Polymorphic Type Checking by Interpretation of Code

Stefan Kahrs*
University of Edinburgh
Laboratory for Foundations of Computer Science
King's Buildings, EH9 3JZ
email: smk@dcs.ed.ac.uk

Abstract

The type system of most modern functional programming languages is based on Milner's polymorphism. A compiler or interpreter usually checks (or infers) the types of functions and other values by directly inspecting the source code of a program.

Here, another approach is taken: The program is first translated into code for a stack machine and then a non-standard interpreter applied to this code checks (or infers) the type of the corresponding values. This can be seen as an abstract interpretation of the code of the program.

1 Introduction

In the early days of Functional Programming in the 1960's, functional programming languages did not have any proper concept of *type*; they (Lisp, ISWIM [9]) were typeless. Classical, monomorphic type-systems are restrictive in the sense that they only support the solution of concrete problems, but not problem schemes. But it is characteristic for the style of Functional Programming to solve problems in an abstract way, and thus a monomorphic type system is not really appropriate for it.

Strachey in 1967 [13] was probably the first to consider more general and abstract type systems, fitting to corresponding general and abstract ways of expressing algorithms. He invented the notion of polymorphism, distinguishing *generic* and *ad hoc* polymorphism. Today, "polymorphism" refers to Strachey's generic polymorphism.

The type systems of most wide-spread (typed) functional programming languages (Haskell, Hope, Miranda[1], SML) are based on Milner's polymorphic types [10] for functional programs. This polymorphism is a proper subsystem of the type system of second-order typed λ-calculus of Girard [3] and Reynolds [12]. An important difference is that ML types can be inferred from the program, while type-inference for second-order typed λ-calculus is not decidable[2].

The idea of polymorphism is to avoid writing the same function several times for types that are different but have the same structure. For example, the definition of a general *list* type and an associated *length* function in Miranda might look like this:

```
list * ::= NIL | CONS * (list *)
length NIL = 0
length (CONS x xs) = 1+length xs
```

The above type definition is parameterised (the asterisk) by the component type. The important point is that the definition does not make any use of the list components, so they could be of any type. `list char` is the type *list of characters* and `list num` the type *list of numbers*. The constructors `NIL` and `CONS` can be used for both types and the function `length` as well. It is not necessary to write an instantiation (like a generic instantiation in Ada[3]) to use the specialised forms of lists.

This is the user's point of view of polymorphism.

The implementor's point of view is surprisingly similar. It is not necessary to produce different object code for the function `length` for different instantiations of type `list *`. The reason is simple: If the function

*The research reported here was partially supported by SERC grant GR/E 78463.

[1] Miranda is a trademark of Software Research Limited.

[2] In a certain sense, type-inference for ML types is also not decidable, because semi-unification is undecidable, see [5, 4]. But it is well-behaved: there are decision procedures that *almost always* find the most general ML type, and the semi-unification algorithm can be supplied with an occur-check that *almost always* guarantees termination.

[3] Ada is a trademark of the US DoD.

is polymorphic, then it does not *access* a certain component of a data structure. But if the function does not access it, why should the code for the function do so? Eventually, if the code does not access it, it is as least as polymorphic as the original function and can be used for all instantiations.

Hence, if the object code of a polymorphic function reflects its polymorphism, then type checking of the code would yield the same result as type checking of the source. But how do we type-check the code?

2 Type Functions

Each function can be seen as a pair of functions: one function operates on values, the other on types. On the type `type`, the function `length` could be defined as follows:

```
length :: type -> type
length (LIST x) = NUM
```

Here, `NUM` and `LIST` are constructors of the type `type` and `list` is a partial function.

Type-inference for `length` means to compute this function. If all functions come in pairs, i.e. as functions on values and functions on types, then we can compute the type of an expression `f t` by computing the types for `f` and `t` and applying the type function corresponding to the type of `f` to the type of `t`. Therefore, application of functions corresponds to application of type functions. So, the idea is to compute the type of an expression by evaluating it in a different environment where all constructors are bound to their corresponding type functions and all basic values (literals) to their corresponding types.

It is not quite as simple as that, because constructors of some type `t` are not constructors of type `type`, i.e. the left-hand sides have to be treated differently. Also, there is currying — we might use `length` without argument:

```
length :: type
length = ABST (LIST x, NUM)
```

This version shows us a different problem: the `x` on the right-hand side is free, it is a placeholder for an arbitrary `type`. Operationally, this means that rewriting has to be replaced by narrowing, when we apply rules to (type) expressions.

These are the underlying ideas of this paper.

To exploit this for practical applications we have to produce the code *before* type checking, i.e. it cannot depend on the result of type checking and even more there has to be some code for ill-typed programs able to reproduce ill-typedness when used by the type checking interpretation of the code.

So we need an (abstract machine) code for an untyped version of the language we want to deal with, likely a code for untyped λ-calculus, i.e. something similar to the SECD machine [8] for λ-calculus, or a corresponding machine for term rewriting. We choose the latter, merely because term rewriting appears a bit closer to functional programming with pattern matching. The choice is not essential.

The idea of type checking by abstract interpretation of code came into my mind when I wanted to add a type checker to an already existing compiler for a typeless variant of Miranda. The compiler consisted of three main phases:

- parsing,
- pre-compilation to an applicative CTRS, see [7]; the methods used are fairly standard, e.g. lambda-lifting, compilation of list comprehensions etc., see [11, 2],
- compilation of the CTRS to abstract code for a stack machine.

The abstract machine was initially designed to execute (applicative) CTRSs under variable strategies, and the first two steps were added later. Then, it was slightly modified to fulfill the needs of a *real life* functional programming language. It was certainly not designed for the task of type checking. To my own surprise, it was neither necessary to change the instruction set of the abstract machine nor any of the compilation steps producing the code to use it for the purpose of type checking and type inference.

It would be probably boring, but certainly misleading to give the type inference algorithm on the *original* (C implementation) data structure for terms (and types) and to include the full instruction set of the *original* abstract machine. Both are too detailed and both are results of many further design decisions. Instead, we use for instructions and terms idealised representations, which are just strong enough to show the idea. For the matter of readability we use Miranda as *presentation* language.

3 The stack machine

First, we give a short description of term structure and the abstract machine.

```
term ::=
        Symbol [char] |
        Apply term term
```

The type `term` refers to ground terms of applicative TRSs, i.e. a `Symbol` can be a constructor as well as a function symbol. Variables do not need to be represented, because the stack machine overtakes their task.

The representation might look oversimplified, because e.g. a lazy interpreter of the stack machine needs to know whether a term is in *whnf* (weak head normal form, see [11]) or not, but adding special representations for constructor application, predefined functions, built-in types, etc. does not change anything principal to the method.

Furthermore, an implementation type for terms would usually (not necessarily for Miranda) contain some kind of closure operator to represent λ-abstractions. Here it would be applied to an instruction sequence equipped with an environment for the global identifiers used by the abstraction. For the dynamic semantics, it is safe to replace λ-abstractions by global combinators (which makes closures superfluous), but this might not be type-safe under all circumstances.

```
instruction ::=
        Isappl | Right | Set | Eq term |
        Push term | Pushi num | Pop | Fail
        Ifnot num | Label num | Goto num
```

The instruction set reflects the representation of terms, i.e. more primitive (implementation-internal) connectives than `Apply` for terms would require corresponding instructions for their construction and destruction. In the last line, the instructions for *if-then-else* are not essential for the approach; they are included here to show how it works in the presence of imperative-style instructions.

Evaluating terms by using this abstract machine works roughly as follows: Each term rewrite rule is represented by a list of instructions. Applying these instructions to a term (at its root) yields either the empty list (rule not applicable), or a singleton list of terms containing the result of the application. This view is independent from the choice of a one-step strategy (concerning strategies see [1] chapter 13, or [6]), because the search for a redex has to be done elsewhere. For many-step strategies such as *lazy evaluation* this is not quite true. But strategies do not affect the type anyway.

Not all elements of the type `[instruction]` are proper instruction sequences, e.g. a `Pop` or a `Right` cannot be the first instruction, all jumps are forwards, etc. A more precise characterisation of correct instruction sequences is given by the following context-free grammar (ignoring instruction arguments):

instruction-sequence	::=	match stack
match	::=	`Eq` | `Set` match' | match'
match'	::=	`Isappl` match `Right` match | ϵ
stack	::=	`Push` | `Pushi` | stack stack `Pop` | stack if
if	::=	`Ifnot` stack `Goto` `Label` stack `Label`

There are further non-context-free conditions on instruction sequences. The two labels in an *if* have numbers corresponding to the numbers of the `Ifnot` (first label) and the `Goto` instruction (second label). So the *if* always forms an *ifthen-else-fi* and the (conditional) jumps are forward jumps. Each `Pushi n` has to be preceded by at least `n+1` `Set` instructions. The usual purpose of the `Set` instruction is to store the values for the pattern variables, a following non-empty *match'* means that the value is stored for a non-variable subpattern.

The following example defines the functional `filter` for our previously defined lists and gives its corresponding instruction sequences:

```
filter p NIL = NIL
filter p (CONS x xs)
    = CONS x (filter p xs), if p x
    = filter p xs, otherwise

filter1 = [Isappl, Isappl, Eq (Symbol "filter"), Right, Right,
           Eq (Symbol "NIL"), Push (Symbol "NIL")]
```

```
filter2 = [Isappl, Set, Isappl, Eq (Symbol "filter"), Right, Set,
           Right, Isappl, Set, Isappl, Eq (Symbol "CONS"), Right,
           Set, Right, Set, Pushi 1, Pushi 3, Pop,
           Ifnot 0, Pushi 2, Pushi 0, Pushi 4, Pop, Pop, Goto 1,
           Label 0, Pushi 0, Pushi 4, Pop, Label 1]
```

`filter1/2` is the code for the first/second part of the definition of `filter`.

Notice the distinction made between constructors (`NIL`, `CONS`), identifiers defined by values (`filter`) and arguments (`p`, `xs`, etc.). Their treatment is similar as in term rewriting, i.e. value identifiers and constructors are considered as *symbols*, arguments as *variables*. In an efficient implementation one may want to represent value identifiers and constructors differently, but this does not necessarily affect the abstract code. It is somewhat easier to produce such code for Miranda or Haskell than for SML or Hope, because the former *lexically* distinguish constructors from other identifiers, while the latter do not. Hence, in Miranda or Haskell the code could be produced while completely *ignoring* type definitions.

The example — and the grammar for instruction sequences — might give an *idea* how the code would be executed, but better would be a *definition*:

```
interpret rules t = take 1 [re | cs <- rules;
                                 re <- run rules t cs [] [] []]

run r t [] p e stack = stack
run r (Apply a b) (Isappl:cs) pat e s = run r a cs (b:pat) e s
run r (Symbol str) (Isappl:cs) pat e s = []
run r t (Right:cs) (u:pat) e s = run r u cs pat e s
run r t (Set:cs) pat env s = run r t cs pat (t:env) s
run r t (Eq u:cs) pat env s
        = run r t cs pat env s, if t = u
        = [], otherwise
run r t (Push u:cs) pat env stack = run r t cs pat env (u:stack)
run r t (Pushi n:cs) pat env st = run r t cs pat env (get n env:st)
run r t (Pop:cs) pat env (x:y:st) = run r t cs pat env (Apply y x:st)
run r t (Fail:cs) p e s = []
run r t (Label l:cs) p e s = run r t cs p e s
run r t (Goto x:cs) p e s = run r t (dropwhile (~= Label x) cs) p e s
run rules t (Ifnot x:cs) p e (b:s)
        = run rules t cs p e s, if eval rules b
        = run rules t (dropwhile (~= Label x) cs) p e s, otherwise

get n ls = reverse ls ! n
```

The function `interpret` tries to apply one of the `rules` to the term `t` at its root position. Each rule is given by its instruction sequence. The selected rule (instruction sequence) is executed by the function `run` which uses three further stacks: `pat` are the remaining terms to be matched, `env` is the environment computed by the matching so far, and `stack` is the stack for creating the instance of the right-hand side of the definition.

A couple of remarks to the interpretation if-then-else instructions: we assume that `Goto` is always a forward jump, thus it is possible to restrict the search for the label to the remaining commands. For the interpretation of `Ifnot` we have used an obscure function `eval` which is supposed to fully evaluate a term w.r.t. a set of rules and then to convert the result term into a `bool`. The definition of such a function depends on the chosen strategy, we omit it here.

Notice that at the end of the instruction sequence evaluation (first equation of `run`), the stack is a singleton list, according to the grammatic restrictions for instruction sequences.

4 Types for Terms

The alternative interpretation of the code on types has to work on an appropriate representation of types and type environments.

Type environments are finite mappings from symbols to types. The abstract machine code contains symbols, because the `Push` and the `Eq` instruction have terms as arguments and each (finite) term contains symbols, according to the definition above. To get the type from the code, we need the types of these symbols. For the sake of simplicity, we moreover assume the term arguments of these instructions *to always be* symbols.

This can safely be done, because an instruction sequence of the form `s++[Push(Apply f a)]++t` is equivalent to an instruction sequence `s++[Push f,Push a,Pop]++t` w.r.t. to the definition of `run`. Similarly, we have for `s++[Eq(Apply f a)]++t` the equivalent `s++[Isappl,Eq f,Right,Eq a]++t`, provided `t` is of a certain form, e.g. if it starts with a `Right` or if it only contains stack instructions (`Push`, `Pushi`, `Pop`). But this is guaranteed by the given grammar for instruction sequences. The assumption is a simplification, because a term `(Apply f a)` can be wrongly typed on its own, while a symbol cannot be.

4.1 Type Structure

In type systems based on polymorphic typed λ-calculus, a type τ can be either a type variable α, or $(C\tau_1 ... \tau_n)$, where C is a n-ary type constructor and the τ_i are types, or $\forall\alpha.\tau'$, where α is a type variable and τ' is a type. However, not all types formed this way occur in the languages treated here, because none of it has a notion of binding type variables. In particular, the type of a function argument is quantifier-free and the type of each global identifier is of the form $\forall\alpha_1 ... \forall\alpha_n.\tau$ where τ is quantifier-free and the α_i are exactly the free type variables occurring in it.

For this reason we do not really need quantifiers in the representation for types, a first-order term structure serves well:

```
typerep ::=
        Tvar num |
        Tcon [char] [typerep]
```

A term `Tvar n` is a type variable with *index* `n` meaning the n-th type variable. A function type $\alpha \rightarrow \beta$ is represented as `Tcon "->" [`α'`,`β'`]`, where α' and β' are the representations of α and β, respectively.

Quantification can be treated on the level of type environments, for example as follows:

```
typeassoc == [(term,typerep)]
typeenv == (typeassoc,typeassoc,num)
```

A type association `typeassoc` is a finite mapping from terms (symbols) to types, represented here as an association list. In a type environment `typeenv` we distinguish between a global and a local type association, where the global association implicitly binds the type variables (the local does not). The third component of a type environment is in fact redundant, it is an upper bound for the numbers not occurring as a variable index in the local environment. This is useful information for creating *fresh* type variables, i.e. type variables not occurring in a certain environment.

This structure is not quite appropriate for local operations and values, because they are usually monomorphic in some variables and polymorphic in some others. We assume here that all local values have been λ-lifted.

4.2 Type Interpretation

An alternative interpretation of code takes a type environment and an instruction sequence and produces a new type environment. As in standard interpretation, a failure can occur: in standard interpretation it is non-applicability of a rule, here it is just a type error. Because an appropriate reporting of the type error is the most important task of a type checker, we want somewhat more explicit messages than "there is a type error in your program", so we supply the failure with an error message. This does not mean that we are producing henceforth the most helpful error messages — this would require more case distinctions. We just want to hint at the treatment of errors.

```
attempt * ::= Success * | Failure [char]
fails (Failure f) = True
fails (Success s) = False
type_interpret :: typeenv -> [instruction] -> attempt typeenv
```

The function `type_interpret` is defined here in a similar way as `interpret` above, but the auxiliary operations are a bit more complicated, because they work on types, type environments and error messages rather than just on terms. The main auxiliary function is `typerun`. It corresponds to `run`, because both traverse the instruction sequence and both work on similar stacks. On success, `typerun` produces two results, a new type environment and the type of the right-hand side.

```
type_interpret (gf,lf,n) cs
      = tr, if fails tr
      = Success (subst_env sub (gf',lf',n')), if b
      = nouni arg res, otherwise
        where
        tr = typerun (gf,(Symbol "",nv):lf,n+1) nv cs [] [] []
        Success (res,(gf',(lhs,arg):lf',n')) = tr
        (b,sub) = unify arg res
        nv = Tvar n
nouni :: typerep -> typerep -> attempt *
nouni t u = Failure ("cannot unify " ++ show t ++ " with " ++ show u)

unify :: typerep -> typerep -> (bool,[(num,typerep)])
typerun :: typeenv -> typerep -> [instruction] -> [typerep] ->
           [typerep] -> [typerep] -> attempt (typerep,typeenv)
```

If `typerun` *fails* (first equation), then there was a type error detected by interpreting the code. If not, it produces the type for the right-hand side `res` and an updated type environment. The updated environment includes the type of the left-hand side `arg`. For a function definition (equation), left-hand and right-hand side need to have the same type. If their types are unifiable (`b`, second equation), the result of `type_interpret` is the application of `sub`, the *mgu* (most general unifier) of `res` and `arg`, to the updated type environment. If not, `type_interpret` fails with a corresponding error message.

For unification we use ordinary first-order unification; a definition is in the appendix. For several kinds of substitution arguments there are several functions for substitution application (`subst`, `subst_env`, `subst_all`), also in the appendix.

The first argument of `typerun` is the type environment, associating types to symbols. It is analogous to the first argument of `run`, because the rewrite rules give the symbols their operational meaning, though in a rather indirect way. At the beginning the type environment used by `typerun` is the same environment as for `type_interpret` enriched by associating the empty symbol (`Symbol ""`) with a fresh type. The empty symbol is assumed not to occur in the instruction sequence, it is a place-holder for the left-hand side of the original definition. Thus, the type associated to the empty symbol is the type of the left-hand side.

4.3 Type Checking an Instruction Sequence

Amongst the auxiliary functions for the code's alternative interpretation, the most interesting one is `typerun`, because it directly interprets the code. We split the definition of `typerun` into several parts because of its length and the need for some explanation.

```
typerun tenv t [] p e [top] = Success (top,tenv)
typerun tenv t (Right:cs) (p:pat) e s = typerun tenv p cs pat e s
typerun tenv t (Set:cs) p env s = typerun tenv t cs p (t:env) s
typerun (gf,lf,n) t (Isappl:cs) pat e s
      = typerun (gf,lf,n+1) (Tcon "->" [nv,t]) cs (nv:pat) e s
        where nv = Tvar n
```

The *type* of the right-hand side of a definition is computed on the stack, the last argument of `typerun`, analogously to producing the *value* of the (instance of the) right-hand side in `run`. After interpreting the entire instruction sequence (third argument is `[]`), we know that the rule did not include a type error and that the stack contains exactly one type — the type of the right-hand side.

The instructions `Right` and `Set` are interpreted similarly as for `run`.

The instruction `Isappl` in the standard interpretation checks whether a term has the form `(f a)`, continuing matching on `f`; finally, matching on `a` is invoked by a corresponding `Right`. For type-inference, this means the following: the second argument of `typerun`, `t`, is the type of `(f a)` and it can be any type; because matching continues with `f`, it has to continue with the type of `f` which is some function type ($\alpha \rightarrow$ `t`) (above: `Tcon "->" [nv,t]`); when matching of `a` is invoked by a `Right` instruction, we need the type of `a` (which is α), so the corresponding type variable `nv` is pushed onto the pattern-stack.

```
typerun tenv t (Eq u:cs) pat env s
      = typerun tenv' t cs pat' env' s, if b
      = nouni t t', otherwise
```

```
          where
          (tenv',[pat',env']) = subst_all sub ((gf,lf,n'),[pat,env])
          (gf,lf,n) = tenv
          (t',n') = gettype tenv u
          (b,sub) = unify t t'
```

The Eq instruction is an equality test (of terms) in the standard interpretation, here it becomes a test for the equality (unifiability) of types. The standard interpretation of `Eq u` is to check whether the current *term* is *equal* to the term `u`, here it becomes to check whether the current *type* is *unifiable* with the type of `u`. We get the type of `u` by `(gettype tenv u)`. Because we have assumed `u` to be a symbol, it cannot contain a type error; but it can be polymorphic and in that case the type variable counter has to be increased (`n'`), because we get a copy of a polymorphic type by replacing the bound variables by fresh variables.

If unification succeeds (`b`), we have to apply the *mgu* `sub` to the type environment, the environment and the pattern stack (arguments 1, 4 and 5). According to the restriction (grammar) for valid instruction sequences, it is not necessary to apply `sub` to the stack — because it is still empty, nor to the current type — because it is thrown away anyway. Notice that in an imperative implementation all these substitution applications (done by `subst_all`) can be implicitly performed as a side-effect of the unification.

```
typerun tenv t (Push u:cs) p env stack
      = typerun (gf,lf,n') t cs p env (t':stack)
        where
        (t',n') = gettype tenv u
        (gf,lf,n) = tenv
typerun tenv t (Pushi n:cs) p env stack
      = typerun tenv t cs p env (get n env:stack)
```

The instruction `Push` is slightly more complicated to interpret than in the standard way. This is partly due to the additional indirection of taking the type of `u` rather than `u` itself, but this type may also be polymorphic, i.e. the fresh-variable-counter is affected.

`Pushi` has the same interpretation — formal parameters are never polymorphic.

```
typerun tenv t (Pop:cs) p env (x:y:stack)
      = typerun tenv' t cs p env' stack', if b
      = nouni x' y, otherwise
        where
        x' = Tcon "->" [x,nv]
        nv = Tvar n
        (b,sub) = unify x' y
        (gf,lf,n) = tenv
        (tenv',[env',stack']) =
            subst_all sub ((gf,lf,n+1),[env,nv:stack])
```

In the standard interpretation, `Pop` creates an application of the two top elements from the stack. Here correspondingly, the two top elements of the stack have to be a function type (`y`) and its domain type (`x`) and they are replaced by the codomain of the function type. A type error can occur in two ways: either `y` is not a function type (and not a type variable), or the domain of `y` is not unifiable with `x`. Above, both errors are treated in the same way using a single unification, but to produce helpful error messages they *should* be treated separately.

Finally for the *if-then-else* instructions we get:

```
typerun (gf,lf,n) t (Fail:cs) p env stack
      = typerun (gf,lf,n+1) t cs p env (Tvar n:stack)
typerun tenv t (Ifnot n:cs) p env (c:stack)
      = typerun tenv' t cs p env' stack', if b
      = Failure ("expected type bool, got " ++ show c), otherwise
        where
        (b,sub) = unify (Tcon "bool" []) c
        (tenv',[env',stack']) = subst_all sub (tenv,[env,stack])
typerun tenv t (Goto l1: Label l2: cs) p env (th:stack)
      = typerun tenv t cs p (th:env) stack
typerun tenv t (Label l: cs) p (th:env) stack
```

```
= typerun tenv' t cs p env' stack', if b
= nouni th (hd stack), otherwise
  where
  (b,sub) = unify th (hd stack)
  (tenv',[env',stack']) = subst_all sub (tenv,[env,stack])
```

While the standard interpretation of the code only assumes that (conditional) jumps go forwards, we additionally exploit their *if-then-else* structure here fully — for example, we ignore the label identifiers. A `Goto` occurs at the end of the *then*-part, hence the type on top of the stack is its type. We push it upon the environment and continue with the *else*-part. When we find a `Label` (not immediately after a `Goto`), it is the end of an *if-then-else*-part and the types of its *then*- and *else*-part are on top of the environment and on top of the stack, respectively.

An alternative interpretation with less assumptions (for a more general code generator) is possible, but rather awkward. Easy to handle are labels and gotos which could keep their meaning in the type interpretation, but conditional jumps are expensive. For the type interpretation of `(Ifnot n:cs)` we would need to execute `typerun` twice, once with code `cs` and once with code `(Goto n:cs)` and unify the resulting types and type environments afterwards. For an imperative implementation it is even worse, because it would have to *copy* the stacks involved for the second run.

4.4 Instances of a Polymorphic Type

To get a type from the environment, we have to distinguish types with bound variables and types with free variables; mixtures do not occur (because we have assumed λ-lifting of local values):

```
gettype (gf,lf,n) s
    = (hd ls,n), if ls!=[]
    = (copytype n . hd . lookup s) gf, otherwise
      where ls = lookup s lf
copytype n
    = cp
      where
      cp (Tvar m) = (Tvar (m+n),m+n+1)
      cp (Tcon s ts) =
         (Tcon s (map fst rec), max (n:map snd rec))
         where rec = map cp ts
```

The function `copytype` creates an instance of a polymorphic type by replacing all type variables by fresh ones. We assume that for every call of `(gettype tenv s)` the term `s` occurs somewhere in the type environment, i.e. checks for non-declared identifiers, dependency analysis etc. have already taken place. `lookup` is defined in the appendix.

5 Problems

Not all features of functional programming languages or their compilers fit very well with our approach. Typical problems occur in the following cases:

- overloading
- type assertions
- preserving polymorphism
- irrefutable patterns
- environments as parameters

Most of these items are related to a certain interaction between the types on the one hand, and expressions and their evaluations on the other. Overloading requires a type-check before evaluation, type assertions can appear in expressions and the feature of irrefutable patterns changes the pattern matching semantics for certain types. So far, we haven taken for granted that compilation preserves the polymorphism of a program, but this is not always the case; irrefutable patterns are one source of type change.

5.1 Overloading

A function `f` is overloaded if there are at least two function definitions with the name `f` working on different types (argument and/or result types). Typically, overloading is resolved at compile time, i.e. it has to be found out at compile time, which of the `f` functions has to be taken at a certain occurrence of the name.

Overloading (*ad hoc* polymorphism) does not fit perfectly with our approach, because its idea is to exploit the result of the type checker to resolve overloading, i.e. to replace an ambiguous identifier by a non-ambiguous one that fits the type context — hence the code is type dependent. In this sense, the code we base our type-check on would necessarily be ambiguous and not yet executable.

Most typed functional languages support some sort of overloading. Haskell's type classes are a rather general form, they can be seen as *constrained type variables.* The unification of a constrained type variable with a type expression τ requires a check whether τ satisfies the constraint and the addition of inferred constraints (not necessarily the same, and not necessarily one at all) to the type variables in τ. The only necessary changes to support this kind of *ad hoc* polymorphism seem to be the addition of constraint unification and a way to update the code, if constrained variables get instantiated. Miranda's restricted overloading (only for parsing, unparsing) can be seen as a special case of type classes, similarly the equality types of SML.

This can easily be adapted to our approach, because it only affects the unification of type expressions.

A much harder problem would be to allow arbitrary overloading as in Ada, i.e. having only the meta-restriction that it can be resolved in *some* way. This is already complicated for Ada and even more complicated in the presence of polymorphism.

5.2 Preserving Polymorphism

We mentioned that the code of a function is *at least* as polymorphic as its source. To make the code a sound basis for a type check, we need a bit more; the code also has to be *at most* as polymorphic as the original function.

But neither the one nor the other condition is always fulfilled.

One problem is the treatment of certain kinds of language features, especially any kind of local declaration (`where`, `case`, lambda abstraction, list comprehensions, etc.). In our approach, any local declaration has to be made global in some way. There are standard techniques to do this, which — of course — preserve the operational meaning, but it is rather delicate to preserve the types as well. For example in Miranda, Release 1 and Release 2 treat local declarations differently: in Release 2 local declarations can be polymorphic, in Release 1 they could not be. But it is not necessary to change the *code* of a local declaration for the second release, because the operational semantics is not affected.

The following example (partitions of a list) is legal in Miranda Release 2, but not in Release 1:

```
parts = foldr op [[]]
        where
        op x [[]] = [[[x]]]
        op x xss = s' (++) (map (glue x)) (map ([x]:)) xss
        s' b f g x = f x $b g x
        glue x = s' (:) ((x:).hd) tl
```

The problem is the local function `s'`, a polymorphic combinator used here with two different types. Any translation technique operationally sound is not necessarily type sound as well — simply globalising `s'` would be unsound for Release 1, because this makes `s'` polymorphic and permits the above use.

A special problem w.r.t. the preservation of the polymorphism of the source program is the treatment of type assertions.

5.3 Type Assertions

Type assertions are useful for several reasons: they can be used to locate the source of a type-error, they are sometimes necessary to resolve overloading, they can restrict the usage of polymorphic objects, and occasionally (rarely) they can enhance polymorphism.

There are two essentially different places for type assertions: (i) on the level of (global) declarations or modules and (ii) inside expressions or patterns.

Type assertions on declaration level do not affect the technique described here very much, they just enforce an additional check, additional to type inference. In most cases this either does not change the type of the concerned identifier at all, or it restricts its type to a less polymorphic instance. In the presence of recursive (particularly mutual recursive) definitions, type assertions can even enhance polymorphism. Consider the following small and rather pathological example:

```
twolist * ** ::= NIL | CONS * (twolist ** *)
length NIL = 0
length (CONS x y) = 1 + length y
```

Deducing the type of `length` in the standard way results in the type `twolist * * -> num`. But the more general type assertion `length::twolist * ** -> num` would also be consistent for `length`. This is not a problem for our approach, because it only affects the type environment for a type interpretation of the code. Of course we have to check that a type assertion about a name is consistent with its definition, but this check does not interfer with our alternative code interpretation. SML does not support such an enhancement of polymorphism by type assertions.

A type assertion on expression level only enforces an additional check too, but there the check interferes with the code, i.e. there is the problem *what* to check. If we check the types by inspecting the code, the level of expressions has vanished. A type assertion at a subexpression would most easily coincide with our approach by adding a `Type` instruction to the abstract machine, which has a meaning for the type check interpretation of the code (similar as `Eq`), but is like a `Skip` in the standard interpretation.

5.4 Irrefutable Patterns

A serious problem is the exploitation of type definitions in the code, especially if the constructors have been replaced by numbers referring to the type definition. In this case, it is possible that functions operating on different types have the same code. If wanted, such code has to be produced in a separate step.

Notice that the feature of *irrefutable patterns* in Miranda[4] is somewhat problematic in this respect, see the following example:

```
ignore (x,y) = "bingo"
```

In a lazy language, evaluation happens on demand during pattern matching. But a pattern like `(x,y)` does not necessarily require any evaluation to check for applicability of the rule, because any value of a pair type has this form. In Miranda, such a pattern can be replaced by a fresh variable, and each occurrence of a component variable of the pattern in the right-hand side of the definition by the application of the corresponding selector function to this variable. In the example there are no such occurrences and we should get:

```
ignore p = "bingo"
```

Obviously, the transformation has changed the type: the argument of `ignore` is no longer guaranteed to be a pair, it is more general. The same effect would occur if the code we base our type check on already reflects the irrefutability of patterns. A solution would again be to generate a `Type` instruction for such patterns, i.e. to add a type assertion, in the example for `p`.

5.5 Environments as Parameters

Some translation techniques provide a way to make certain environments at certain expressions visible that is not type-safe: it is to introduce an additional parameter for the expression which becomes bound to the corresponding environment. This is not type-safe if the environment contains identifiers of a polymorphic type and if the expression uses them in different instances.

Operationally, there is nothing wrong with this method, but ML-like type systems do not support the kind of type which would be needed here.

Such a translation typically appears in the context of fancy modularisation techniques, e.g. functors in SML, parameterised modules in Miranda, or the class system of Haskell. In particular, the translation proposed in [14] to handle the classes of Haskell is of such a form.

[4] In Haskell and SML the same problem occurs, although for slightly different reasons. In Haskell, irrefutable patterns are an explicit rather than an implicit feature, but this has similar consequences. SML is eager, so irrefutability has no semantical consequences, but the code for matching can be affected similarly though.

6 Conclusions

We have presented a method to infer the types of a functional program by abstract interpretation of its code. Advantages of this approach are methodological as well as practical.

A methodological advantage is related to the principle *divide and conquer*: many steps of analysis or translation can be done independently from any knowledge about types, and performing the type check on the produced code *forces* such a separation of tasks.

A practical advantage is that the type check can exploit structural information gained from earlier translation steps. If an optimisation finds *anonymous* variables (variables with only defining occurrences), the code will ignore them and hence the type check will ignore them as well — this is safe, because an anonymous variable has an arbitrary type. If an optimisation finds common subexpressions and reduces the code to create only one instance of a subexpression, then type checking will be performed for only this copy — this is safe, because compound expression are never polymorphic in ML-like type systems.

An apparent disadvantage is that optimisations based on type information cannot be used. This particularly holds for the representation of types, e.g. the use of the same representation for different types. But such optimisations can be done afterwards.

There are two more serious problems. One is the soundness of the approach, i.e. it is only sound if the code generation preserves the types, in particular if it preserves the degree of polymorphism. This requires a careful investigation of each code generation step. Fortunately, most translations do not change the type, i.e. in practice it is more the kind of problem one has to keep in mind, rather than the kind of problem one has to do something about. Another serious problem is to produce *useful* error messages. Among the usual (type independent) compilation techniques for functional languages are methods like the translation of list comprehension into combinators or λ-lifting of local operations which transform a program in a non-trivial way, and if the error messages of the type checker are based on the already transformed program (because the code is produced for it), then they may not be very helpful.

It would be interesting to see whether this approach can be applied to other forms of abstract interpretation as well.

References

[1] Hendrik P. Barendregt. *The Lambda-Calculus, its Syntax and Semantics.* North-Holland, 1984.

[2] A. J. Field and P. G. Harrison. *Functional Programming.* Addison-Wesley, 1988.

[3] J. Y. Girard. Une extension de l'interprétation de godel a l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium.* North Holland, 1971.

[4] Fritz Henglein. *Polymorphic Type Inference and Semi-Unification.* PhD thesis, Courant Institute of Mathematical Sciences, New York University, 1989.

[5] A.J. Kfoury, J. Tiuryn, and P. Urcyczyn. The undecidability of the semi-unification problem. In *22nd Annual ACM Symposium on Theory of Computing*, pages 468–476, 1990.

[6] Jan Willem Klop. Term rewriting systems, a tutorial. *EATCS bulletin*, 32:143–183, 1987.

[7] Jan Willem Klop. Term rewriting systems. In S. Abramsky, D. M. Gabbai, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science.* Oxford University Press, 199? (to appear).

[8] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.

[9] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9:157–166, 1966.

[10] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[11] Simon Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[12] John Reynolds. Towards a theory of type structure. In *Paris Colloquium on Programming*, 1974.

[13] Christopher Strachey. Fundamental concepts in programming languages. In *Notes for the International Summer School in Computer Programming*, 1967.

[14] P. Wadler and S. Blott. How to make *ad hoc* polymorphism less *ad hoc.* In *16th ACM Symposion on Principles of Programming Languages*, pages 60–76, January 1989. Austin, Texas.

Appendix: Unification Algorithm

In the text, the function `unify` was used several times, but it has not been defined yet. As mentioned, we need just ordinary first-order unification, so we give its definition here for the sake of completeness.

The algorithm uses a simple divide-and-conquer strategy, i.e. it unifies two terms by unifying their subterms and composing the resulting unifiers.

```
unify (Tvar n) x
      = (True,[]), if x=Tvar n
      = (True,[(n,x)]), if ~ occur n x
      = (False, undef), otherwise
unify x (Tvar n) = unify (Tvar n) x
unify (Tcon s tl) (Tcon s' tl') =
      foldr comp (True,[]) (zipWith unify tl tl'), if s=s'
unify p q = (False, undef), otherwise

zipWith f (a:as) (b:bs) = f a b : zipWith f as bs
zipWith f x y           = []
```

The function `unify` does proper unification, `comp` tries to compose the unifiers of the subproblems. It is assumed that each type constructor `s` has a fixed arity a_s, i.e. for any `(Tcon s ls)` the list `ls` has length a_s. This assumption is not really necessary for first-order unification, but it is simply true for the languages with which we are concerned.

```
occur n (Tvar m) = n=m
occur n (Tcon f ls) = [t|t<-ls; occur n t] ~= []
```

Ordinary unification of first order terms requires an occur check, here performed by the function `occur`. It would be no problem to enrich the approach to cyclic types, i.e. to use trees with finitely many different subtrees instead of finite trees, but this is beyond the scope of this paper and is of no significance for the approach.

```
comp (True,x)(True,y) = foldr comp1 (True,x) y
comp x y = (False,undef), otherwise

comp1 x (False,y) = (False,undef)
comp1 (n,t) (True,s)
      = comp (unify t (hd ua)) (True,s), if ua~=[]
      = (True,s), if t'=Tvar n
      = (False,undef), if occur n t'
      = (True,(n,t'):s'), otherwise
        where
        ua = lookup n s
        t' = subst s t
        s' = subst_assoc [(n,t')] s
```

The function `comp` composes two unifiers. An occur check (for `comp1`) is necessary again, because the type `t`, the substitute for `n`, might contain a type variable `m`, which might be substituted in `(True,s)` by a type `u`, which again might contain the type variable `n`.

```
lookup lab ls = [val | (x,val)<-ls; x=lab]
subst sub (Tvar n)
        = hd u, if u~=[]
        = Tvar n, otherwise
          where u = lookup n sub
subst sub (Tcon f ls) = Tcon f (map (subst sub) ls)

subst_assoc sub ls = [(x,subst sub t) | (x,t)<-ls]
subst_env sub (g,l,n) = (g,subst_assoc sub l,n)
subst_all sub (tenv,lss) =
      (subst_env sub tenv,map (map (subst sub)) lss)
```

Application of a substitution is done by `subst`, for types; by `subst_assoc`, for association lists; by `subst_env`, for type environments; and by `subst_all`, for a certain combination of those.