ELSEVIER

Theoretical Computer Science 187 (1997) 147–165

Theoretical Computer Science

Indexed types

Christoph Zenger

Universität Karlsruhe, Institut für Algorithmen und Kognitive Systeme, D-76128 Karlsruhe, Germany

Abstract

A new extension of the Hindley/Milner type system is proposed. The type system has algebraic types, that have not only type parameters but also value parameters (indices). This allows for example to parameterize matrices and vectors by their size and to check size compatibility statically. This is especially of interest in computer algebra.

Keywords: Type system; Functional languages; Type inference; Dependent types; Constraints

1. Introduction

In a functional language with a Hindley/Milner type system, as described in [20] and [10], we declare matrix multiplication and determinant for integer matrices with the following types:

```
matmult :: Matrix Int -> Matrix Int -> Matrix Int
determinant :: Matrix Int -> Int
```

With these declarations the compiler/type-checker cannot see, whether we try to multiply two matrices of incompatible sizes. For taking the determinant of a non-square matrix it is just the same. Errors of this kind will result in run-time errors or, even worse, simply return an inadequate result. The solution we present, solves this problem by introducing algebraic types that take complex numbers, called indices, as parameters. Furthermore, one can constrain these indices by requiring that certain polynomial equations in these indices hold. As an example the definition of a vector might look as follows:

```
data Vector a #n =
   Vnil, n = 0 |
   Vcons a (Vector a m), n = m+1
```

* E-mail: zenger@ira.uka.de.

PII S0304-3975(97)00062-5

n is such an index (In type definitions we will prefix indices by a # to distinguish them from type parameters). The constraints are that empty vectors that have zero length, and that vectors constructed from first element and rest have length one greater than the rest. `(Vcons 3 Vnil)` has type `(Vector Int 1)`. Variables that do not occur on the left hand side (`m` in the example) are existentially quantified indices.

The above functions can now be declared as follows:

```
matmult :: Matrix Int n m -> Matrix Int m k -> Matrix Int n k
determinant :: Matrix Int n n -> Int
```

Here we specified the constraint that only matrices with compatible sizes can be multiplied and that for calculating the determinant the two size parameters of a matrix must be equal.

Given such declarations of types and function signatures the compiler/type checker finds out about problems like multiplication of matrices with incompatible sizes and determinant of a non-square matrix at compile-time and reports a type error.

Indexed types do not allow to type more expressions than the Hindley/Milner system but they do allow to type more fine-grained, such that we can detect more errors by type checking. This static error detection will be even more important for specification languages than for functional languages, because there are no run-time tests.

Constraints have to be enforced by the type system during construction of objects of the algebraic types and they can be used as known facts when decomposing in the case construct respectively pattern matching. There may be a different set of constraints for each constructor. A difficult question is, what kind of constraints on the indices should be allowed. It turns out, that polynomial equations (equations of the form $p(\ldots)=0$ where p is a polynomial) are a suitable choice. They are expressive enough to describe many constraints that arise in practice, yet we can still have type inference using Gröbner basis techniques (See Buchberger [1] for an overview on Gröbner bases). Still other kinds of constraints have to be considered. We will discuss that in Section 5.

In the second section we will now introduce the system of indexed types. In the third section we will give some examples that can be expressed in the type system, before we discuss some of its limitations in section four. We shortly discuss the constraint system before we describe the type inference. We conclude by relating our approach to work done by others and discussing some topics for future research.

2. System of indexed types

We propose an extension of the Hindley/Milner system by indexed types. The type language is constructed along the lines of qualified types [14], where the predicates are polynomial equations. But here types may also have polynomials as parameters.

Definition 1 (*Types*). Types can be constructed as follows:

(i) polynomials

$$p \in \mathbb{C}[\bar{n}]$$

where $\bar{n}$ are polynomial variables, and $\mathbb{C}$ denotes the complex numbers.

(ii) types

$$\tau = \tau \to \tau | T \bar{\tau} \bar{p} | \alpha,$$

where T is a type constructor for an algebraic type and α is a type variable.

(iii) qualified types [1]

$$\rho = \bar{p} \Rightarrow \rho | \tau$$

(iv) type schemes

$$\sigma = \forall \alpha.\sigma | \forall n.\sigma | \rho$$

The intuition behind a qualified type $\bar{p} \Rightarrow \tau$ is that an expression of the qualified type is of type τ if the constraints $\bar{p}$ are satisfied.

$\mathrm{tv}(\sigma)$ denotes the free type variables, $\mathrm{pv}(\sigma)$ the free polynomial variables. Similarly tv and pv will be used on contexts Γ and constraints G.

The following are the rules for variable introduction, arrow introduction and elimination, quantifier introduction and elimination, and the let-rule:

$$(var)\frac{(x:\sigma) \in \Gamma}{G|\Gamma \vdash x:\sigma}$$

$$(\to E)\frac{G|\Gamma \vdash E:\tau' \to \tau \qquad G|\Gamma \vdash F:\tau'}{G|\Gamma \vdash EF:\tau}$$

$$(\to I)\frac{G|\Gamma_x, x:\tau' \vdash E:\tau}{G|\Gamma \vdash \lambda x.E:\tau' \to \tau}$$

$$(\forall E)\frac{G|\Gamma \vdash E:\forall \alpha.\sigma}{G|\Gamma \vdash E:[\tau/\alpha]\sigma}$$

$$(\forall I)\frac{G|\Gamma \vdash E:\sigma}{G|\Gamma \vdash E:\forall \alpha.\sigma} \quad \alpha \notin \mathrm{tv}(\Gamma)$$

$$(let)\frac{G|\Gamma \vdash E:\sigma \qquad G'|\Gamma_x, x:\sigma \vdash F:\tau}{G' + G|\Gamma \vdash (\textbf{let } x = E \textbf{ in } F):\tau}$$

All of them are straightforward generalizations of the rules of the Hindley/Milner system. The G on the left side denotes a set of constraints on the polynomial variables in the judgement, that are not bound by quantifiers.

[1] We do not distinguish between $p \Rightarrow q \Rightarrow \rho$ and $p, q \Rightarrow \rho$.

We need a special rule for building fixpoints, because in contrast to e.g. ML types of recursive functions need to be declared.

$$(fix)\ \frac{G|\Gamma_x, x:\sigma \vdash E:\sigma}{G|\Gamma \vdash (\textbf{fix}\ x::\sigma\ \textbf{in}\ E):\sigma} \begin{cases} \mathrm{tv}(\sigma)=\emptyset \\ \mathrm{pv}(\sigma)=\emptyset \end{cases}$$

If we did not allow type-schemes here, we would not be allowed to use special instances of the function f on the right-hand side of the definition of f. For example in Gofer the following definition for the length of a list

```
len Nil = 0,
len (Cons x Nil) = 1,
len (Cons x xs) = (len xs) + (len [1])
```

has type `List Int -> Int` and not the more general type `List a -> Int`. Otherwise type inference would not be computable [21, 17]. This is obviously an artificial example, and indeed it is not much of a restriction in languages like Gofer, Haskell or ML, but as we shall see in the case of indexed types we need true polymorphic recursion. Besides Odersky and Läufer [23] showed already how to get true polymorphic recursion by declarations.

Because we also have quantification over index variables, we need introduction and elimination rules for these. They are rather similar to introduction and elimination rules for ordinary quantification.

$$(\forall_n E)\ \frac{G|\Gamma \vdash E:\forall n.\sigma}{G|\Gamma \vdash E:[p/n]\sigma}$$

$$(\forall_n I)\ \frac{G|\Gamma \vdash E:\sigma}{G|\Gamma \vdash E:\forall n.\sigma}\quad n \notin \mathrm{pv}(G)\cup \mathrm{pv}(\Gamma)$$

Besides, the constraints on the left-hand side tell us, that some types are equivalent ((`Vector a n`) and (`Vector a m`) will be equivalent types, if there is a constraint `m=n`). The rule (*equiv*) allows to change the type according to this equivalence. To formulate it we first need a definition:

Definition 2 (*Structural equivalent*). Two types τ and τ' are structural equivalent ($\tau \equiv^s \tau'$), iff they differ only in their polynomial parts. By $\tau - \tau'$ we denote the weakest constraint set, that implies equality of τ and τ'.

So, (`Vector Int n`) $\equiv^s$ (`Vector Int m`) and (`Vector Int n`) $-$ (`Vector Int m`) $=$ `n-m = 0`. $\sqrt{G}$ denotes the radical ideal, that is the ideal containing all polynomials p such that there is some m with $p^m \in G$.

$$(equiv)\frac{G|\Gamma \vdash E:\tau}{G|\Gamma \vdash E:\tau'} \begin{cases} \tau \equiv^s \tau' \\ (\tau - \tau') \subseteq \sqrt{G} \end{cases}$$

Before we introduce rules which allow to shift constraints from left to right and vice versa, we need another definition:

Definition 3 (*Elimination Ideal*). $\mathrm{Elim}(G,V) := G \cap k[\mathrm{pv}(G) \setminus V]$.
$\mathrm{eliminable}(G,V) : \Leftrightarrow (\exists V.G) \subseteq \mathrm{Elim}(G,V)$.

The elimination ideal $\mathrm{Elim}(G,V)$ is the set of polynomial constraints that are in G, but do not contain variables from V. $\mathrm{eliminable}(G,V)$ checks, whether a solution of $\mathrm{Elim}(G,V)$ can always be extended to a solution of G.

We can calculate elimination ideals using Gröbner bases and we can also check the condition for elimination. Furthermore we can calculate whether a set P of polynomial constraints implies another set Q of constraints (each polynomial in Q vanishes at points where every polynomial from P vanishes), because this is equivalent to one ideal being a subideal of the radical ideal of the other ideal. See standard textbooks on Gröbner bases computations such as [6] or [3] for that. We are going to use these computations for inferring types.

$$(\Rightarrow E)\ \frac{G|\Gamma \vdash E : P \Rightarrow \rho}{G|\Gamma \vdash E : \rho} \quad P \subseteq \sqrt{G}$$

$$(\Rightarrow I)\ \frac{G + P|\Gamma \vdash E : \rho}{G|\Gamma \vdash E : P \Rightarrow \rho}$$

$$(Elim)\ \frac{G\,|\,\Gamma \vdash E : \tau}{\mathrm{Elim}(G,n)|\Gamma \vdash E : \tau} \begin{cases} n \notin \mathrm{pv}(\Gamma),\ n \notin \mathrm{pv}(\tau) \\ \mathrm{eliminable}(G,n) \end{cases}$$

Basically we can eliminate a constraint on the right side, if it is implied by G, and we can shift constraints from the left to the right. Constraints that do not really constrain the right-hand side may be eliminated. This applies for example if we have the judgement $\mathtt{m}=\mathtt{n}|x : (\mathtt{Vector\ n}) \vdash x : (\mathtt{Vector\ n})$. Here `m` does not occur anywhere else, so `m=n` can be safely eliminated.

We need to test membership in the radical ideal because $n^2 = 0$ implies $n = 0$, though $\langle n \rangle$ is not a subideal of $\langle n^2 \rangle$ but only of $\sqrt{\langle n^2 \rangle} = \langle n \rangle$. To make sure that we can really eliminate we have to test, whether an `n` always exists. We can do that using Gröbner basis techniques.

Finally, we want to have algebraic types in our type system, otherwise we could not build any types with indices in the first place. Algebraic types can be declared as follows:

$$\mathtt{data\ T}\ \bar{\alpha}\bar{n} = C_1\bar{\tau}_1, Q_1 | \ldots | C_k\bar{\tau}_k, Q_k$$

We assume here that the polynomial variables occur at most once in each $\bar{\tau}_i$. This is no effective constraint, because a data declaration `data T n=D n n` can always be rewritten as `data T n=D n m, n=m`. In practice the first would be regarded as syntactic sugar for the second.

When elements of algebraic types are constructed, the type system has to ensure that the constraints Q_i are satisfied. On the other hand, when destructing, we may assume Q_i in the C_i alternative of the case construct.

$$(algI)\ \frac{}{G|\Gamma \vdash C_i : \forall \bar{n}.\forall \bar{m}.\forall \bar{\alpha}.Q_i \Rightarrow \bar{\tau}_i \to T\,\bar{\alpha}\bar{n}}$$

$$(algE)\ \frac{G|\Gamma \vdash z : T\,\bar{\mu}\bar{q} \qquad G + \theta Q_i|\Gamma_{\bar{x}}, \bar{x} : \theta\bar{\tau}_i \vdash E_i : \tau'}{G|\Gamma \vdash \textbf{case}\ z\ \textbf{of}\ \{C_i\bar{x}_i \Rightarrow E_i\}_{i=1\ldots k} : \tau'} \begin{cases} \theta = [\bar{\kappa}/\bar{m}, \bar{q}/\bar{n}, \bar{\mu}/\bar{\alpha}] \\ \kappa \notin \mathrm{pv}(\tau') \end{cases}$$

Here, $\bar{q}$ are polynomials, $\bar{\tau}$ types, $\bar{\kappa}$ are variables that play the role of Skolem constants, and $\bar{m}$ are the polynomial variables occurring in $\bar{\tau}_i$ but not in $\bar{n}$. Note, that existential quantification is always encapsulated in a constructor and is immediately cancelled with an universal quantification. Logically this relates to a rule

$$\frac{C \vdash \exists\kappa.A \qquad C \vdash \forall\kappa.(A \to B)}{C \vdash B} \qquad \kappa \notin \mathrm{tv}(B)$$

The encapsulation of the existential quantification in the type constructor is necessary to make type inference possible.

3. Using indexed types

To make the examples in this section more readable, we are going to use pattern matching syntax instead of explicit case expressions and recursive definitions with type declarations as they are common in Haskell or Gofer instead of building explicit fixpoints. There are standard techniques to reduce this to the above constructs [25].

3.1. Scalar product

We start with recalling the type declaration of vectors:

```
data Vector a #n =
   Vnil, n = 0 |
   Vcons a (Vector a m), n = m+1
```

The definition of a scalar product is straightforward:

```
sprod :: Vector Int n -> Vector Int n -> Int
sprod Vnil Vnil = 0
sprod (Vcons x xs) (Vcons y ys) = x*y + (sprod xs ys)
```

We have to declare the type of the recursive function, otherwise the definition is quite similar to a definition for lists. The difference, however, is that the expression `(sprod Vnil (Vcons 1 Vnil))` will be rejected by the compiler and a type error will be reported.

Note that the type of `sprod` at the recursive invocation has different indices $(n-1)$ than the type of the `sprod` function on the left hand side (n), that is we have true polymorphic recursion in the index. Due to this fact we need to declare types.

3.2. Vector comprehensions

List comprehensions, first used by Turner [27], are very elegant and frequently used in functional programming. For example the map function for lists can be expressed nicely as follows:

```
map f l = [ f x | x <- l ]
```

This gives as a result the list of all `f x`, where `x` is from `l`, very similar to the mathematical set comprehension $\{f(x)|x \in L\}$. An analog for vector comprehensions can also be established:

```
vmap f v = << f x | x <- v >>
```

This is a very intuitive formulation and there is no possibility of errors at the vector boundaries. One may even allow vectors in list comprehensions, as for example in a function that converts vectors to lists.

```
v2l :: Vector a n -> [a]
v2l v = [ x | x <- v ]
```

However, the converse is not true. Lists cannot be used in vector comprehensions, for it is impossible to check the length statically. Also filters, a very effective feature of list comprehensions, cannot possibly be used in vector comprehensions, but we shall see shortly how to get around this.

3.3. Sorting

Due to the lack of filters in vector comprehensions we cannot rewrite the following `quicksort` easily for vectors.

```
quicksort [] = []
quicksort (x:xs) = (quicksort [ y | y <- xs, y <= x ]) ++ [x]
   ++ (quicksort [ y | y <- xs, y > x ])
```

We have to split a vector into two vectors, although the actual sizes of the subvectors are only known at run-time. But we know that the sum of the two lengths of the subvectors is exactly the same as the size of the original. By introducing a new data type `SplitVector`, we can make use of that fact.

```
data SplitVector a #n =
   Spv (Vector a m) (Vector a k), m + k = n
```

Now we may write a function `vfilter` similar to the common `filter` function in Hindley/Milner languages. `vfilter` splits a vector into two, where the first one

contains all the elements for which the function f yields true, the second one all the others.

```
vfilter :: (a -> Bool) -> Vector a n -> SplitVector a n
vfilter f Vnil = (Spv Vnil Vnil)
vfilter f (Vcons x xs) =
   if (f x) (Spv (Vcons x l) r) (Spv l (Vcons x r)) where
      (Spv l r) = (vfilter f xs)
```

We cannot just use the pair constructor Pair instead of SplitVector, because then the information that the sum of the lengths is just the length of the argument would be lost and a type could not be found.

The quicksort function itself is almost as short as the original for lists and has the advantage that it checks that the resulting vector has exactly the same length as the original. If we forget to add the partition element x, the compiler reports a type error.

```
quicksort :: Vector Int n -> Vector Int n
quicksort Vnil = Vnil
quicksort (Vcons x xs) = vappend (quicksort l)
                         (Vcons x (quicksort r)) where
   (Spv l r) = vfilter (\y -> y < x) xs
```

Here vappend is the vector analog to append. We can formulate vappend also with an accumulating parameter [2].

3.4. *Vectors and lists*

We saw v2l, a function that converts vectors to lists. The converse is not possible, because we cannot know the length in advance. However we know that a length will exist and encapsulate this in the constructor AnyVec:

```
data AnyVector a = AnyVec (Vector a n)
```

Now we can write a function l2av that converts a list to such a vector.

```
l2av :: (List a) -> (AnyVector a)
l2av Nil = (AnyVec Vnil)
l2av (Cons x xs) = (AnyVec (Vcons x xs'))
   where (AnyVec xs') = (l2av xs)
```

Now we can again use this to implement quicklist, a quicksort on lists using our routine quicksort.

```
quicklist :: (List a) -> (List a)
quicklist xs = (v2l (quicksort xs'))
   where (AnyVec xs') = (l2av xs)
```

The advantage is that quicksort is internally checked. v2l and l2av seem to introduce a large run-time overhead, but we hope to reduce it by using deforestation techniques [28].

3.5. Matrix lists

When we supply a data type with additional parameters, here indices, to reflect more of the structure in the type, a typical problem that arises is that some lists become inhomogeneous in the type. As an example consider a list of matrices of different sizes. This was a homogeneous list, when the data type matrix did not have size parameters, but is not so now.

But there is a way out: depending on what we want to do afterwards, we can build special lists with size constraints. For example a list which contains square matrices.

```
data SqMatrList a =
   Snil |
   Scons (Matrix a n m) (SqMatrList a), n = m
```

We can then e.g. calculate the sum of all the determinants:

```
detsum :: SqMatrList Int -> Int
detsum Snil = 0
detsum Scons m l = (determinant m) + (detsum l)
```

We can also declare lists of matrices, which have compatible sizes for multiplication:

```
data MatrixList a #m #n =
   Mnil, m = n |
   Mcons (Matrix a m k1) (MatrixList a k2 n), k1 = k2
```

Now we can multiply all the matrices in a non-empty list

```
mult :: MatrixList Int n k -> Matrix Int n k
mult (Mcons m Mnil) = m
mult (Mcons m l) = (matmult m (mult l))
```

Note, however, what we cannot do is to return a unit matrix for the empty list. We will discuss this problem in the next section.

4. Limitations

We have seen some nice examples with indexed types. This section now shows some of the deficiencies of indexed types as they were presented above.

4.1. Usage of the Index

We can determine the size of a vector with

```
vlen :: Vector a n -> Int
vlen Vnil = 0
vlen (Vcons x xs) = 1 + (vlen xs)
```

just as we can compute the length of a list, but we cannot use the index as in

```
vsize :: Vector a n -> Int
vsize _ = n
```

though this would be an elegant notation. Another example is the construction of a zero vector or a unit matrix as above:

```
zero_vector :: Vector Int n
zero_vector = Vnil
zero_vector = (Vcons 0 (zero_vector))
```

Here we would hope that after type-inference we could supply an n and from that choose the right branch at run-time.

One reason for the failure is that the type inference only checks whether the constraints are satisfied and does not actually construct n, not even a way to compute it at run-time. If the compiler did this, it would require the implementation to carry around all the indices at run-time and it is not clear whether the ability to use the index would outweigh this performance overhead.

In the second example we see an additional problem: The semantics is influenced by the type inference. Consider the expression `(sprod zero_vector zero_vector)`! The term looks well-typed but what should its semantics be? The type inference cannot infer the lengths of the zero-vectors.

Note here that multiple solutions to the type-problem do not hurt us, because the solution doesn't influence the semantics of execution.

4.2. *Dependent types*

Suppose we want to read two vectors from a file, calculate a scalar product and print the result. There is no way to check statically whether they are of equal size and a program doing this by using the data type `Vector` would be necessarily ill-typed. A way out of this problem would be to allow explicit dependent types: dependent sums `(n:Int, Vector Int n)` and dependent products `(n:Int -> Vector Int n)`. Dependent sums could be used to read a pair of a vector and its length. We would then need a special if clause, such that in the true alternative of an `if (n=m)` we would know (statically!) that `Vector Int n` and `Vector Int m` have matching sizes.

Similarly dependent products could be used. We would first read the size of the vectors and then the vectors themselves. A function reading a vector with a size given as argument would then have a dependent product type.

Another problem from computer algebra, where we would want to have dependent types, is a data type `IntMod n` which represents integers modulo n.

```
data IntMod #n = Mod Int n:Int
```

We could then write functions for the multiplication and Chinese remaindering with detailed type information.

```
imul :: IntMod n -> IntMod n ->  IntMod n
chinese :: IntMod n ->  IntMod m ->  IntMod (m*n)
```

Mixing of different moduli would be detected as type error. The difficulty in introducing dependent types lies in the integration of explicit dependent types and indices. Besides dependent types can only be checked if we use term-equality on the values.

To see how far we can get without dependent types we look again at the `zero_vector` example. First a typing with dependent types:

```
zero_vector :: n:Int -> Vector Int n
```

But using a definition

```
data Int' #n =
  Zero, n =0 |
  Succ (Int' m), n = m + 1 |
  Pred (Int' m), n = m - 1
```

we can write something quite similar:

```
zero_vector :: (Int' n) -> (Vector Int n)
```

and use this to multiply two zero-vectors of length one.

```
(sprod (zerovector (Succ (Zero))) (zerovector (Succ (Zero))))
```

Of course we would have to invent syntactic sugar for that. How far we can get using this approach seems a promising research topic.

4.3. *Vector access*

Of course we can write a function, accessing the n-th element of a vector

```
access :: (Vector a n) -> Int -> a
access (Vcons x xs) 0 = x
access (Vcons x xs) m = (access xs (m-1))
```

but we have no way of checking statically, that the access doesn't fail. This would need dependent types as well as a comparison operator in the constraints. But in functional programming these accesses are rarely used. Instead we use `vmap` and this gives us security.

4.4. *Higher order functions*

Some constructions which work in the Hindley/Milner system cease to work when introducing indexed types. To illustrate this we look at the following definition for the transposition of a list of lists, adapted from [13].

```
transpose :: [[a]] -> [[a]]
transpose = foldr (\xs xss -> zipWith (Cons) xs xss) Nil
```

where foldr and zipWith are standard functions

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith z (a:as) (b:bs) = z a b : zipWith z as bs
zipWith _ _ _ = []
```

Now, what happens if we try the same with vectors? The corresponding vfoldr and vzipWith are easy to rewrite:

```
vfoldr :: (a -> b -> b) -> b -> (Vector a n) -> b
vfoldr f z Vnil = z
vfoldr f z (Vcons x xs)  = f x (vfoldr f z xs)

vzipWith :: (a -> b -> c) -> (Vector a n) ->
   (Vector b n) -> (Vector c n)
vzipWith z (Vcons a as) (Vcons b bs) =
   Vcons (z a b) (vzipWith z as bs)
vzipWith z Vnil Vnil = Vnil
```

but when we rewrite transpose as

```
transpose :: Vector (Vector a m) n -> Vector (Vector a n) m
transpose = vfoldr (\xs xss -> vzipWith (Vcons) xs xss) Vnil
```

we end up with a type error, because (\xs xss -> vzipWith (Vcons) xs xss) has type (Vector a n) -> (Vector (Vector a m) n) -> (Vector(Vector a k) n), k = m + 1, and this does not match the argument type of vfoldr.

The reason behind this is, that we have to fix the size of the vectors on which Vcons works, when we pass it to vfoldr. But then vfoldr cannot see that it indeed works on all sizes. Allowing type schemes instead of just types as arguments to functions as in [23] may solve this problem, but it is not yet clear whether this can be easily combined with indexed types.

5. Constraints

5.1. *Indexing with Integers/Complex*

All the way we used types which we thought of as indexed by integers or even non-negative integers, though actually the indices are complex numbers. Does this pose a serious problem? No! Many formulas, e.g. formulas of the form $p(\bar{x}) = 0 \Rightarrow q(\bar{x}) = 0$, that are valid for complex numbers are also valid for the integers.

The problem that arises is twofold. Firstly, a program might type-check, though the user would not expect it to. For example there is no difference between the types

`(Vector a n) -> (Vector a n)` and `(Vector a (2*n)) -> (Vector a (2*n))` and `f (Vcons 3 Vnil)` would type-check if `f` has either of the above types. Hence, there is no way to specify that an index has to be a multiple of another, that it has to be a square of an integer and so on.

The other problem is that a program may not type-check, because type correctness relies on the fact that only integers are substituted for constraint variables.

5.2. *Other systems*

In this paper we presented the constraint system of polynomial equations.[2] An extension of this system to include also comparisons using cylindrical algebraic decomposition [7] is worth considering. However, it is unclear, whether the second problem above might be an obstacle. We cannot prove $n > 0 \Rightarrow n \geqslant 1$, though the type-checker might need this in typical situations.

Another choice which has to be investigated is Presburger arithmetic [16] (thanks to the referee for pointing that out). Besides comparison operators this constraint system has the advantage of being over the integers, but on the other hand we lose the possibility of general polynomial constraints.

6. Type inference

We present the type inference along the lines of Jones [14]. After giving some auxiliary definitions, we start with presenting a syntax-directed deduction system $\vdash^S$, in which for every term of the language there exists exactly one rule to derive its type. The second step will be a deduction system $\vdash^W$, which can be read as an attribute grammar and thus gives an algorithm W for computing the type of an expression.

We want to define a relation "more general" denoted $\leqslant$ on constrained type-schemes $(P|\sigma)$, which has the property

$$(P|\sigma) \geqslant_\Gamma (P'|\sigma') \text{ iff } (P'|\Gamma \vdash E : \sigma' \text{ implies } P|\Gamma \vdash E : \sigma).$$

We start with the special case $P' = \emptyset$ and $\sigma' = R \Rightarrow \mu$ and call $\leqslant$ "generic instance" in this context.

Definition 4 (*Generic instance*). $R \Rightarrow \mu$ is a generic instance of $(P|\forall\bar{\alpha}.\forall\bar{n}.Q \Rightarrow v)$ under Γ iff there exist $\bar{\tau}$ such that

(i) $\text{Elim}(P \wedge Q \wedge [\bar{\tau}/\bar{\alpha}]v - \mu, V) \subseteq R$

(ii) $\text{eliminable}(R \wedge P \wedge Q \wedge [\bar{\tau}/\bar{\alpha}]v - \mu, V)$

where $V = \text{pv}(Q,P,v) \setminus \text{pv}(\Gamma)$, $\bar{\alpha},\bar{n} \notin \text{pv}(\Gamma)$ and $\text{pv}(R,\mu) \cap \text{pv}(P,Q,v) \subseteq \text{pv}(\Gamma)$. We are allowed to achieve these conditions by renaming variables that do not occur free in Γ.

[2] Since the writing of the paper we generalized this to quantified boolean formulas over polynomial equations.

We will denote the generic instance relation by $R \Rightarrow \mu \leqslant_\Gamma (P|\forall\bar{\alpha}.\forall\bar{n}.Q \Rightarrow \nu)$. We take σ as a shorthand for $(\emptyset|\sigma)$.

Definition 5 (*More general*). $(P|\sigma)$ is more general than $(P'|\sigma')$ under Γ iff every generic instance of $(P'|\sigma')$ is also a generic instance of $(P|\sigma)$. We denote this by $(P|\sigma) \geqslant_\Gamma (P'|\sigma')$.

We still need a definition of a generalization function, then we will be able to present the syntax-directed system.

Definition 6 (*Generalization*). $Gen_\Gamma(\rho) = \forall(\mathrm{tv}(\rho) \setminus \mathrm{tv}(\Gamma)).\forall(\mathrm{pv}(\rho) \setminus \mathrm{pv}(\Gamma)).\rho$.

Definition 7 (*Syntactic type system*).

$$(var)^S \frac{(x:\sigma) \in \Gamma \qquad (G \Rightarrow \tau) \leqslant_\Gamma \sigma}{G|\Gamma \vdash^S x : \tau}$$

$$(\rightarrow E)^S \frac{G|\Gamma \vdash^S E : \tau' \rightarrow \tau \qquad G|\Gamma \vdash^S F : \tau'}{G|\Gamma \vdash^S EF : \tau}$$

$$(\rightarrow I)^S \frac{G|\Gamma_x, x : \tau' \vdash^S E : \tau}{G|\Gamma \vdash^S \lambda x.E : \tau' \rightarrow \tau}$$

$$(let)^S \frac{G|\Gamma \vdash^S E : \tau' \qquad G'|\Gamma_x, x : \sigma \vdash^S F : \tau \qquad \sigma = Gen_\Gamma(G \rightarrow \tau')}{G'|\Gamma \vdash^S (\textbf{let } x = E \textbf{ in } F) : \tau}$$

$$(fix)^S \frac{G + P|\Gamma_x, x : \sigma \vdash^S E : \tau \quad \sigma = \forall\bar{\alpha}.\forall\bar{n}.P \Rightarrow \tau}{G + P|\Gamma \vdash^S (\textbf{fix } x :: \sigma \textbf{ in } E) : \tau} \begin{cases} \bar{n} \notin \mathrm{pv}(\Gamma, G) \\ \bar{\alpha} \notin \mathrm{tv}(\Gamma) \\ \mathrm{pv}(\sigma) = \emptyset \\ \mathrm{tv}(\sigma) = \emptyset \end{cases}$$

$$(algI)^S \frac{G \Rightarrow \tau \leqslant_\Gamma \forall\bar{n}.\forall\bar{m}.\forall\bar{\alpha}.\bar{\tau}_i \rightarrow T\bar{\alpha}\bar{n}}{G|\Gamma \vdash^S C_i : \tau}$$

$$(algE)^S \frac{G|\Gamma \vdash^S z : T\bar{\mu}\bar{q} \quad G + \theta Q_i|\Gamma_{\bar{x}}, \bar{x} : \theta\bar{\tau}_i \vdash^S E_i : \tau'}{G|\Gamma \vdash^S \textbf{case } z \textbf{ of } \{C_i\bar{x}_i \Rightarrow E_i\}_{i=1\ldots k} : \tau'} \begin{cases} \theta = [\bar{\kappa}/\bar{m}, \bar{q}/\bar{n}, \bar{\mu}/\bar{\alpha}] \\ \bar{\kappa} \notin \mathrm{pv}(\tau') \end{cases}$$

We claim a soundness property of $\vdash^S$ with respect to the original system. We believe, that $\vdash^S$ is also complete as stated below. However this seems less important, when we note, that at the next step to $\vdash^W$ we lose completeness.[3]

[3] Furthermore for a more sophisticated system developed in the meantime, completeness seems easier to prove.

Proposition 8 (Soundness). *If* $G|\Gamma \vdash^S E:\tau$, *then* $G|\Gamma \vdash E:\tau$.

Conjecture 9 (Completeness). *If* $G|\Gamma \vdash E:\sigma$, *then exists* G',τ *with* $G'|\Gamma \vdash^S E:\tau$ *and* $(G|\sigma) \leqslant_\Gamma Gen_\Gamma(G' \Rightarrow \tau)$.

All derivations trees in $\vdash^S$ for the type of an expression have the same form. Now we want to find a most general derivation. To this end we define a new deduction $\vdash^W$, which is computational in that it can be read as an attribute grammar that computes an ideal G, a substitution S on type variables, and a type τ, such that $G|S\Gamma \vdash^W E:\tau$ from Γ and E. Moreover, each derivation in $\vdash^W$ will be a derivation in $\vdash^S$. $\vdash^W$ uses some auxiliary functions:

- $univar(\tau)$ computes a type τ', which is structural equivalent to τ, but each polynomial in τ' consists of a single new polynomial variable, each polynomial variable occurring exactly once in τ'.
- $(I,S) = mgu(\bar{\tau})$ iff S is the most general substitution, such that $S\tau_i \equiv^s S\tau_j$ and $I = \sum_{ij} S\tau_i - S\tau_j$.
- $(\tau',S) = umgu(\bar{\tau})$ iff $(\_,S) = mgu(\bar{\tau}), \tau' = univar(S\tau_1)$.
- $(I,S) = mgs(\bar{T},\bar{\alpha})$ iff S is the most general substitution, such that $ST_i\alpha_k \equiv^s ST_j\alpha_k$ and $I = \sum_{ijk} ST_i\alpha_k - ST_j\alpha_k$.
- $(T',S) = umgs(\bar{T},\bar{\alpha})$ iff $(\_,S) = mgs(\bar{T},\bar{\alpha}), T' = [univar(S\alpha_k)/\alpha_k]$.
- Finally there is a function $mgc(\bar{Q},\bar{H},V_E,V_K)$ that computes an ideal G such that $H_i \subseteq G + Q_i$ (V_E, V_K are used for heuristic purposes only). The algorithm should find a large ideal G (few constraints). However, in general an optimal solution does not exist ($\langle m=0\rangle$ and $\langle m=n\rangle$ are both solutions to $mgc(n=0,m=0)$ but none is better than the other). The following algorithm for $mgc(\bar{Q},\bar{H},V_E,V_K)$ seems to work well in practice:
  - Compute a Gröbner basis in lex order, such that reduction with this Gröbner basis tends to eliminate V_E and to keep V_K.

    $$Q_i' = \text{gröbnerlex}(Q_i)$$

    where $v > w > u$ if $v \in V_E, w \notin V_E \cup V_K, u \in V_K$
  - Try to eliminate V as far as possible from H_i

    $$H_i' = \text{normalize}_{Q_i'}(H_i)$$

  - Eliminate heuristically variables from the Q_i'

    $$Q_i'' = \text{Elim}(Q_i', \text{pv}(Q_i') \setminus \text{pv}(H_i'))$$

    if $\text{eliminable}(Q_i', \text{pv}(Q_i') \setminus \text{pv}(H_i'))$.
  - If $\forall i. Q_i'' \neq \emptyset$ then return $\sum H_i'$.

- Choose j such that $Q_j'' = \emptyset$ and return

$$H_j + mgc(\langle Q_i'' \rangle_{i \neq j}, \langle H_i' + Q_j'' \rangle_{i \neq j}, V_E, V_K))$$

Now we can specify $\vdash^W$:

Definition 10 (*Type inference*).

$$(var)^W \frac{(x : \forall \bar{\alpha}.\forall \bar{n}.G \Rightarrow \tau) \in \Gamma \qquad \bar{\beta}, \bar{m} \text{ new}}{[\bar{m}/\bar{n}]G|\Gamma \vdash^W x : [\bar{\beta}/\bar{\alpha}][\bar{m}/\bar{n}]\tau}$$

$$(\rightarrow E)^W \frac{G|S\Gamma \vdash^W E : \tau \qquad H|S'S\Gamma \vdash^W F : \tau' \quad (I, U) = mgu(S'\tau, \tau' \rightarrow \alpha) \quad \alpha \text{ new}}{I + G + H|US'S\Gamma \vdash^W EF : U\alpha}$$

$$(\rightarrow I)^W \frac{G|S(\Gamma_x x : \alpha) \vdash^W E : \tau \qquad \alpha \text{ new}}{G|S\Gamma \vdash^W \lambda x.E : S\alpha \rightarrow \tau}$$

$$(let)^W \frac{G|S\Gamma \vdash^W E : \tau' \quad G'|S'(S\Gamma_x, x : \sigma) \vdash^W F : \tau \quad \sigma = Gen_\Gamma(G \Rightarrow \tau')}{G'|S'S\Gamma \vdash^W (\textbf{let } x = E \textbf{ in } F) : \tau}$$

$$(fix)^W \frac{\begin{array}{c} G|S(\Gamma_x, x : \sigma) \vdash^W E : \tau' \qquad \sigma = \forall \bar{\alpha}.\forall \bar{n}.P \Rightarrow \tau \\ (I, S') = mgu(\tau, \tau') \quad S'\tau' \equiv^s \tau \quad G' = mgc(P, I + G, \bar{n}, \emptyset) \\ \mathrm{tv}(\sigma) = \emptyset \qquad \mathrm{pv}(\sigma) = \emptyset \quad \bar{n} \notin \mathrm{pv}(G') \quad \bar{\beta}, \bar{m} \text{ new} \end{array}}{[\bar{m}/\bar{n}]P + G'|[\bar{\beta}/\bar{\alpha}, \bar{m}/\bar{n}]S'S\Gamma \vdash^W (\textbf{fix } x :: \sigma \textbf{ in } E) : [\bar{\beta}/\bar{\alpha}, \bar{m}/\bar{n}]\tau}$$

$$(algI)^W \frac{\bar{\beta}, \bar{k}, \bar{l} \text{ new}}{G|\Gamma \vdash^W C_i : [\bar{k}/\bar{m}, \bar{l}/\bar{n}, \bar{\beta}/\bar{\alpha}](\bar{\tau}_i \Rightarrow T\bar{\alpha}\bar{n})}$$

$$(algE)^W \frac{\begin{array}{c} G|S\Gamma \vdash^W z : \tau \quad (I, U) = mgu(T\bar{\beta}\bar{l}, \tau) \quad \bar{\beta}, \bar{l}, \bar{\kappa}_i \text{ new} \\ G_i|S_iU(S\Gamma, \bar{x} : [\bar{\kappa}_i/\bar{m}, \bar{l}/\bar{n}, \bar{\beta}/\bar{\alpha}]\bar{\tau}_i) \vdash^W E_i : \mu_i \\ (\tau', S') = umgu(\bar{\mu}, \mathrm{pv}(US\Gamma)) \quad (W, W') = umgs(S'\bar{S}, \mathrm{pv}(US\Gamma)) \\ H_i = G_i + (\tau' - S'\mu_i) + (W -_{\mathrm{pv}(US\Gamma)} W'S'S_i)) \\ K = mgc(\bar{H}, \overline{[\bar{\kappa}/\bar{m}, \bar{l}/\bar{n}]Q}, \bar{\kappa}_i, \mathrm{pv}(WUS\Gamma)) \qquad \bar{\kappa}_i \notin \mathrm{pv}(K) \end{array}}{G + I + K|WUS\Gamma \vdash^W \textbf{case } z \textbf{ of } \{C_i\bar{x}_i \Rightarrow E_i\}_{i=1\ldots k} : W\tau'}$$

The soundness of W with respect to $\vdash^S$ was already informally stated above.

Proposition 11 (Soundness W). *If* $G|S\Gamma \vdash^W E : \tau$ *then* $G|S\Gamma \vdash^S E : \tau$.

We do not have completeness for this system.[4] For the examples in the paper, however, W infers an adequate type.

7. Relation to other work

There are no proper dependent types in the presented system, because, if we regard the indices as types, there is still no parameterization of types with values. Nevertheless we can, as the examples above show, express many things for which other systems use dependent types.

In AXIOM [15] there are very general dependent types, dependent products, and sums. However, equality on values is defined by term-equality. This means for example that `(Vector n-1+1)` and `(Vector n)` are not equal types, because $n-1+1$ and n are not equal as terms. This allows to handle the integer modulo case elegantly, but on the other hand one cannot for example rewrite the above quicksort example in AXIOM.

In some type theoretic theorem provers like Coq [5], LEGO [19], and NuPrl [4] there are dependent types as well. Again, `(Vector n-1+1)` and `(Vector n)` do not have the same type, but it is easy to construct a conversion function from one to the other by giving a proof of $n-1+1=n$. But the burden of the proof is on the programmer. Coquand describes a type checker for such a theorem prover [8].

Existential quantification was studied by Cardelli and Wegner [9], Mitchell and Plotkin [22] for the description of abstract data types. Type inference for existential quantification in algebraic types is studied in detail by Perry [24] and Läufer [18]. We do not use this in full generality here, because we allow existential quantification only for indices.

Sulzmann [26] examines type inference for a general class of constraint systems. But this framework does not support algebraic types with different constraints for the constructors, as we need it in our case.

Hughes, Pareto and Sabry [12] describes a very similar system using Presburger arithmetic. But besides using a different constraint there are further essential differences. They do not have arbitrary indices, but the indices are always an upper bound for the number of constructors, whereas we are more flexible to choose the semantics, as we saw in the matrix-list examples. They use induction on the index to show the type correctness of a recursive definition, where we basically use induction on the recursion depth. As a result of this we can type the `reverse` version with accumulating parameter, but on the other side, we might infer an "incorrect" type for a non-terminating or always failing (in pattern-matching) function.

[4] At the time of publication we now use the more general constraint system of quantified boolean formulas over polynomial equations. We think that for this system we have complete type inference.

8. Conclusion

We presented a type system in which we can describe types in more detail using indices. This allows us to detect more type errors at compile-time, for example the incompatibility of sizes in a matrix multiplication. We believe that this extension could be integrated into the type systems of popular functional languages like Haskell or ML:

In recent languages with type classes [11, 13, 29] we can declare the more general type

```
matmult :: Matrix Int -> Matrix Int -> Matrix Int
determinant :: Matrix Int -> Int
```

thereby allowing matrix multiplication and determinant for matrices whose elements are from rings other than the integers. Indexed types should combine well with overloading calculi such as type classes and type declarations like the following look natural.

```
data Num a => Vector a n =
Vnil, n = 0 |
   Vcons a (Vector a m), m + 1 = n
```

However, we cannot expect to describe `(Matrix a n m)` as an instance of `Num a`, because we cannot add or multiply any two matrices. But we could add and multiply square matrices of a fixed size. Whether such an instance relation can be specified is to be investigated.

As already indicated in the section on limitations, the combination of dependent types and indexed types is interesting. The approach with `Int` and `Int'` indicated in the section on dependent types has to be pursued further.

Since the writing of the paper we extended this to quantified boolean formulas over polynomial equations, which have the advantage that we seem to get complete type inference using implication instead of *mgc*. Furthermore we get rid of the elimination condition using existential quantification over constraints. It seems promising to examine also other constraint systems, e.g. Presburger arithmetic. Finite sets with subset and element relation to parameterize the polynomial type with the set of variables could also be interesting. The overall goal here should be to understand the connection between constraint systems and algebraic types in more generality.

At the time of publication we have implemented a type checker for a slightly more sophisticated system than the one presented here. We use a very simple heuristic constraint solver, that works very well with our simple examples. We still have to see, whether it is feasible to use Gröbner bases computations which are known to be very costly, for the type inference of larger examples. We are confident that it will be feasible, because in practice most constraints will be linear.

Acknowledgements

We thank the anonymous referees for their valuable comments on the paper.

References

[1] B. Buchberger, Gröbner Bases: An algorithmic method in polynomial ideal theory, in: N.K. Bose (Ed.), Recent Trends in Multidimensional Systems Theory, D. Reidel, Dordrecht, 1985, pp. 184–232.

[2] R. Bird, P. Wadler, Introduction to Functional Programming, Prentice-Hall, Englewood Cliffs, 1988.

[3] T. Becker, V. Weispfenning, Gröbner Bases, Springer, Berlin, 1993.

[4] R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, J.T. Sasaki, S.F. Smith, Implementing Mathematics with the Nuprl Proof Development System, Prentice-Hall, Englewood Cliffs, NJ, 1986.

[5] C. Cornes, J. Courant, J. Filiâtre, G. Huet, P. Manoury, C. Munôz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, B. Werner, The Coq Proof Assistant Reference Manual. INRIA-Rocquencourt – CNRS – ENS Lyon, 5.10 ed., 1995.

[6] D. Cox, J. Little, D. O'Shea, Ideals, Varieties, and Algorithms, Springer, Berlin, 1992.

[7] G.E. Collins, Quantifier elimination for real closed fields by cylindric algebraic decomposition, in: 2nd GI Conf. on Automata Theory and Formal Languages, Lecture Notes in Computer Science, vol. 33, Springer, Berlin, 1975.

[8] T. Coquand, An algorithm for type-checking dependent types, preprint, March 1996.

[9] L. Cardelli, P. Wegner, On understanding types, data abstraction, and polymorphism, ACM Comput. Surveys 17 (1985) 471–522.

[10] L. Damas, R. Milner, Principal type-schemes for functional programs, in: Proc. 9th POPL, 1982, pp. 207–212.

[11] P. Hudak, S.P. Jones, P. Wadler, editors. Report on the Programming Language Haskell Version 1.2, vol. 27. ACM SIGPLAN, May 1992.

[12] J. Hughes, L. Pareto, A. Sabry, Proving the correctness of reactive systems using sized types, in: ACM Symp. Principles of Programming Languages, January 1996.

[13] M.P. Jones, An introduction to Gofer, 1994.

[14] M.P. Jones, Qualified types: theory and practice, Ph. D. thesis, University of Nottingham, 1994.

[15] R.D. Jenks, R.S. Sutor, AXIOM, Springer, Berlin, 1992.

[16] G. Kreisel, J.L. Krevine, Elements of Mathematical Logic, North-Holland, Amsterdam, 1967.

[17] A.J. Kfoury, J. Tiuryn, P. Urzyczyn, Type reconstruction in the presence of polymorphic recursion, ACM Trans. Programming Languages and Systems 15 (April 1993) 290–311.

[18] K. Läufer, Polymorphic type inference and abstract data types, Ph. D Thesis, Department of Computer Science, New York University, July 1992.

[19] Z. Luo, R. Pollack, LEGO proof development system: user's manual, Department of Computer Science, University of Edinburgh, 1992.

[20] R. Milner, A theory of type polymorphism in programming, J. Comput. System Sci. 17 (1978) 348–375.

[21] A. Mycroft, R.A. O'Keefe, A polymorphic type system for Prolog, Artificial Intelligence 23 (1984) 295–307.

[22] J. Mitchell, G. Plotkin, Abstract types have existential type, ACM Trans. Programming Languages 10 (1988) 470–502.

[23] M. Odersky, K. Läufer, Putting type annotations to work, in: Proc. 23rd ACM Symp. Principles of Programming Languages (January 1996) pp. 65–67.

[24] N. Perry, The implementation of practical functional programming languages, Ph. D. Thesis, Imperial College of Science, Technology, and Medicine, University of London, 1990.

[25] S. Peyton-Jones (Ed.) Implementation of Functional Languages, Prentice-Hall, Englewood Cliffs, NJ, 1990.

[26] M. Sulzmann, Typinferenz mit Constraints (German), Master's Thesis, University of Karlsruhe, Mai 1996.

[27] D.A. Turner, Recursion equations as a programming language, in: Darlington et al. (Ed.) Functional Programming and Its Applications, Cambridge Univ. Press, Cambridge, 1982.

[28] P. Wadler, Deforestation: Transforming programs to eliminate trees, Theoret. Comput. Sci. 73 (1990) 231–248.

[29] P. Wadler, S. Blott, How to make ad-hoc polymorphism less ad-hoc, in: Proc. 16th ACM Symp. Principles of Programming Languages, 1989, pp. 60–76.