Non-uniform Recursion: The solution (minimal sorting for `fold`)

Martin Wehr
University of Edinburgh, Department of Computer Science (LFCS)
Mayfield Road, Edinburgh EH9 3JZ, Scotland U.K.
`email: wehr@dcs.ed.ac.uk web: http://www.ed.ac.uk/home/wehr`

March 25, 2000

Abstract

A central data structure in functional programming languages like ML or Haskell are algebraic data types. I argue in this paper that the natural program structure to deal with algebraic data types is not yet provided either in Haskell or in ML. One might think that that pattern matching together with general recursion is the natural program structure to deal with algebraic data types. But the use of general recursion runs into trouble when faced to deal with non-uniform recursive data through fundamental problems of type inference (the non-decidability of polymorphic recursion).

What we provide for algebraic data types is a structural recursion mechanism that derives generalised fold functions from algebraic data specification. Especially non-uniform folds are an byproduct of these more general folds. The structural recursion mechanism is presented together with a typing method which relies on a method of minimal sorting for algebraic data specifications.

1 Introduction

A central data structure in functional programming languages like ML or Haskell are algebraic data types. I argue in this paper that the natural program structure to deal with algebraic data types is not yet provided either in Haskell or in ML. The reader might find as it is commonly held that pattern matching together with general recursion is the natural program structure to deal with algebraic data types. But the use of general recursion is a problem when faced to deal with non-uniform recursive data. This limitation will be lifted by our mechanism for structural recursion.

I argue that instead of *general* recursion a notion of *structural* recursion is the natural program structure. The situation is similar in dealing with control flow the goto statement offers a general method but structured ways of handling control flow are to be preferred. What we provide for algebraic data types is a structural recursion mechanism that is parametrised by the cases and the types of algebraic data specifications. The mechanism *derives* generalised fold functions from algebraic data specification.

Algebraic data specifications allow more than just recursive type definitions but also recursive *type constructor* definitions called non-uniform recursive types. Attempts to write non-uniform fold functions run into trouble through the violation of the mono-recursion restriction of the type inference system. So a programmer could write non-uniform fold function which would implement the algorithm, but the type inference of Haskell or ML would not accept the program. An result of

Henglein [Hen93] tells that this typing problem cannot be overcome because of the non-decidability of poly-recursion. I will provide a minimal sorting algorithm which allows to compute the types for non-uniform fold. This type computation is possible because of the restriction from general to structural recursive algorithms.

The aim of this paper is threefold : first we want to define the structural recursion construct; second we want to illustrate its use by examples; third we want to formulate a general theory underlying the derivation of program structure from algebraic data types.

2 The Problem

In this section I will explain why general recursion is a problem in languages like Haskell or ML. Especially I will argue that general recursion is not the natural method to deal with algebraic data types. Consider the following algebraic data specification :

```
data List a = Cons  (a, List a)        | Nil
data Bush a = Bcons (a, Bush (Bush a)) | Bnil
```

The specification defines two types `List a` and `Bush a`. The first line is a recursive type definition which is indicated by the recursive use of `List a` (the type to be defined) on the right hand side. We say a type definition is *simple* recursive if the type occurring on the right hand side are either copies of the left hand side `List a` or simple type variables (like `a`). Note that in fact a type constructor has been defined because `List` takes a type argument and produces a type (for instance `List int` or `List bool`).

The second line is a recursive definition of a type constructor indicated by the use of the constructor `Bush` on the right hand side. This is an instance of non-uniform recursion. We say a type definition is non-uniform recursive if types occur on the right hand side which are not copies of the left hand side (like `Bush (Bush a)`).

Recursive Types

The difference between lists and bushes shows up when we attempt to program the canonical recursion construct for these types. The fold function for list is well known to functional programmers :

```
Lfold cs nl Cons(a, la) = cs (a, Lfold cs nl la)
Lfold cs nl Nil         = nl
```

This function can be used for instance to sum the elements of a list of numbers by the call `Lfold + 0 [1,2,3]`. Similarly to multiply the elements of a list we would call `Lfold * 1 [1,2,3]`.

Recursive Type Constructors (Non-uniform Recursion)

With a bit of experience the fold function for bushes can be defined :

```
Bfold bc bn Bcons(a, bba) = bc (a, Bfold bc bn
                                   (Bmap (Bfold bc bn) bba))
Bfold bc bn Bnil          = bn
  where
    Bmap f Bcons(a, bba)  = Bcons(f a, Bmap (Bmap f) bba)
    Bmap f Bnil           = Bnil
```

Suppose we are given a bush with integer entries. The fold can be used again to sum over all elements in a bush by calling `Bfold + 0 bush`.

The point of this example is that it *needs* an experienced programmer to write this basic basic function for structural recursion. A beginner in functional programming would struggle to develop this code. Maybe this is the cause why the expressiveness of non-uniform recursion has hardly been explored since lately [Oka98].

We might interpret the above definition as a term rewrite system by directing the equation from left to right. In doing so we get a operational semantic for the functions `Bfold Bmap`. Studying these functions one sees that termination for finite bushes and the subject reduction property are valid (i.e. if the arguments of the functions are well typed the result of the computation will be again well typed) [Kah96].

Polymorphic Recursion

The problem is that the type inference of Haskell or ML would not accept the definition. Please recall that the definition of `Bush a` involves twice the use of the type constructor `Bush` on the right hand side. This is repeated in the definition of the function `Bfold` by calling the function twice on the right side, similarly in definition of `Bmap`. In the definition of `Bush a` the two occurrences of `Bush` differ by having either the type variable `a` or the type `Bush a` as argument. Similarly in the definition of `Bfold` the recursive calls differ in type. The function at place a has type `Bush (Bush a) -> ..` while the function at place b has type `Bush a -> ...`

$$\begin{aligned} \texttt{..}\quad\texttt{=}\quad & \texttt{bc (a,}\overbrace{\texttt{Bfold bc bn}}^{a} \\ & \qquad\texttt{(Bmap (}\underbrace{\texttt{Bfold bc bn}}_{b}\texttt{) bba))} \end{aligned}$$

This typing problem cannot be solved in general. Henglein [Hen93] proved that polymorphic typing for general recursive functions is undecidable. This result says you cannot have both: on the one hand definitions of polymorphic functions by general recursion (such that the function to be defined occurs on the right hand side of the definition in different instances (like `Bfold`)) and on the other hand type inference for such definitions.

So a functional programmer would annotate these function definition by types:

```
Bmap  :: (a->b) ->             (Bush a -> Bush b)
Bfold :: ((a,a) ->a)  -> a -> (Bush a -> a)
```

A second look at these types tells that the fold function could be given a more flexible type. To do so we must leave the realm of first order polymorphism and enter rank-2 polymorphism with quantification over type constructors.

```
Bfold :: forall b:*->*.
           forall a. (a,(b (b a)) ->b a) ->
           forall a. (b a)                ->
             forall a. (Bush a -> b a)
```

There exists a versions of Haskell [Jon98] which support type checking for this type discipline.

3 The solution

We have seen that the use of general recursion runs in difficulties when attempting to program with non-uniform data types. Here I propose a solution which offers structural recursion derived from algebraic data specification.

The above analysis suggests a solution for non-uniform recursion. One has just to write a preprocessor reading the algebraic data specifications and generating the program texts defining the map and fold function with type annotations. This generated program text can then be passed to a Haskell compiler. This technique would relieve the programmer from the burden of writing fold functions for every data type.

I believe that structural recursion construct is the natural program structure to deal with algebraic data types. Therefore it should be embedded in a more general way into functional programming languages like Haskell or ML. I propose that a functional programmer should deal with two kinds of entities :

functional These are the values and functions used to date. These entities are all first order polymorphically typed.

algebraic This new kind of *algebraic* entities will be generated by the structural recursion construct. To use an algebras in a program it has to be applied to an instantiation type. Such an application converts an algebra to a function which takes arguments of the instance type and produces results of a reduced type (like summation reduces lists of integers to integers). The reduction of the instantiation type is an essential part of an algebra. The following examples illustrates how to use these algebras.

3.1 Examples

We start our examples with the familiar list types.

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

The definition consists of the two cases `Cons` and `Nil`. Recall that the fold function needs an argument for each of these cases (for example the two arguments `+` and `0` to define summation on lists by `Lfold + 0`). A definition of an algebra accepts an argument for each case of an algebraic data type. I suggest a new notation for structural recursion in order to define algebras. The previous example should read like :

```
sum = <Cons => + | Nil => 0>
```

Note that `sum` is an algebra in order to instantiate an algebra it can be applied like `sum [1,2,3]`. But we can use an algebra more generally by instantiating with a type :

```
double_sum = sum <inst:  List List>
```

The above example would return a function of type `List(List int) -> int` summing over the elements of a list of list of integers.

Reducing Lists

Look at the following examples to get more familiarity with the use of algebras.

```
l '++' l' = <Cons => Cons | Nil => l'> l
flatten   = <Cons => ++>
```

For the first definition we used the Haskell convention to write the append combinator in infix notation. The first case `Cons => Cons` is redundant. So we suggest as syntactic sugar to write the right hand side in short as `<Nil => l'> l`. In the second definition we adopted this convention for a very short definition of flatten. In functional programming flatten is a mere function with type `List(List a) -> List a`, but the above definition gives an algebra. So we could instantiate this algebra to get variants of flatten :

```
flatten <inst: List List>  :: List (List (List a)) -> List a
```

Using the notation <List> similar to the fold function is suggestive as syntactic sugar. So informally this might be considered like a function :

```
<List> ::  ((a, b) -> b) -> b -> (List a -> b)
```

Of course it is *not* a function but an algebra which takes two arguments associated to the cases and can be instantiated by a type. So the following equation of algebras is valid :

```
<List> + 0 = <Cons => + | Nil => 0>
```

We want to finish our examination of lists by a definition of a filter on lists.

```
filter = <List> <inst:  (List, List)> (divide test) ([], [])
   ::  (List a) -> (List a, List a)
```

Where we used divide as a helping function to divide list elements to the left or right side of a tuple depending on a predicate.

```
divide pred (a, (leftlist, rightlist)) = if (pred a)
   then ([a :  leftlist], rightlist)
   else (leftlist, [a :  rightlist])
```

Reducing multiple Type Constructors

In the previous examples we considered algebras which reduced only one type constructor namely List. In the following example we define an algebra which reduces lists and trees.

```
data Tree a = Br (Tree a, Tree a) | Leaf a | Bot

id   x = x
reduce = <List, Tree> ++ Nil ++ id Nil <inst: List(Tree(List))>
           :: List(Tree(List(List a))) -> List a

transform = < Cons ta a => Br(Leaf a, ta)
            | Nil       => Bot
            | Br        => ++
            | Leaf a    => [a]
            | Bot       => [] ><inst: List(Tree(List))>
              :: List(Tree(List a)) -> Tree(List(Tree a))
```

Note that by convention absent cases are assumed to be trivial so we assume the that the cases <Bcons => Bcons | Bnil => Bnil> live in the definition of the algebra transform. So the following instantiation is valid :

```
transform <inst:  Bush List> ::  Bush(List a) -> Bush(Tree a)
```

3.2 Operational Semantics

The presentation so far raises then question about how an algebra computes. To be precise we have to ask how an *instantiated* algebra which belongs in the functional realm computes. Suppose we are given an algebra f = < .. | C => e_C | ..>. In order to get a function we have to instantiate the algebra by a type <inst: typ>. If we apply this function to a well typed value v of type typ the computation should result in a well typed value of reduced type (see section 4.2 for the definition of the reduced type). We will define a term reduction system which transforms in each step the instantiation type `typ` and the value `v`. The term reduction system consists of three cases :

```
f <inst: typ> v  is computed by ...
 (empty)        f <inst: > v
                   → v
 (tuple)        f <inst:(t1, ... , tn)> (v1, ... , vn)
                   → (f <inst: t1> v1, ... , f<inst: tn> vn)
 (constructor)  f <inst:T typseq> C(v)
                   → e_C(f <inst: argtyp> v)
```

In order to understand the last case **(constructor)** we have to explain the sequence of types `typseq` and how to construct the type `argtyp`. To do so we observe that the data constructor `C` is a case in the definition of the type constructor `T`.

```
data T a1 ... ak = ... | C (t1, ... ,tm) | ...
```

We assume that type constructor `T` takes k type arguments. So the argument `typseq` is indeed a sequence of k types $\texttt{typseq}_1 \dots \texttt{typseq}_k$. Observe that the type variables `a1 ... ak` can occur in the argument type `(t1, ... ,tm)` of the data constructor `C`. We define the type `argtyp` by substituting each type variable `ai` occurring in `(t1, ... ,tm)` by the type $\texttt{typseq}_i$ for each $i \leq k$.

We assumed that the value `C(v)` is well typed of type `T typseq`. This well-typing implies that the value `v` is well typed of type `argtyp`. A similar argument validates the subject reduction in case **(tuple)** . The **(empty)** case is the basis for the structural inductive definition of fold. In the base case the type should not be reduced and subject reduction is trivial. From the above argument we see that the term reduction system associated to the instantiated algebra `f<inst: typ>` satisfies subject reduction. For details please have a look at theory appendix.

4 Minimal Sorting

To each value of an algebraic data type exists a minimal typing calculated by the type inference algorithm. Here I will present an algorithm which gives a minimal sorting for each algebraic data specification. We will employ this minimal sorting to compute a typing for algebras.

The basic idea is very simple. Consider the following variants of lists. (see also Bird/Meertens paper [BM98] on these types)

```
data List     a = Cons  (a, List a)         | Nil
data Switch a b = Idcons (a, Switch a b)    | Swnil
                | Swcons (b, Switch b a)
type Pair     a = (a, a)
data Nest     a = Ncons (a, Pair (Nest a)) | Nnil
data Bush     a = Bcons (a, Bush (Bush a)) | Bnil
```

We want to assign sorts to the arguments and result of the type constructors. We will start in most generality and then take all constraints into account to compute the sorting. So we assume a sort for each result of type constructors which gives the set {`List.res, Switch.res, Pair.res, Nest.res, Bush.res`}. Similar we get sorts for the type constructor arguments {`List.1, Switch.1, Switch.2, Pair.1, Nest.1, Bush.1`}. So we introduced an obvious naming convention for arguments and results.

We have two ways to introduce constraints on these sorts.

left-hand right-hand coincidence means that if a type variable occurs on the right side of a data definition then it must coincide with the corresponding type argument of the left hand side. For instance in the definition of the `Switch` the variable `b` (that is the second argument of the left hand side so `Switch.2`) occurs in the case `Swcons` at the first argument of the type constructor inducing the constraint `Switch.2 = Switch.1`.

argument result coincidence means that if a type constructor takes another type constructor as argument in the right hand side of a data definition, then the sorts have to coincide. For instance at the right hand side of definition `Nest` the `Pair` constructor takes `Nest` as argument inducing the constraint `Pair.1 = Nest.res`.

Collecting all the constraints and calculating the reflexive transitive closure we get an equivalence relation on the arguments and results of the type constructors. The equivalence classes of this relation will be from now on called the sorts.

For example we get the two equivalence classes $\mathtt{a} = \{\mathtt{List.1}\}$ and $\mathtt{b} = \{\mathtt{List.res}\}$. So the minimal sorting for lists is $\{$`List a -> b`$\}$ as expected. We complete the example by listing all the sorts of the above mentioned definition.

$$\begin{array}{lcl} \mathtt{c} & = & \{\mathtt{Switch.1}, \mathtt{Switch.2}\} \\ \mathtt{d} & = & \{\mathtt{Switch.res}\} \\ \mathtt{e} & = & \{\mathtt{Nest.1}\} \\ \mathtt{f} & = & \{\mathtt{Nest.res}, \mathtt{Pair.1}\} \\ \mathtt{g} & = & \{\mathtt{Pair.res}\} \\ \mathtt{h} & = & \{\mathtt{Bush.res}, \mathtt{Bush.1}\} \end{array}$$

The result can be seen as a sorting for the type constructor signature.

$\{$ `List a->b, Switch c c -> d, Nest e->f, Pair f->g, Bush h->h` $\}$

4.1 From sorted Signatures to fold Types

From the sorted signature of type constructors we can immediately read of the types for the fold functions. For instance we can see now how to derive the types for the fold of bushes (recall section 2).

Reductive Type

```
<Bush> :: ((h,h) ->h)  ->      -- Bcons
                   h  ->      -- Bnil
            (Bush h -> h)
```

The type above is still a polymorphic type where the type variable `h` is supposed to be all quantified in front. We will call this the *reductive* type to a fold function. This type is formed by taking a type variable for each sort involved. Then for each case (here `Bcons` and `Bnil`) the associated sorted function type is used. Finally the result function type (here `Bush h -> h`) is just the sorted function type associated to the sorting of the type constructor.

Parametric Type

```
<<Bush>> :: forall b:*->*.
           forall h. (h, b (b h)) ->b h) ->  -- Bcons
           forall h.               (b h) ->  -- Bnil
             forall h. (Bush h -> b h)
```

We will call this the *parametric* type to a fold function. This type is build by quantifying over a type constructor variable (here `b`) for each involved type constructor (here `Bush`). For each case (here `Bcons` and `Bnil`) a case type is used. For instance the case type associated to `Bcons` is :

```
forall h.  (h, b (b h)) ->b h)
```

So a case type a locally bound function type (here `h` is the locally bound type variable). Note that case types are build by substituting the type constructors by its associated type constructor variable (here `Bush` is substituted by `b`) Finally the result function type

```
forall h.  (Bush h -> b h)
```

is a locally bound function type reducing the type constructor `Bush` to its associated type constructor variable `b`

Example: `Nest`

A look at the following example illustrates how to derive fold types from the sorted type signature. We recall the relevant part of the data specification and sorted signature.

```
type Pair     a = (a, a)
data Nest     a = Ncons (a, Pair (Nest a)) | Nnil

     { ...  , Nest e->f, Pair f->g, ...  }
```

The reductive type for `Nest`:

```
<Nest> :: ((f,f) -> g)      ->  -- Pair
          ((e,g) -> f)      ->  -- Ncons
                   f        ->  -- Nnil
             (Nest e -> f)
```

The parametric type for `Nest`:

```
<<Nest>> :: forall p:*->*  n:*->* .
    forall f. (f,f)       -> p f)  ->  -- Pair
    forall e. (e, p(n e)) -> n e)  ->  -- Ncons
    forall e.               (n e)  ->  -- Nnil
      forall e. Nest e -> n e
```

4.2 Computing the Type of an Algebra

Here we want to address the question of well typing for algebras. So given an algebraic specification $\mathcal{A}$, when is an algebra like `< ..  | C => e`$_C$ `| ..>` well typed ? Recall that the minimal sorts for the specification $\mathcal{A}$ are equivalence classes. We will call a class which contains a type constructor result (like `List.res`) a result sort. The programmer has to specify for each result sort whether it is to be handled reductive or parametric. Above we introduced already a notation for reductive `<Bush>` versus parametric `<<Bush>>` algebras.

So we assume that an algebra is specified like

`f = <<..  | C => e`$_C$ `| ..> ..  | C' => e`$_{C'}$ `| ..>`

to a algebraic data specification

$$\mathcal{A} = \begin{cases} \quad\vdots & & & \vdots \\ \texttt{T ..} & \texttt{=} & \texttt{..} & \texttt{| C (t1, .. , tn) | ..} \\ \texttt{T' ..} & \texttt{=} & \texttt{..} & \texttt{| C' (t1', .. , tm') | ..} \\ \quad\vdots & & & \vdots \end{cases}$$

such that `T.res` lives in the class to a parametric result sort `a` and `T'.res` lives in the class to a reductive result sort `b`.

The functional entities $\texttt{e}_\texttt{C}$ associated to the cases should be of course well typed by a type `typC` in the polymorphic type discipline. The algebra `f` is well formed depending on each case :

parametric In this case an arity $n_\texttt{a} \in \mathbb{N}$ is associated to the parametric sort `a`. For each type constructor (for example `T`) in class `a` there is a type `typT` which should be a cartesian product of that length $\texttt{typT = (typT}_1\texttt{, .. ,typT}_{n_\texttt{a}}\texttt{)}$. For each case of this sort (like `C`) the type `typC` should coincide with type `typT` of its associated type constructor.

reductive In this case there is one type `typb` associated to the reductive sort `b`. For each case of this sort (like `C'`) the type `typC'` should coincide with type `typb`.

We say a type constructor `T` occurs in an algebra `f` if a case `C` of the that type constructor occurs in the algebra `f = < .. | C => .. >`. An algebra `f` which is well formed in the above sense associates a type (i.e. `typT` or `typT'`) to each type constructor (here `T` or `T'`) which occurs in `f`. We extend this map to a total function f^{typ} from all type constructors of the specification $\mathcal{A}$ to types by mapping non-occurring type constructors `T''` to the type `T'' a1 ... an` (filling in the sorts for this type constructor).

Given an instantiation type `<inst: typ>` we define a reduced type `rtyp` by substituting all the type constructors `T` occurring in `typ` by the type f^{typ}(`T`). This gives the function type of the instantiated algebra :

```
f <inst: typ> :: typ -> rtyp
```

For the details see the theory appendix.

5 Conclusion

Previous Work

The present paper is clearly inspired by Birds, de Moors [Bir87] work toward a algebra of programming. The notion of algebra which is introduced in this paper has parallels with the methodology of programming with bananas and lenses [MFP91, MH95]. The above work does address only simple recursive data types. It would be fruitful to extend the present work along suggestive directions from this work namely to look for dual recursion constructs like unfold, recursion constructs for algebraic data type with function types and finally general transformation laws. The work of Moggi et al. [BJM96] on functorial ML has similarities to the present work in a typing discipline which allows type constructor parameters. But still this work does not deal with non-uniform recursion. The work on polytypic programming promotes a programming method which is parametric over types [Mee96, JJ96]. Again non-uniform recursion is not yet included in polytypic programming. Recall that the algebras of the present work can be instantiated by types, so they can be seen as first non-uniform polytypic programs. The paper on nested data types [BM98] addresses the problem of non-uniform recursion by giving a functor categorical semantics. The notion of parametric and reductive folds are borrowed from this source. The proposed methodology of algebras and minimal sorting seems to be an original result of the present paper.

Present Work

I have presented a method to embed general structural recursion into functional programming languages like Haskell or ML. I proposed a formalism to specify new entities called *algebras*. An algebra can be instantiated with any type in order to create generalised fold functions. Especially non-uniform folds which have been considered as a problem are an byproduct of this general approach. The use of this new program construct is illustrated by many examples.

The proposed method is presented together with a typing mechanism and an operational semantics which validates subject reduction. The typing mechanism consists first in a minimal sorting for the type constructors of an algebraic data specification. In a second step the typing of algebras is read of from the sorting of the type constructors. The programmer can choose reductive or parametric typing.

Future Work

This paper is one step in a journey toward a new programming discipline of higher dimensional programming (like functional programming or logical programming are programming disciplines).

In the theory appendix we mention that algebraic data types and the associated structural recursion live in dimension two of a hierarchy of languages. Type inference with a minimal typing property has already been observed for the languages of this hierarchy [Weh98]. A conceptually simple categorical characterisation as iterated free monoid object construction in monoidal categories has been found for this hierarchy [Weh99]. The next steps will be the development of general structural recursion for the hierarchy in style of this paper, and the development of higher dimensional parametricity to derive program transformations similar to the analysis for system F [Wad93, BFSS90, PA93].

I am currently working on applications exploring the expressive power of dimension three. Preliminary results deal with simply typed lambda calculus, internalisation of algebraic data specifications and examples from Okasakis purely functional data types [Oka98]. A challenge is to internalise unification, type inference and the operational semantic for structural recursion.

The work in dimension three lead to an exiting observation that properties like balancing for trees can be encoded in types such that well typing implies satisfaction of these properties and subject reduction implies invariance of these properties. This opens a road where type inference is employed for program verification. The goal of establishing a higher dimensional programming discipline is exactly the point to banish the barrier between programming and verification.

References

[BFSS90] E. S. Bainbridge, P. J. Freyd, A. Scedrov, and P. J. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70:35–64, 1990.

[Bir87] Richard S. Bird. *An introduction to the theory of lists*, volume F36 NATO ASI Series of *Logic of Programming and Calculi of Discrete Design*, pages 5–42. Springer-Verlag, 1987.

[BJM96] G. Bellè, C.B. Jay, and E. Moggi. Functorial ML. In K. Herbert and S.D. Swierstra, editors, *Programming Languages: Implementations, Logics, and Programs*, volume 1140 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996. TR SOCS-96.08.

[BM98] Richard Bird and Lamber Meertens. Nested datatypes. In 4^{th} *Int. Conf. Mathematics of Program Construction*, Marstrand, Sweden, June 98, June 1998. LNCS 1422.

[Hen93] Fritz Henglein. Type inference with polymorphic recursion. *TOPLAS*, 15(2):253–286, April 1993.

[JJ96] Johan Jeuring and Patrik Janson. Polytypic programming. In J. Launchbury, Erik Meijer, and T. Sheard, editors, *Advances in Functional Programming*, pages 68–111, Olympia, WA, USA, 1996. Springer Verlag. LNCS 1129.

[Jon98] Mark Jones. The hugs98 user manual. `http://www.cse.orgi.edu/PacSoft/projects/pages/hugsman/index.html`, 1998.

[Kah96] Stefan Kahrs. Limits of ML-definability. In H. Kuchen and S. Swierstra, editors, *Proceedings of PLIP '96*, pages 17–32, Aachen, September 1996. LNCS 1140.

[Mee96] Lambert Meertens. Calculate polytypicaly! In H. Kuchen and S. Swierstra, editors, *Proceedings of PLIP '96*, pages 1–16, Aachen, September 1996. LNCS 1140.

[MFP91] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes, and barbed wire. In John Hughes, editor, *Proceedings Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA*, pages 124–144. ACM, Lecture Notes in Computer Science 523, August 1991.

[MH95] Erik Meijer and Graham Hutton. Bananas in space: Extending fold and unfold to exponential types. In *FPCA '95*, pages 324–333, La Jolla, California, June 1995. ACM.

[Oka98] Chris Okasaki. *Purely Functional data structures*. Cambridge University Press, 1998.

[PA93] Gordon Plotkin and Martin Abadi. A logic for parametric polymorphism. In *Typed Lambda Calculi and Applications TLCA '93*, pages 361–375, LNCS 664, March 1993. Springer Verlag.

[Wad93] Philip Wadler. Theorems for free! In *Proc. of FPCA '89*, pages 347–359. ACM, 1993.

[Wag94] Eric G. Wagner. *Handbook of Logic in Computer Science*, volume 3, chapter Algebraic Semantics, pages 323–393. 1994.

[Weh98] Martin Wehr. Higher order algebraic data types (preprint). Technical report, University of Edinburgh, URL `http://www.dcs.ed.ac.uk/home/wehr`, July 1998.

[Weh99] Martin Wehr. Higher dimensional syntax. In *CTCS 99*, Edinburgh, Scotland, September 1999.

6 Apendix: Theory

In previous work I have developed a hierarchy of languages. At dimension zero live words build by an alphabet of letters. At dimension one live sorted terms. At dimension two live algebraic data types. This hierarchy continues for any dimension $n \in \mathbb{N}$. I have already presented a type inference for minimal typing at each dimension and the parametric fold and map constructs [Weh98, Weh99]. In this paper I will have a closer look at the morphisms of this language.

Each language in this hierarchy comes with a notion of signature, free language construction on this signature and language morphisms. In the following I will present these notions. Finally we will see that the general fold is just a two dimensional morphism.

6.1 Dimension Zero

A signature of dimension zero is just a set S which might also be called the alphabet and the elements might be called letters. The words $u, v \in S^*$ of the free language are build by juxtaposition of letters. So algebraically S^* has the structure of a monoid. The freeness of this monoid is expressed in the fact that each monoid homomorphism $f : S^* \to T^*$ is induced by a function $f^\flat : S \to T^*$. In the following we will denote by u_i the i-th letter in word u. By $u_{<i}$ we will denote the prefix before position i and by $|u|$ we will denote the length of the word.

6.2 Dimension One

6.2.1 Signature (dim = 1)

Given a set S of sorts we define a S-sorted signature Σ. Such a signature is a tuple $\Sigma = (C, a : C \to S^* \times S)$ consisting of a set C of constructors and an arity functions a. So any constructor contained in the signature Σ will be denoted as $g : (u, s) \in \Sigma$ with its sorting (u, s) consisting of an input word u and an letter s for the output sort.

6.2.2 Language (dim = 1)

We define by induction the free language to a signature. The "words" of this language are the tuples of well-sorted terms formed with the constructors of the signature and sorted variables. Algebraically this is the free cartesian category to a signature. In the literature [Wag94] the names algebraic theories or Lawvere theories are also used. We define the terms through a type system where the statement $u \vdash_\Sigma t : v$ means that term t is well-formed. We will use a de Bruijn variable naming convention. So given a word $u \in S^*$ we will assume the existence of a sorted variable $x_i : u_i$ to each position i in the word. The terms are defined inductively by three cases.

$$\textbf{(var)}\ \frac{}{u \vdash_\Sigma x_i : u_i} \qquad \textbf{(juxt)}\ \frac{u \vdash_\Sigma t^1 : v^1 \quad u \vdash_\Sigma t^2 : v^2}{u \vdash_\Sigma t^1 t^2 : v^1 v^2}$$
$$\textbf{(constr)}\ \frac{u \vdash_\Sigma t : v \qquad g : (v, s) \in \Sigma}{u \vdash_\Sigma g(t) : s}$$

Note that the statement $u \vdash_\Sigma t$ says that the variables $x_i : u_i$ might occur in term t. The statement $t' : u$ tells that t' is a sequence of singelton terms $t'_i : u_i$. Observe that the sort for variable x_i and singelton t'_i coincide. We get a new well sorted term if we subsitute all occurences of variables in term t (say x_i) by the associated singelton of sequence t' (here t'_i). We will denote the result of this substitution by $t \bullet t'$

6.2.3 Morphism (dim = 1)

We will use Σ^{cc} as a short notation for the collection of well formed terms. Assume we have a S-sorted signature Σ and a T-sorted signature Σ'. We want to understand the notion of language morphism $f : \Sigma^{cc} \to \Sigma'^{cc}$. Similar to the zero dimensional case we expect that by freeness the morphism is induced by a simpler (mere signature morphism) $f^\flat : \Sigma \to \Sigma'^{cc}$. This signature mophism $f^\flat$ should validate a sort compatibility condition on an underlying sort morphism $f^{srt} : S^* \to T^*$. This compatibility condition says that the signature morphism maps each constructor $g : (v, s) \in \Sigma$ to a well sorted term

$$f^{srt}(v) \vdash_{\Sigma'} f^\flat(g) : f^{srt}(s)$$

Given such a signature morphism $f^\flat$ We define the language morphism by structural induction on the three cases :

the mapping $\quad u \vdash_\Sigma t : v \mapsto f^{srt}(u) \vdash_{\Sigma'} f(t) : f^{srt}(v) \quad$ is defined by ...

(var)	$u \vdash_\Sigma x_i : u_i$	$\mapsto$	$f^{srt}(u) \vdash_{\Sigma'} x_{l+1} \dots x_{l+k} : f^{srt}(u_i)$
	where $l = \lvert f^{srt}(u_{<i}) \rvert$ and $k = \lvert f^{srt}(u_i) \rvert$		
(juxt)	$u \vdash_\Sigma t^1 t^2 : v^1 v^2$	$\mapsto$	$f^{srt}(u) \vdash_{\Sigma'} f(t^1) f(t^2) : f^{srt}(v^1 v^2)$
(constr)	$u \vdash_\Sigma g(t) : s$	$\mapsto$	$f^{srt}(u) \vdash_{\Sigma'} f^\flat(g) \bullet f(t) : f^{srt}(s)$

6.3 Dimension Two

Signatures of dimension two are generalisation of algebraic data types. They come with a type inference algorithm computing the minimal typing and a notion of morphism similar to the fold algorithm.

We will repeat the definition of dimension one with a slight generalisation. So signatures of dimension two will be defined using the notions of dimension one. Remember that the sorts (v, s) for a one dimensional signature consisted of a input-word v and an output-sort s. In this style we define a one-dimensional sort consisting of a input-term and an output-constructor.

$$\textbf{(sort)}\ \frac{u \vdash_\Sigma t \qquad u \vdash_\Sigma g(t')}{u \vdash_{\Sigma srt} (t, g(t'))}$$

6.3.1 Signature (dim = 2)

Given a one dimensional signature Σ we define a Σ-typed signature $\mathcal{A}$. Such a signature is a tuple $\mathcal{A} = (D, a : D \to \Sigma srt)$ consisting of a set D of data constructors and an arity functions a. So any data constructor contained in $\mathcal{A}$ will be denoted as $C : (t, g(t')) \in \mathcal{A}$ with its type $(t, g(t'))$ consisting of an input-type t and an output-type $g(t')$. The constructor g will be called type constructor For instance if the algebraic data type $\mathcal{A}$ defines lists there will the data constructors `Cons` and `Nil`. These constructors will have the typing `Cons:(a (List a), List a)`$\in \mathcal{A}$ and `Nil:( , List a)`$\in \mathcal{A}$ so `List` is the type constructor used in these cases.

6.3.2 Language (dim = 2)

We will again define the free language to a two dimensional signature. The "words" of this language are the well-typed values (like lists are values to an algebraic data type). We define values through a type system inductively by three cases :

$$\textbf{(var)}\ \frac{}{t \vdash_\mathcal{A} x_i : t_i} \qquad \textbf{(juxt)}\ \frac{t' \vdash_\mathcal{A} v^1 : t^1 \quad t' \vdash_\mathcal{A} v : t^2}{t' \vdash_\mathcal{A} v^1 v^2 : t^1 t^2}$$

$$\textbf{(constr)}\ \frac{t^2 \vdash_\mathcal{A} v : t^3 \qquad t^3 = t \bullet t^4 \qquad C : (t, g(t')) \in \mathcal{A}}{t^2 \vdash_\mathcal{A} C(v) : g(t') \bullet t^4}$$

Note that in a statement $t \vdash_\mathcal{A} v : t'$ we assume that the types t, t' are well-formed in the underlying one dimensional language formalised by $\vdash_\Sigma$. Further to define the constructor case **(constr)** we employed the composition $\bullet$ of dimension one. So in the constructor case the notion of instantiation for a polymorphic type system shows up.

6.3.3 Morphism (dim = 2)

Assume we have a Σ-typed signature $\mathcal{A}$ and a Σ'-typed signature $\mathcal{B}$. We want to understand the notion of morphism f between the associated free languages. We expect that by freeness the morphism is induced by a mere signature morphism $f^\flat : \mathcal{A} \to \mathcal{B}^{val}$. The signature morphism $f^\flat$ should validate a type compatibility condition on an underlying type morphism $f^{typ} : \Sigma^{cc} \to \Sigma'^{cc}$. This compatibility condition says that the signature morphism maps each data constructor $C : (t, g(t')) \in \mathcal{A}$ to a well typed value.

$$f^{typ}(t) \vdash_\mathcal{B} f^\flat(C) : f^{typ}(g(t'))$$

Given such a signature morphism $f^\flat$ We define the language morphism by structural induction on the three cases :

$$\text{the mapping} \qquad t \vdash_{\mathcal{A}} v : t' \mapsto f^{typ}(t) \vdash_{\mathcal{B}} f(v) : f^{typ}(t') \qquad \text{is defined by ...}$$

$$\begin{array}{lll} \textbf{(var)} & t \vdash_{\mathcal{A}} x_i : t_i & \mapsto \quad f^{typ}(t) \vdash_{\mathcal{B}} x_{l+1} \ldots x_{l+k} : f^{typ}(t_i) \\ & \quad \text{where } l = |f^{typ}(t_{<i})| \text{ and } k = |f^{typ}(t_i)| & \\ \textbf{(juxt)} & t \vdash_{\mathcal{A}} v^1 v^2 : t^1 t^2 & \mapsto \quad f^{typ}(t) \vdash_{\mathcal{B}} f(v^1) f(v^2) : f^{typ}(t^1 t^2) \\ \textbf{(constr)} & t \vdash_{\mathcal{A}} C(v) : g(t') & \mapsto \quad f^{typ}(t) \vdash_{\mathcal{B}} f^\flat(C) \bullet f(v) : f^{typ}(g(t')) \end{array}$$

The above definition of language morphism follows just the lines of the language definition. It is a straight forward argument by structural induction that well-forming is preserved by these morphisms. It is now obvious how this preservation of well-forming implies the subject reduction property for the operational semantics of algebras (defined in section 3.2). Given an algebraic data specification $\mathcal{A}$ and an algebra `f = < ..  | C => e`$_{\texttt{C}}$` | ..>` using the cases of the specification. From the algebra we can read of an endo signature morphism $f : \mathcal{A} \rightarrow \mathcal{A}^{val}$ mapping a data constructor `C` to a well typed value `e`$_{\texttt{C}}$. The well typeness of the algebra `f` induces the underlying type signature morphism $f^{typ\,\flat}$. Finally we can reformulate the operational semantics in the style of this section as :

$$\begin{array}{lll} \textbf{(var)} & f < x_i > v & \rightarrow \quad v \\ \textbf{(juxt)} & f < tt' > vv' & \rightarrow \quad (f < t > v)(f < t' > v') \\ \textbf{(constr)} & f < g(t'') > C(v) & \rightarrow \quad e_C(f < t \bullet t^3 > v) \\ & \quad \text{where } C : (t, g(t')) \in \mathcal{A} \text{ and } t'' = t' \bullet t^3 & \\ & \quad \text{and } f =< \ldots | C \Rightarrow e_C | \cdots > & \end{array}$$