applying a substitution and for computing the set of free type variables, respectively, are overloaded to operate on quantified types, simple types or typing contexts. We include only the (more interesting) definition of *apply* for simple types, which uses the fact that *lookup* returns the type associated to the first occurrence of a type variable in the list representing a substitution:

```
instance Subs Type where
    apply s (TVar u) = case lookup u s of { Just t -> t; Nothing -> TVar u }
    apply s (TAp l r) = TAp (apply s l) (apply s r)
    apply s t         = t
```

Types t and t' unify if there exists a substitution S such that $S(t) = S(t')$. Its constructive (algorithmic) definition is straightforward and well-known, and is omitted. For simplicity, we use *unify*`::(`*t*`,`*t*`)->`*Subst* as returning a substitution or error, instead of *Maybe Subst* (the latter would enable issuing more meaningful error messages). Function *unify* is also overloaded to unify lists of types.

5. Typing Contexts and Principal Typings

A *principal typing* solution `(`*t*`,`*g*`)` of a typing problem `(`*e*`,`*g0*`)` is such that typing context *g* requires less and type t provides more than any other typing solution. context (*g0*) in a typing problem allows the specification of fixed assumptions, that is, assumptions for variables that are visible in the represented scope. A typing context is a list of assumptions, i.e. a list of pairs `(`*x*`,`*sc*`)`, where *x* is a variable and *sc* is a quantified type:

```
data Assump     = Id :>: (Kind_of_def, Type) deriving Eq
data Kind_of_def= LET | LAM deriving Eq
type TypCtx     = [Assump]
```

An assumption for a variable also includes information about the kind of its defining occurrence. We distinguish between let-bound (*LET*) and λ-bound (*LAM*) variables. The distinction is used to identify type variables that may not be quantified (which are those occurring in types of λ-bound variables).

An important characteristic of `MMo` is the way typing contexts are used, throughout, for the purpose of computing principal typings. The results of type inference functions are "contextualized": for example, function *tiExpr* (Section 6.) returns a contextualized simple type — i.e. an inferred type together with a corresponding minimal context required for its inference — and *tiBindGroup* (Section 6.) returns a contextualized typing context (i.e. a list of contextualized assumptions, of type *InfCtx*):

```
type Typing = (Type, TypCtx); type IdTyping = (Id , Typing);
type InfCtx = [IdTyping]
```

Function *tc* gives the principal typing solution of typing problems (*x*,*g*) (see Figure 3), where *TI* is the type inference monadic type constructor and variable *g* is used to denote typing contexts.

```
tc     ::  Id -> TypCtx -> TI Typing
tc i g  = if null found_sc then do t <- newTVar
                                   return (t, [i :>:(LET, toScheme t)])
          else do t <- freshInst $ head found_sc
                    return (t, [i :>:  (LET, toScheme t)])
          where found_sc = find i g
find   ::  Id -> TypCtx -> [Scheme]
find i g = [sc |(i' :>:  (kdo, sc)) <- g, i'==i ]
```

Figure 3: Principal typings for variables

Another important characteristic of MMo is the possibility of an inferred typing context to have more than one assumption for the same variable (input typing contexts, on the other hand, can be simple typing contexts, with just one type assumption for each variable). Consider, for example, the problem of inferring principal typings for expression *x x*, in a given typing context $g0$, where the definition of x might occur after the use of this expression. It is well-known that there is no principal *typing* for expression *x x* in the Damas-Milner system [Damas, 1984, Jim, 1996, Figueiredo and Camarão, 2001].[2] If $g0$ does not include any assumption for x, tc assigns a fresh type variable as the type of each occurrence of x in $x\,x$, say a and b. (Obtaining fresh type variables is the job of *newTVar* above, which uses a simple monad for updating the integer value corresponding to the last fresh type variable used.) As usual in type inference algorithms (cf. Section 6.), this will result in unifying a with $b \to a'$, where a' is another new fresh type variable. The typing returned for $x\,x$ will be then (a', [$x : b \to a', x : b$]).

Type a' can then be closed, by using function *close* (Figure 5), which works so as not to close any type variable in the type of a λ-bound assumption, in the given typing context. This yields principal typing ($\forall a.\, a$,[$x : b \to a', x : b$]) for typing problem ($x\,x$,$\emptyset$).

Operators (|+) for overriding a typing context with another in let-bindings, (|-|) for removing assumptions from a typing context after let-bindings, and (|-), which also removes assumptions from a typing context, but after λ-expressions, are defined in Figure 4. The latter requires checking that no variable occurring in a pattern at the left-hand side of a function definition occurs twice in the resulting typing context. For

[2]The reason is that the greatest type derivable for this expression, $\forall a.\, a$, can only be derived in a typing context with the assumption $x : \forall a.\, a$. Other typing solutions exist — for example, assumption $x : \forall a.\, a \to a$ can be used to derive $x\,x : \forall a.\, a \to a$, but, while using a typing context that requires less, yields a type that also provides less.

example, two assumptions for x are inserted in the resulting typing context for $x\,x$ (as explained above). We can think of this as meaning that x would need to have a polymorphic type, in order for $x\,x$ to be typable (and therefore $\backslash x.\,x\,x$ is detected as not type correct).

```
g0 |+ g = g ++ filter compl g0
    where compl (x:>:_) = not(x `elem` (dom g))

(|-) ::  TypCtx -> [Id] -> TypCtx
g |- xs = if length elems_xs <= length xs then non_elems_xs
          else error ("parameter used polymorphically")
    where (elems_xs,non_elems_xs) = partition( (x:>:_) -> x `elem` xs) g

(|-|) ::  TypCtx -> [Id] -> TypCtx
g |-| is = filter ( (i:>:_) -> i `notElem` is) g
```

Figure 4: Overriding and removing entries from typing contexts

6. Inferring Principal Pairs

tiProgram and *tiBindGroup* receive a typing context and a binding group and return a contextualized typing context, i.e. a list of contextualized types, one for each definition in the binding group. The data structures used are defined below:

```
type BindGroup       = ([AnnotType_bind], [InferType_bind])
type InferType_bind  = (Id, [Alt])
type AnnotType_bind  = (Id, Type, [Alt])
type Alt             = ([Pat], Expr)

tiProgram            ::  TypCtx -> BindGroup -> TypCtx
tiProgram g0 bg      = (map close) $ runTI $ tiBindGroup g0 bg
```

```
lambda_assump (i :>: (LAM,_))   = True
lambda_assump _                 = False

lambda_assumps = filter lambda_assump

close      ::  IdTyping -> Assump
close (i,(ti,gi)) = i:>:(LET, quantify (tv ti \\ tv (lambda_assumps gi)) ti)
```

Figure 5: Closing simple types

tiProgram calls the monadic deconstructor function (*runTI*), with the result of *tiBindGroup*. The definitions of monadic operations, used to generate fresh type variables (by *newTVar*) and, based on this, fresh instances of polymorphic types (by *freshInst*), as well as the treatment of patterns (of type *Pat*), follow Mark Jones's work [Jones, 1999, Section 10], and are omitted for brevity. However, as we will see later in this section, we will need another function for generating special fresh instances of types, named *supInst*.

Function *tiBindGroup* returns a contextualized typing context, which is a list of the contextualized types of all definitions in a binding group. The contextualized types are pairs `(`*ts*`,`*gs*`)`; *ts* is a list of so-called *inferred types* and *gs*, a list of *inferred typing contexts*, contains assumptions whose types are called *required types*. Function *close* is used to quantify the inferred types. The definitions of *tiBindGroup* and *close* are given in Figures 7 and 5, respectively. Inferred and required types are unified, by folding function *getTs* (Figure 9) over the initial contextualized simple types, given by *tiInferTyping*. This is explained in more detail below. Required types of each λ-bound variable are then unified, and the resulting substitution s is applied to each inferred type and typing context. Before returning the obtained contextualized typing context, it is checked whether annotated types are *correct*, meaning that they are *equal* (simply compared by using `==`) to the corresponding types inferred by considering g — with symbols with annotated types as assumptions (see Figure 7) — as the input typing context.

Contextualized types are inferred for definitions with and without type annotations, contained respectively in the lists *expl_bg* and *impl_bg*. *tiBindGroup* uses *tiInferTyping* to infer a list of (initial) contextualized simple types, one for each definition. The definition of *tiInferTyping* essentially calls *tiExpr* (see below) for computing contextualized simple types for each name defined in the binding group.

List *is_infTypings*, returned by *tiInferTyping*, consists of the names defined in a binding group (*xs*) and corresponding contextualized simple types `(`*ts*`,`*gs*`)`. Let's say *xs* is formed by $x_1, \ldots, x_n$, and similarly for *ts* and *gs*. The crucial job of function *getTs* (Figure 9) is to return all pairs `(`*t*`,`*t'*`)` such that i) *t'* occurs in an assumption $x_i : t'$ in some g_j ($j \in \{1, \ldots, n\}$); ii) t is obtained from t_i by renaming, to a fresh type variable, each type variable that does not occur free in the type of some λ-bound variable — in other words, by renaming (to a fresh type variable) each type variable in $tv(t_i)\backslash\backslash nqtvs_i$, where $nqtvs_i$ `=` *tv*`(`*lambda_assumps* g_i`)`.

This renaming is done by function *supInst* (used by *getTs* and defined in Figure 6). The renaming corresponds to the creation of new (fresh) variables occurring in formulas in the left-hand side of inequalities in the underlying semi-unification problem (SUP). This SUP is such that each inequality ($t_i \leq^i t'$) corresponds to a polymorphic use (with type t') of some defined variable (with inferred type t_i).

```
supInst ::  [Tyvar] -> Int -> Type -> TI Type
supInst vs n  (TAp l r) = do t1<-supInst vs n l; t2<-supInst vs n r; return $ TAp t1 t2
supInst    vs  n  (t@(TVar tv@(Tyvar v l)))
  | tv 'elem' vs   = return t
  | otherwise      = return $ TVar $ Tyvar v (n:l)
supInst    _  _   t  = return t
```

Figure 6: Generating indexes that register type variable dependencies

After these pairs `(`t`,`t'`)` are obtained, a substitution is computed by unification of types in each of these pairs. Then it is tested whether this substitution has (so-called) "circular dependencies" between type variables (see below), reporting an error if this happens; if not, the substitution can either modify inferred or required types, in which case the whole process is repeated, or not — in the latter case the process terminates,

```
tiBindGroup ::  TypCtx -> BindGroup -> TI InfCtx
tiBindGroup g0 (expl_bg,impl_bg) =
 do let g = g0 |+ [ v:>:(CW,sc) | (v,sc,alts) <- expl_bg ]
      is_infTypings<-tiInferTyping g (impl_bg++(map(\(v,_,alts)->(v,alts)) expl_bg))
      ins_infTypings' <- unify_inf_req is_infTypings
      checkAnnot expl_bg $ is_infTypings'

unify_inf_req::  InfCtx -> TI InfCtx
unify_inf_req isIinfTypings =
  do let { (is,infTypings) = unzip is_infTypings; (t_i,g_i) = unzip infTypings;
             all_g_i = concat g_i; lbtvs = tv $ lambda_assumps all_g_i }
       tsReq_tsInf <- foldM(getTs lbtvs (zip is t_i)) [] (zip [0..] all_g_i)
       (tsReq_tsInf1,infCtx) <- unif_app lbtvs (tsReq_tsInf,is,infTypings)
       return infCtx

unif_app::[Tyvar]->([(Type,Type)],[Id],[Typing])->([(Type,Type)],[Typing])
unif_app lbtvs (tsReq_tsInf,is,infTypings) =
  do let { s = unify $ unzip tsReq_tsInf; infTypings' = apply s infTypings }
       tsReq_tsInf' <- foldM (getTs lbtvs (zip is (map fst infTypings'))) []
                                (zip [0..] $ concat $ map snd infTypings')
       if stop s (map fst infTypings ++ (
                    map (\(_:>:(_,Forall t))->t) $ concat $ map snd infTypings))
         then return (tsReq_tsInf',zip is infTypings')
         else if circular_dep s then error(
                   "Cannot (semi-)unify inferred with required types\n")
              else unif_app lbtvs (tsReq_tsInf',is,infTypings')
stop s ts = null (domain s `intersect` tv ts)
```
Figure 7: Principal pairs of recursive definitions

giving (successfully) contextualized inferred types.

This process is performed by function *unif_app* (defined in Figure 7) over the list *tsReq_tsInf*, returned by *getTs*. The updating of an inferred type, by the application of a substitution, occurs because the use of a variable in the corresponding definition *requires less* than it could; in other words, this use of the variable requires a type that is more general than that given by its definition. The key idea that allows this process (in fact, the underlying semi-unification) to terminate is the detection of circular dependencies between type variables. What exactly is a circular dependency and how this dectection is carried out? Briefly, and informally, each new fresh variable α is created in this process with a representation that "remembers" all fresh variables β from which it has been originated (informally, we say "on which it depends"); a test of circular dependency is merely then a test of whether a substitution requires replacing α by a type in which some of these type variables β occur.

This representation is simply a sequence of integer indices — where distinct indexes correspond to distinct polymorphic uses (i.e. distinct is): creating a new fresh variable from a type variable α that occurs in t_i simply amounts to placing a new head (namely, i) in the sequence of indices of α. Function *circular_dep*, responsible for testing

```
circular_dep                              :: Subst -> Bool
circular_dep s                            = any (==True) $ map circ s

circ    (v,         TAp l r)              = circ (v,l) || circ (v,r)
circ    (Tyvar v l, TVar (Tyvar v' l'))   = v==v' && l `subStr` l'
circ    (_,         _)                    = False

subStr  []          _                     = True
subStr  l           (_:l')                = l == l' || l `subStr` l'
subStr  _           _                     = False
```

Figure 8: Testing circular dependencies between type variables

circular dependencies between type variables, is defined in Figure 8.

Function *stop* (Figure 7) determines when to stop the process of unifying inferred and required types. This occurs when no inferred or required type is updated.

Function *tiExpr*, called by *tiInferTyping*, computes principal typings for expressions. For reasons of space and focus, Figure 10 includes only the cases of variables, applications and let-bindings at the outermost level.

```
getTs vs i_ts pts (n,i:>:(_,sc@(Forall t))) =
-- getTs updates the list of pairs of types pts with pairs (t,t'), where
--   t' = supInst vs n t" for all i:t" ∈ i_ts, t'' ≠ t (modulo renaming of tvars)
-- vs indicate type variables that shall not be quantified
-- n denotes inequality index in the underlying SUP
--    corresponding to distinct uses of a defined variable

do let getT (i,sc) = case lookup i i_ts of
            Just t' -> let sc' = quantify (tv t' \\ vs) t' in
              if sc==sc' then return Nothing
              else do t'' <- supInst vs n t'; return (Just t'')
            Nothing -> return Nothing
   maybe_t <- getT (i,sc)
   case maybe_t of Just t' -> return ((t,t'):pts)
                   Nothing -> return pts
```

Figure 9: Unification of inferred and required types

7. Examples

As a simple example of type inference with polymorphic recursion, consider the definitions of function *len* and the constructors of data type*Seq a* in Figure 1. The value of *is_infTypings*, that defines a typing for each name in the binding group, is given by:

```
[(Nil, (Seq a, [])),
 (Cons, (b → Seq (b, b) → Seq b, [])),
 (len,(Seq c → Int, [(+):Int → Int → Int, len:Seq(c, c) → Int]))]
```

```
tiExpr :: TypCtx -> Expr -> TI Typing

tiExpr g (Var i)        = tc i g

tiExpr g0 (Ap e1 e2) =
  do (t1,g1) <- tiExpr g0 e1
     (t2,g2) <- tiExpr g0 e2
     a <- newTVar
     let { s0 = unify (t1, fn t2 a); g1' = apply s0 g1; g2' = apply s0 g2;
           pts = types_common_lambda_vars g1' g2';
           s1 = unify (unzip pts); s  = s1 @@ s0 }
     return (apply s a, apply s1 g1' `union` apply s1 g2')

tiExpr g0 (Let bg e) =
  do infCtx <- tiBindGroup g0 bg
     let { is = map fst infCtx; g = infCtx2TypCtx infCtx;
           gi = bigUnion $ map (snd.snd) infCtx; g0'= apply s g0;
           s = unify $ unzip $ types_common_lambda_vars g0 gi }
     (t,g') <- tiExpr (g0' |+ apply s g) e
     let s' = unify $ unzip $ types_common_lambda_vars g0' g'
         s'' = s'@@s
     return (apply s'' t, apply s'' (g'|-| is) `union`
                          apply s'' (lambda_assumps gi))
```

Figure 10: Principal typings for expressions

As a result of folding *getTs* over *is_infTyping*, the following pairs of types are produced: $[(Seq(c,\ c) \to Int, Seq\ c^0 \to Int)]$, where c^0 is a fresh type variable, created by *supInst*, from *c*). The application of the substitution obtained as a result of unifying the pairs of types given by getTs — namely, the identity substitution on all type variables but c^0, which is mapped to (c, c) — does not modify inferred or required types.

As a simple (but now contrived) example for which types are indeed modified by the unification of inferred with required types, consider:

$$\begin{aligned} h\,x &= (g\,x) + 1 \\ g\,x &= h\ (g\,x) \end{aligned}$$

In this case, *is_infTyping* is given by:

$$\begin{aligned} [(&h,\ (c \to Int,\ [(+){:}Int \to Int \to Int,\ g{:}c \to Int])), \\ (&g,\ (a \to b,\ [g{:}a \to d,\ h{:}\ \ d \to b]))] \end{aligned}$$

getTs returns the following pairs of types: $[(d \to b,\ c^0 \to Int),\ (a \to d,\ a^1 \to b^1),\ (c \to Int,\ a^2 \to b^2)]$. The unification of these pairs of types, performed by *unif_app*, causes type variable *b* to be replaced by *Int*. Since (inferred/required) types are modified by the application of this substitution, another call to *unif_app* follows. Pairs of inferred and required types are obtained from the contextualized simple types in:

$$\begin{aligned} [(&h,\ (c \to Int,\ [(+){:}Int \to Int \to Int,\ g{:}c \to Int])), \\ (&g,\ (a \to Int,\ [g{:}a \to Int,\ h{:}\ \ Int \to Int]))] \end{aligned}$$

getTs now returns: `[(`$a \rightarrow$ *Int*`,` $a^1 \rightarrow$ *Int*`),` `(`$c \rightarrow$ *Int*`,` $a^1 \rightarrow$ *Int*`)]`. Applying the substitution obtained by unifying these pairs of types modifies neither inferred nor required types (modulo renaming of type variables), and *unif_app* is thus completed.

The simplest case in which there is a circular dependency between type variables appears in the types of inferred and required types is that of a "direct dependency", which occurs for example in the case of the definition f x `=` f. Inferred and required types for this definition are, respectively, $a \rightarrow b$ and b. A call to *getTs* returns, then, the list `[(`b`,` $a^0 \rightarrow b^0$`)]`. The unification of pairs of types in this list gives a substitution that maps b to $a^0 \rightarrow b^0$. A call to *circular_dep*, with this substitution as parameter, returns *True*, originating then a type error message.

Examples of indirect circular dependencies may be constructed — based on the expression used in [Henglein, 1993] to show that typability in the Milner-Mycroft calculus is reducible to semi-unification — by varying the value of n in the following pseudo-Haskell example (where we use $(v_1, \ldots, v_n)$.**i** to denote v_i, for $i = 1, \ldots, n$, corresponding to Haskell functions *fst*, *snd* etc.):

```
k x y = x
f x1 x2 ... xn = k (\x -> x x1 x1, ..., \x -> x xn xn)
      (\ y1 y2 ··· yn -> (f y1 y2 ... yn).1       == x2, ...,
       \ y1 y2 ··· yn -> (f y1 y2 ... yn).(n − 1) == xn,
       \ y1 y2 ··· yn -> (f y1 y2 ... yn).n       == x1)
```

The time taken by *circular_dep* to detect the circular dependency (issuing then a type error) grows exponentially with an increase of n. A similar well-typed example can be obtained by changing the definition so that f has $n+1$ parameters instead to n, change calls accordingly and use x_{n+1} instead of x_1 in the last line above. Then f becomes typable, and has a big type; for $n = 2$ (i.e. f has 3 parameters) this type can be written as:

$$\begin{aligned}&\forall a, b, c, d, e, f, g, h.\ a \rightarrow ((b \rightarrow b \rightarrow c) \rightarrow c) \rightarrow\\ &\qquad\qquad ((((d \rightarrow d \rightarrow e) \rightarrow e) \rightarrow ((d \rightarrow d \rightarrow e) \rightarrow e) \rightarrow f) \rightarrow f) \rightarrow\\ &\quad ((a \rightarrow a \rightarrow g) \rightarrow g, (((b \rightarrow b \rightarrow c) \rightarrow c) \rightarrow ((b \rightarrow b \rightarrow c) \rightarrow c) \rightarrow h) \rightarrow h)\end{aligned}$$

8. Conclusion

We have presented an algorithm (called `MMo`) for typing polymorphic recursive definitions, with the aim of providing a readable description for programmers, language designers and developers familiar with Haskell. `MMo` is an adaptation into Haskell of an algorithm that can be straightforwardly obtained from the rewriting system RSUP [Figueiredo and Camarão, 2002]. Its correctness and termination follow from that of RSUP, which have been proved in that paper. The algorithm allows type inference to be simplified, by eliminating the need to examine the call graph of a program in order to determine an order in which to infer types. The worst case time complexity of `MMo` is exponential, as occurs in the case of type inference for core ML. However, these worst case examples simply do not occur in practice: both in the case of core ML and with polymorphic recursion, exponentiality requires the existence of big types (for which the number of type variables is exponentially larger than the size of the input expression), and these do not represent useful abstractions. For small types, the time complexity of the algorithm is polynomial. The algorithm is expected thus to behave well in practice.

References

Bird, R. and Meertens, L. (1998). Nested Datatypes. In *Proc. 4th Intn'l Conf. Math. Prog. Construction, LNCS 1422*, pages 52–67.

Connelly, R. H. and Morris, F. L. (1995). A generalisation of the trie data structure. *Mathematical Structures in Computer Science*, 6(1):1–28.

Damas, L. (1984). *Type Assignment in Prog. Languages*. PhD thesis, Univ. Edinburgh.

Damas, L., Milner, R. (1982). Principal type schemes for func. progs. *POPL'82,* 207–212.

Figueiredo, L. and Camarão, C. (2001). Principal Typing and Mutual Recursion. In *Proc. of the Intn'l Workshop on Functional and Logic Programming*, pages 157–170.

Figueiredo, L. and Camarão, C. (2002). Semi-unifiability is decidable. *Submitted*.

Henderson, F. et al. (2003). The Mercury Project. http://www.cs.mu.oz.au/research/mercury/.

Henglein, F. (1993). Type inference w/ polymorphic recursion. *TOPLAS*, 15(2):253–289.

Jim, T. (1996). What are principal typings and what are they good for? *POPL'96*, 42–53.

Jones, M. (1999). Typing Haskell in Haskell. In *Proc. 1999 Haskell Workshop.*

Jones, S. P. et al. (1998). The Haskell 98 Report. http://haskell.org/definition.

Kanellakis, P. and Mitchell, J. (1989). Polymorphic unification and ML typing. In *2^{nd} European Symposium on Programming*.

Kanellakis, P., Mairson, H. and Mitchell, J. (1991). Unification and ML type reconstruction. In *Computational Logic: Essays in Honor of Alan Robinson*, pages 444–478.

Kfoury, A. J., Tiuryn, J., and Urzyczyn, P. (1993). Type reconstruction in the presence of polymorphic recursion. *ACM TOPLAS*, 15(2):290–311.

Leijen, D. (2003). Parsec. http://www.cs.ruu.nl/˜daan/parsec.html.

Lennart Augustsson (2001). Message to the Haskell mailing list, May 3, 2001, available from http://www.mail-archive.com/haskell@haskell.org/.

Mairson, H. (1990). Deciding ML typability is complete for deterministic exponential time. In *Proc. ACM POPL'90*.

Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375.

Milner, R., Tofte, M., and Harper, R. (1989). *The Definition of Standard ML*. MIT Press.

Mitchell, J. (1996). *Foundations for Programming Languages*. MIT Press.

Okasaki, C. (1997). Catenable double ended queues. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 66–74.

Okasaki, C. (1998). *Purely Functional Data Structures*. Cambridge University Press.

Pierce, B. (2002). *Types and Programming Languages*. MIT Press.

Thompson, S. (1999). *Haskell: The Craft of Functional Programming*. Addison-Wesley.