Higher Order Algebraic Data Types

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

July 23, 1998

Abstract

We investigate a natural hierarchy of free Lawvere theories. Algebraic data types turn out to live at the second level of this hierarchy. This is the basis for the main two applications presented here : We solve the problem of finding a general fold algorithm and we introduce the novel programming feature of higher order algebraic data types. These data types are presented with a general type inference method. Further we identify the iteration schemes *map* and *fold* in the higher order case.

1 Introduction

This paper might have also been entitled either "Bananas in Higher Dimensions" following [MFP91, MH95] or "Higher order Polytypic Programming" related to [Mee96]. At the foundation of this paper is the opinion that typeful programming is the righteous way to go. Typeful programming should increase the reuse of software (through all sorts of "poly" type disciplines) and it should also increase the reliability of software (by extending Milner's slogan "well-typed programs can't go wrong"). Further as type systems grow more and more complex the programmer should not be bothered with type annotations. This burden should be carried by a type inference system. With these goals in mind the paper develops a hierarchy of type systems with a type inference algorithm and the basic iteration schemes *map* and *fold*.

The Idea

First we analyse the structure of algebraic data types (adt's). This program construct is well known from languages like ML or Haskell. Have a look at the following examples:

```
List a   =   Nil | Cons a * List a
Tree a   =   Leaf a | Br Tree a * Tree a
Brkt a   =   Opn Brkt (Brkt a) | Cls a
```

The ingredients of the above adt specification $\mathcal{A}$ can be put into an quintuple $\mathcal{A} = (\mathcal{T}, a, \mathcal{D}, t, ag)$. The specification introduces two kinds of new names firstly the type constructors $\mathcal{T}$ = { `List, Tree, Brkt` } and secondly the data constructors $\mathcal{D}$ = { `Nil, Cons, Leaf, Brnch, Opn, Cls` }. Each of the type constructors comes with a string of type variables. The number of variables is called the arity of the type constructor and is represented by the function $a : \mathcal{T} \to \mathbb{N}$. Types are constructed by type variables and type constructors respecting the arity.

In a similar way values are constructed by term variables and data constructors respecting the typing. For type correctness we have to consider that every data constructor belongs to a type constructor expressed by a function $t : \mathcal{D} \to \mathcal{T}$. In our example $t($ `Nil` $) =$ `List` , $t($ `Leaf` $) =$ `Tree` , etc. Further each data constructor has a type for its arguments. In our example $ag($ `Cons` $) =$ `a * List a` , $ag($ `Cls` $) =$ `a` , etc. The precise formalisation of an adt is the subject of section 3.1.

```
mapList f Nil                      =   Nil
mapList f (Cons(a, la))            =   Cons(f a, mapList f la)
mapTree f (Leaf a)                 =   Leaf (f a)
mapTree f (Brnch( ta, ta'))        =   Brnch (mapTree f ta, mapTree f ta')
mapBrkt f (Opn bba)                =   Opn (mapBrkt (mapBrkt f) bba)
mapBrkt f (Cls a)                  =   Cls (f a)
foldList v f Nil                   =   v
foldList v f (Cons(a, la))         =   f(a, foldList v f la)
foldTree f1 f2 (Leaf a)            =   f1 a
foldTree f1 f2 (Br (ta, ta'))      =   f2(foldTree f1 f2 ta, foldTree f1 f2 ta')
```

Figure 1: Iteration schemes for the examples

The point is that we have *two* sets of constructors living in the specification $\mathcal{A}$, that is the type constructors and the data constructors. In the work [SOW98] of Odersky, Sulzmann and the author we analysed the use of constraint type systems. These kind of type systems use a third level language of constraints on types (examples are: overloading, subtyping, dimensions). In this paper we present constraint adt's. A constraint adt specification $\mathcal{B}$ contains *three* sets of constructors where the third set deals with constraint names. Of course this can be identified as a step in a hierarchy of adt's. We define the n^{th}-order adt consisting of $n+1$ constructors glued together in a typeful way (section 3.3 and overview figure 4). Type inference and the minimal typing property can be extended to this hierarchy (see section 4).

Iteration Schemes

For each adt there exist the iteration schemes *map* and *fold*. See figure 1 for the iteration schemes corresponding to our example adt's. The algebraic programming approach promotes the iteration schemes as similar to structured programming. While arbitrary recursive functions are considered as harmful like "spaghetti code".

The careful reader may have noticed that the definition of `foldBrkt` is missing. To the knowledge of the author fold algorithms of this kind have not been defined to date. Fold was defined for the subset of *regular* adt's. In our examples the type `List a` is regular because it is defined recursively by use of the type `List a`. Whereas `Brkt` is not regular because it is define by recursive use of the *type constructor* `Brkt` in argument type `Brkt (Brkt a)`. In section 6 we present the general map and fold algorithm and we extend these algorithms to the n^{th}order.

2 Background

This section gives a background on strings and free theories. Most of this material can be found in [Wag] so we mainly introduce notation here. The structures in strings will be used in the definition of free theories. We will see in section 3 that the structure of free theories will be used to define algebraic data types (adt). Formalising this use of simple structures for defining complex structures will eventually lead to the hierarchy of higher order adt's.

2.1 Strings

For each set (alphabet) T we can construct the free monoid T^* of strings ranged over by u and v. The associative multiplication of this monoid is the concatenation of strings and the unit is the empty string ϵ. We denote by $|u| \in \mathbb{N}$ the length of a string u. We call the obvious function $\bar{u} : [|u|] \to T$ the representation of the string where we use notation $[n] := \{1, \ldots, n\}$ for $n \in \mathbb{N}$. So $u_i = \bar{u}(i)$ is the i-th letter occuring in

tt'	juxtaposition or pairing of terms
$t \bullet t'$	composition of terms
$t \sqsubseteq t'$	instance relation on terms

Figure 2: structures in the free theory T_Σ

the string. A function on alphabets $f : T \to S$ induces a function on strings $f^* : T^* \to S^*$ by defining $f^*(u)_i := f(u_i)$. Notice that, since $1^* \simeq \mathbb{N}$, the length function $|\_| : S^* \to \mathbb{N}$ is the string function $!_T^*$ for the unique function $!_T : T \to 1$ to a singleton set 1.

2.2 Trees

2.2.1 Signatures

A S-sorted signature Σ is a set of generators with sorting $f : (u, s)$. Where a sorting (u, s) is an element of $S^* \times S$ and $|\Sigma|$ denotes the set of unsorted generators. If we take the singleton 1 as sort set S then we get a signature Σ with arity sorting $1^* \times 1 \simeq \mathbb{N}$. So a 1-sorted signature looks like $\Sigma = \{f : n \mid f \in |\Sigma|\}$. We call the elements of $|\Sigma|$ sometimes constructors.

2.2.2 Terms

Each S-sorted signature specifies a set of terms Tm_Σ. In more detail for a string $u \in S^*$ we define $X^{(u)} := \{x_i : u_i \mid i \in [|u|]\}$ to be the set of variables associated to the string u. For each sort $s \in S$ we define the set $Tm^s_\Sigma(u)$ of terms consists of the elements :

(var) $x_i : u_i \in X^{(u)}$ with $u_i = s$.

(constr) $ft_1 \ldots t_n$ for $f : (u', s) \in \Sigma$ with $|u'| = n$ and $t_i \in Tm^{u'_i}_\Sigma(u)$ for $i \in [n]$

We denote an element by either $t \in Tm^s_\Sigma(u)$ by $t : s$ or $u \vdash t : s$.

2.2.3 Free Theories

We define the set T_Σ which is called the free S-sorted theory (or Lawvere theory) by $T_\Sigma(u, v) := Tm^{v_1}_\Sigma(u) \times \cdots \times Tm^{v_{|v|}}_\Sigma(u)$. So each free term $t \in T_\Sigma(u, v)$ (in short $u \vdash t : v$) is a string of sorted terms $u \vdash t_i : v_i$. This fact introduces a natural length function $|\_| : T_\Sigma \to \mathbb{N}$ on free terms.

Further we define the set $u \downarrow T_\Sigma := \bigcup_{v \in S^*} T_\Sigma(u, v)$ caring just for the sorting of the variables and the set $T_\Sigma \downarrow v := \bigcup_{u \in S^*} T_\Sigma(u, v)$ caring just for the sort of the term.

The free theory has more structure $(T_\Sigma, \bullet, \sqsubseteq)$ with a partially defined composition $\bullet : T_\Sigma \times T_\Sigma \rightharpoonup T_\Sigma$ and a partial order $\sqsubseteq \subseteq T_\Sigma \times T_\Sigma$ called the instance order. For sorted terms $u \vdash t : v$ and $w \vdash t' : u$ the composition $t \bullet t'$ is defined by substituting the variables $x_i : u_i$ occuring in t by the terms $t'_i \in Tm^{u_i}_\Sigma(w)$. Note that the result of the composition is sorted as $w \vdash t \bullet t' : v$. This defines the instance order by $t \sqsubseteq t''$ iff there is a term t' with $t \bullet t' = t''$.

2.2.4 Theory Morphisms

A function of signatures $f : \Sigma \to \Sigma'$ from a S-sorted signature Σ to a T-sorted signature Σ' consists of an alphabet map $f_a : S \to T$ and a sort-compatible constructor map $f_c : |\Sigma| \to |\Sigma'|$. That is for all constructors $f : (u, s) \in \Sigma$ we get result sorting $f_c(f) : (f_a^*(u), f_a(s)) \in \Sigma'$.

A function of signatures $f : \Sigma \to \Sigma'$ induces a function on terms $f^{tm} : Tm_\Sigma \to Tm_{\Sigma'}$ in the obvious way. More specifically we get a sort compatible map on the terms $f^{tm} : Tm^t_\Sigma(u) \to Tm^{f_a(t)}_{\Sigma'}(f_a^*(u))$. By using the the equality $T_\Sigma = Tm^*_\Sigma$ we extend function from terms to theories by setting $f^t := (f^{tm})^* : T_\Sigma \to T_{\Sigma'}$.

2.2.5 Sub-Theories

A sub-theory is a subset $T' \subseteq T_{\Sigma'}$ where $t, t' \in T'$ implies $tt', t \bullet t' \in T'$ if juxtaposition and composition is defined in $T_{\Sigma'}$. So the length function $|\_| : T' \to \mathbb{N}$

For a type string $\tau = \tau_1 \dots \tau_n \in Ty_{\mathcal{A}}(n, \_)$

right of typing	$\tau \vdash \dots$	typotheses	$x_1 : \tau_1, \dots, x_n : \tau_n$
left of typing	$\dots \vdash v : \tau$	cartesian product type	$\tau_1 \times \dots \times \tau_n$
right of composition	$\dots \tau' \bullet \tau \dots$	substitution	$\{\alpha_1 \mapsto \tau_1, \dots, \alpha_n \mapsto \tau_n\}$

Figure 3: interpretation of type strings

and the T-sorting discipline $T'(u, v)$ (by assumption Σ' is a T-sorted signature) are just restrictions according to the subset relation $T' \subseteq T_\Sigma$. Obviously the image $T' := img(f^t)$ of the theory morphism f^t induces a sub-theory. In this case we have a S-sorting of terms $t \in T'$ by $u \vdash t : v$ if there is a term $t' \in T_\Sigma(u, v)$ with $f^t(t') = t$.

3 Algebraic data types (adt)

First we define algebraic data types. Next we will explicitly define second order adt's. Both definitions consists in a definition of a specification (similar to the signatures of free theories) and the definition of the values induced by this specification (similar to the terms of a free theory). In the definition of free theories we used the structure of strings as the sorting. Something very similar will happen with the structure of the types used in in the definition of an adt. The observation that the types of an adt are the values of an adt of previous order will lead to the hierarchy of adt's.

Method

The main result in this section is the general construction of values starting from a specification. To this end we define two signatures: firstly the untyped signature giving the free theory of pre-values, secondly the explicitly typed signature giving the the free theory of explicitly typed values. We will see that the erasure of type annotations is a theory morphism between these free theories. Finally the image of the erasure morphism defines the implicitly typed values of an adt. So the values are a sub-theory of the pre-values.

3.1 First Order ADT's

3.1.1 Specifications

The specification of an adt is a quintuple $\mathcal{A} = (\mathcal{T}, a, \mathcal{D}, t, ag)$ where $\mathcal{T}$ is the set of type constructors and $\mathcal{D}$ is the set of data constructors. The function $a : \mathcal{T} \to \mathbb{N}$ is the arity in the signature $\Sigma\mathcal{T}$ with $|\Sigma\mathcal{T}| = \mathcal{T}$ and $\Sigma\mathcal{T} := \{T : a(T) \,|\, T \in \mathcal{T}\}$. We call the free theory $Ty_{\mathcal{A}} := T_{\Sigma\mathcal{T}}$ the types of the adt $\mathcal{A}$. The function $t : \mathcal{D} \to \mathcal{T}$ gives the sort of a data constructor. The function $ag : \mathcal{D} \to Ty_{\mathcal{A}}$ gives the argument type of a data constructor. The argument type of a data constructor $C \in \mathcal{D}$ can use only the $n = a(T)$ type variables of its sort $T = t(C)$. So for all constructors $C \in \mathcal{D}$ the restriction $ag(C) \in a(t(C)) \downarrow T_\Sigma$ is needed.

3.1.2 Values

We define the $Ty_{\mathcal{A}}$-sorted signature $\Sigma\mathcal{D}^{exp}$ of explicitly typed data constructors by :

$$\begin{aligned}\Sigma\mathcal{D} := \ & \{C\tau : (\tau_C \bullet \tau, T\tau) \,|\, C \in \mathcal{D} \text{ with} \\ & ag(C) = \tau_C, t(C) = T, \tau \in Ty_{\mathcal{A}} \downarrow a(T)\}\end{aligned}$$

We further get the untyped arity signature $\Sigma\mathcal{D}^{pre}$ for pre-values by $|\Sigma\mathcal{D}^{pre}| = \mathcal{D}$ and $\Sigma\mathcal{D}^{pre} := \{C : |ag(C)| \,|\, C \in \mathcal{D}\}$

Remember that the length function $|\_| : Ty_{\mathcal{A}} = T_{\Sigma\mathcal{T}} \to \mathbb{N} \simeq 1^*$ is the string function $(!_{Ty_{\mathcal{A}}})^*$ induced by the unique function $!_{Ty_{\mathcal{A}}} : Ty_{\mathcal{A}} \to 1$ to the singleton 1. So we get a function of signatures $f^{impl} : \Sigma\mathcal{D}^{exp} \to \Sigma\mathcal{D}^{pre}$. This is a function from a

Order	Specification	Generators	Values	Types	Constraints
$n=0$	Σ	$\|\Sigma\|$	T_Σ	arities $1^* \simeq \mathbb{N}$	trivial 1
$n=1$	$\mathcal{A}$	$\mathcal{T}, \mathcal{D}$	$Val_\mathcal{A} = img(\|\| \_ \|\|)$	$Ty_\mathcal{A} = T_{\Sigma\mathcal{T}}$	arities $1^* \simeq \mathbb{N}$
$n=2$	$\mathcal{B}$	$\mathcal{C}, \mathcal{T}, \mathcal{D}$	$Val_\mathcal{B} = img(\|\| \_ \|\|)$	$cTy_\mathcal{B} = Val_{\mathcal{B}_{tadt}}$	$Cnst_\mathcal{B} = T_{\Sigma\mathcal{C}}$
$\vdots$					
$n-2$	$\mathcal{S}^{--}$				
$n-1$	$\mathcal{S}^{-}$				
n	$\mathcal{S}$	$G\mathcal{S}_0, \ldots, G\mathcal{S}_n$	$Val_\mathcal{S} = img(\|\| \_ \|\|)$	$Ty_\mathcal{S} = Val_{\mathcal{S}^-}$	$Cnst_\mathcal{S} = Ty_{\mathcal{S}^-} = Val_{\mathcal{S}^{--}}$

where $\| \_ \|: T_{\Sigma^{exp}} \to T_{\Sigma^{pre}}$ is the erasure function
and Σ^{exp} is the explictly typed signature
and Σ^{pre} is the untyped pre-value signature

Figure 4: Structures in higher order algebraic data types

$Ty_\mathcal{A}$-sorted signature to a 1-sorted signature : The alphabet part is defined as $f_a^{impl} :=!_{Ty_\mathcal{A}}$ and the constructor part $f_c^{impl} : |\Sigma\mathcal{D}^{exp}| = \mathcal{D} \times Ty_\mathcal{A} \to |\Sigma\mathcal{D}^{pre}| = \mathcal{D}$ is just defined by the first projection $f_c^{impl}(C\tau) = C$. So we get a morphism on free theories $\| \_ \| := (f^{impl})^t : T_{\Sigma\mathcal{D}^{exp}} \to T_{\Sigma\mathcal{D}^{pre}}$ called the erasure. The elements of the image image of the erasure $Val_\mathcal{A} := img(\| \_ \|)$ are called the values of the adt $\mathcal{A}$ or the implicitly typed data terms.

3.1.3 Composition

A value $v \in Val_\mathcal{A}$ has a typing $\tau \vdash v : \tau'$ (alternative notation $v \in Val_\mathcal{A}(\tau, \tau')$) iff there is an explicitly typed value $v' \in T_{\Sigma\mathcal{D}^{exp}}(\tau, \tau')$ with erasure $\| v' \| = v$. So two values $\tau \vdash v$ and $v' : \tau'$ compose $v \bullet v'$ iff there exists a common upper bound τ'' on the types that is $\tau \sqsubseteq \tau''$ and $\tau' \sqsubseteq \tau''$.

3.2 Second Order ADT's

3.2.1 Specification

A second order adt specification is a 7-tuple $\mathcal{B} = (\mathcal{C}, ac, \mathcal{T}, tt, tag, \mathcal{D}, td, dag)$. As a synonym we use the name constraint adt. We have three sets of generators; the constraint names $\mathcal{C}$, the type constructors $\mathcal{T}$ and the data constructors $\mathcal{D}$. The quintuple $\mathcal{B}_{tadt} = (\mathcal{C}, ac, \mathcal{T}, tt, tag)$ must be an adt. We call the free theory $D \in Cnst_\mathcal{B} := T_{\Sigma\mathcal{C}} = Ty_{\mathcal{B}_{tadt}}$ the constraints of $\mathcal{B}$. We call $\tau \in cTy_\mathcal{B} := Val_{\mathcal{B}_{tadt}}$ the constrained types of $\mathcal{B}$. We use the formula $D \Rightarrow \tau :: D$ as the alternative denotation for for $\tau \in cTy_\mathcal{B}(D, D') = Val_{\mathcal{B}_{tadt}}(D, D')$ to state that the type variables $\alpha_i :: D_i$ occuring in type τ are constrained and the types $\tau_j :: D'_j$ are constrained.

3.2.2 Second Order Values

The function $td : \mathcal{D} \to \mathcal{T}$ gives the sorting of data constructors and the function $dag : \mathcal{D} \to cTy_\mathcal{B}$ gives the constrained type of a data constructor. Given a data constructors $C \in \mathcal{D}$ then only the constrained type variables $\alpha_i :: D_i$ can occur in the argument type $\tau_C = dag(C)$. Where D is the type constructor constraint argument $D = tag(T)$ associated to the type constructor that is $T = td(C)$. So the specification satisfies for any data constructor $C \in \mathcal{D}$ the property $dag(C) \in tag(td(C)) \downarrow cTy_\mathcal{B}$.

Again (as in section 3.1.2) we define the signature of explicitly typed data constructors :

$$\Sigma\mathcal{D}^{exp} := \{C\tau : (\tau_C \bullet \tau, T\tau) \,|\, C \in \mathcal{D}, \tau_C = dag(C), T = td(C), \tau \in cTy_\mathcal{B} \downarrow D, tag(T) \sqsubseteq D\}$$

Note that we have to restrict the instantiation type $\tau :: D$ to a well constrained type. We get the usual pre-value signature $\Sigma\mathcal{D}^{pre}$ and the erasure $\| \_ \|$: $T_{\Sigma\mathcal{D}^{exp}} \rightarrow T_{\Sigma\mathcal{D}^{pre}}$ as before. So we define the values of $\mathcal{B}$ as the image of the erasure $Val_{\mathcal{B}} := img(\| \_ \|)$ and get again the notion of a typing $\tau \vdash v : \tau'$. If the types in the typing are constrained like $D \Rightarrow \tau$ and $D \Rightarrow \tau' :: D'$ then we denote a constrained typing by $D \Rightarrow \tau \vdash v : \tau' :: D'$.

3.3 Higher Order ADT's *or* The Full Hierarchy

We will now define the notion of n^{th}-order adt. for every adt specification $\mathcal{S}$ of n^{th}-order we will define the set of values $Val_{\mathcal{S}}$ of n^{th}-order. The set of values of n^{th}-order will be a sub-theory of a free theory where the sorts are values of order $n-1$. This sub-theory will be induced by the image of an erasure-like theory morphism.

3.3.1 Specification

We define the specification of an n^{th}-order adt inductively. The specification of order 0 consists just of an arity signature $\Sigma\mathcal{S}$. So the values of order 0 are just the free terms $T_{\Sigma\mathcal{S}}$ and the types are the arities $1^* \simeq \mathbb{N}$.

In the case $n \geq 1$ a specification $\mathcal{S}$ consist of a specification $\mathcal{S}^-$ of order $n-1$ extended by a triple $(G\mathcal{S}_n, st_n, agt_n)$, where : $G\mathcal{S}_n$ is a set of generators, st_n is the sorting of generators and agt_n is the argument typing of generators. The generator sorting is a function $st_n : G\mathcal{S}_n \rightarrow G\mathcal{S}_{n-1}$ where $G\mathcal{S}_{n-1}$ is the generator set of the extension for $\mathcal{S}^-$ in case $n > 1$ or the set of constructors $|\Sigma\mathcal{S}|$ in the case $n = 1$ so we set $G\mathcal{S}_o := |\Sigma\mathcal{S}|$.

The argument typing is a function $agt_n : G\mathcal{S}_n \rightarrow Val_{\mathcal{S}^-}$. We call the values of specification $\mathcal{S}^-$ the types of specification $\mathcal{S}$ (in short $Ty_{\mathcal{S}} := Val_{\mathcal{S}^-}$). The argument type function agt_n has to statisfy the restriction for all generators $f \in G\mathcal{S}_n.agt_n(f) \in domfxagt_{n-1}(st(f))Ty_{\mathcal{S}}$. As mentioned above, the types for the adt of order zero are the arities $1^* \simeq \mathbb{N}$. So we define $agt_0 : G\mathcal{S}_o = |\Sigma\mathcal{S}| \rightarrow \mathbb{N}$ to be the arity of the constructors $f \in |\Sigma\mathcal{S}|$.

3.3.2 Values

We will now define the explicitly typed signature $\Sigma\mathcal{S}^{exp}$ and the arity signature of pre-values $\Sigma\mathcal{S}^{pre}$. We will see that $\Sigma\mathcal{S}^{exp}$ is a $Ty_{\mathcal{S}}$-sorted signature. With this at hand we define the erasure $\| \_ \|$: $T_{\Sigma\mathcal{S}^{exp}} \rightarrow T_{\Sigma\mathcal{S}^{pre}}$ and define the values of $\mathcal{S}$ as the image of the erasure $Val_{\mathcal{S}} := img(\| \_ \|)$ which we call the set of implicit typed values of n^{th}-order.

To define the arity signature $\Sigma\mathcal{S}^{pre}$ we use the fact that $Ty_{\mathcal{S}}$ is a sub-theory so we can apply the length function $|\_| : Ty_{\mathcal{S}} \rightarrow \mathbb{N}$ and get a definition $\Sigma\mathcal{S}^{pre} := \{f : |agt_n(f)| \,|\, f \in G\mathcal{S}_n\}$ by use of the argument typing function $agt_n : G\mathcal{S}_n \rightarrow Ty_{\mathcal{S}}$.

To define the $Ty_{\mathcal{S}}$-sorted signature $\Sigma\mathcal{S}^{exp}$ we use the fact that the specification $\mathcal{S}^-$ consists of specification $\mathcal{S}^{--}$ of order $n-2$ and an extension $(G\mathcal{S}_{n-1}, st_{n-1}, agt_{n-1})$ with $agt_{n-1} : G\mathcal{S}_{n-1} \rightarrow Val_{\mathcal{S}^{--}} = Ty_{\mathcal{S}^-}$. We will define the constraints of specification $\mathcal{S}$ as $Cnst_{\mathcal{S}} := Val_{\mathcal{S}^{--}} = Ty_{\mathcal{S}^-}$. We need further that $Ty_{\mathcal{S}} = Val_{\mathcal{S}^-}$ is a sub-theory so there exists a partially defined composition $\tau \bullet_{\mathcal{S}^-} \tau'$ for types $\tau, \tau' \in Ty_{\mathcal{S}}$. Recall paragraph 3.1.3 that composition $\tau \bullet_{\mathcal{S}^-} \tau'$ is defined if $D \vdash \tau$ and $\tau' : D'$ with $D \sqsubseteq_{\mathcal{S}^{--}} D'$ where we denote by $\sqsubseteq_{\mathcal{S}^{--}}$ the instance order in the theory of constraints $Cnstr_{\mathcal{S}}$.

Altogether we can define the explicitly typed signature as :

$$\begin{aligned}\Sigma\mathcal{S}^{exp} := \{ & f\tau : (\tau_f \bullet_{\mathcal{S}^-} \tau, T\tau) \,|\, f \in G\mathcal{S}_n, \\ & agt_n(f) = \tau_f \in Ty_{\mathcal{S}}, st_n(f) = T \in G\mathcal{S}_{n-1}, \\ & \tau \in Ty_{\mathcal{S}} \downarrow D, agt_{n-1}(T) \sqsubseteq_{\mathcal{S}^{--}} D\}\end{aligned}$$

4 Type Inference

Due to lack of space we state here just the main result, namely the type inference algorithm and the minimal typing property. The details can be found in the full paper [Weh98].

4.1 Well-Typedness

To any given algebraic data type $\mathcal{A}$ we get the minimal typing property.

Definition 1 *(Minimal Typing) A typing $\tau \vdash v : \tau'$ is called minimal iff any other typing $\tau_1 \vdash v : \tau_2$ of* *the same value v is a super instance $\tau\tau' \sqsubseteq \tau_1\tau_2$ of the original typing. (We assume w.o.l.g. that $|\tau| = |\tau_1|$)*

Proposition 1 *(Minimal Typing) For each value $v \in Val_{\mathcal{A}}$ there exists a minimal typing $\tau \vdash v : \tau'$.*

This property is witnessed by an algorithm $\mathit{tinf}_{\mathcal{A}} : T_{\Sigma\mathcal{D}^{pre}}(n,m) \rightharpoonup Ty_{\mathcal{A}}(k,n) \times Ty_{\mathcal{A}}(k,m)$ to any adt $\mathcal{A}$. The algorithm $\mathit{tinf}_{\mathcal{A}}$ takes a pre-value $v \in T_{\Sigma\mathcal{D}^{pre}}$ as argument and terminates iff it is a value $v \in Val_{\mathcal{A}}$ otherwise it will raise an error. The result in case of termination $\mathit{tinf}(v) = (\tau, \tau')$ will be the minimal typing $\tau \vdash v : \tau'$.

4.2 Constrained Type Inference

Given a constrained adt $\mathcal{B} = (\mathcal{C}, ac, \mathcal{T}, tt, tag, \mathcal{D}, td, dag)$ we can extract two adt's living in that specification. There are the type adt $\mathcal{B}_{tadt} = (\mathcal{C}, ac, \mathcal{T}, tt, tag)$ and the data adt $\mathcal{B}_{dadt} = (\mathcal{T}, ab, \mathcal{D}, td, dag)$ with $ab = |\_| \circ tag : \to \mathbb{N}$. Previous paragraph 4.1 stated the existence of a type inference procedure with minimal typing property for each adt. We use the algorithms $\mathit{tinf}_{\mathcal{B}_{tadt}}$ and $\mathit{tinf}_{\mathcal{B}_{dadt}}$ to define a constrained type inference algorithm $\mathit{ctinf}_{\mathcal{B}}$:

$\mathit{ctinf}_{\mathcal{B}} : T_{\Sigma\mathcal{D}^{pre}} \rightharpoonup Cnst_{\mathcal{B}} \times cTy_{\mathcal{B}} \times cTy_{\mathcal{B}} \times Cnst_{\mathcal{B}}$
  $\mathit{ctinf}(v) =$
 /* type inference on the value ...
   $(\tau, \tau') := \mathit{tinf}_{\mathcal{B}_{tadt}}(v)$;
 /* ... constraint inference on the infered types
   $(D, D''D') := \mathit{tinf}_{\mathcal{B}_{dadt}}(\tau\tau')$;
    so that $D \Rightarrow \tau :: D''$ and $D \Rightarrow \tau' :: D'$
   $\mathsf{return}(D, \tau, \tau', D')$;

Constrained inference algorithm can be applied to a pre-value $v \in T_{\Sigma\mathcal{D}^{pre}}$. In case of termination the result $\mathit{ctinf}_{\mathcal{B}}(v) = (D, \tau, \tau', D')$ gives a minimal constrained typing $D \Rightarrow \tau \vdash v : \tau' :: D'$. The minimality property is just a corollary which follows by application of minimality for adt's $\mathcal{B}_{tadt}$ and $\mathcal{B}_{dadt}$.

4.3 General Type Inference

The general type inference algorithm for adt's of n^{th}-order is built by following observation. The quintuple $\mathcal{S}_{adt} = (G\mathcal{S}_{n-1}, sa_{n-1}, G\mathcal{S}_n, st_n, agt_n)$ with $sa_i = |\_| \circ agt_i : G\mathcal{S}_i \to \mathbb{N}$ for $i \in [n]$ forms an adt. So in case $n = 1$ we can define the type inference algorithm as $tp_{\mathcal{S}} := \mathit{tinf}_{\mathcal{S}_{adt}} : T_{\Sigma\mathcal{S}^{pre}} \rightharpoonup Ty_{\mathcal{S}} \times Ty_{\mathcal{S}}$. In case $n > 1$ we define the general algorithm as :

$tp_{\mathcal{S}} : T_{\Sigma\mathcal{S}_n} \rightharpoonup lTp_{\mathcal{S}} \times rTp_{\mathcal{S}}$
  with $lTp_{\mathcal{S}} := lTp_{\mathcal{S}^-} \times Ty_{\mathcal{S}}$ and
  $rTp_{\mathcal{S}} := Ty_{\mathcal{S}} \times rTp_{\mathcal{S}^-}$
$tp_{\mathcal{S}}(v) =$
  $(\tau, \tau') := \mathit{tinf}_{\mathcal{S}_{adt}}(v)$;
  $(t, t_1't_2') := tp_{\mathcal{S}^-}(\tau\tau')$;
   with $|t_1'| = |\tau|$ and $|t_2'| = |\tau'|$
  $\mathsf{return}((t, \tau), (\tau', t_2'))$;

By n applications of the minimal typing property of adt's we get then in case of termination $tp_{\mathcal{S}}(v) = (t, t')$ a minimal typing $t \vdash v : t'$.

5 Term Transformations

We call a partial function $f : T_\Sigma \rightharpoonup T_\Sigma$ on free theories a term transformation iff there is a sorting restriction so that the function $f : T_\Sigma(u, v) \to T_\Sigma(u', v')$ is total.

In the following we define a series of term transforms. These term transforms on free theories are used to define algebraic functions on adt's. Eventually we get a representation of maps and folds (section 6). Finally we will see that this is the right generalisation which extends to iterations schemes of n^{th}order (section 7).

5.1 Occurrences

The generators $f \in |\Sigma|$ of a tree $t \in Tm_\Sigma$ occur at certain places in the tree. In order to name these places we introduce the set of occurrences $\mathcal{O} := 1^{**} \simeq \mathbb{N}^*$. We will use o for an occurrence and $O \subseteq \mathcal{O}$ for sets of occurrences. Note that occurrences are strings so the prefix order $\sqsubseteq^p$ is defined. Because of the isomorphy $\mathcal{O} \simeq \mathbb{N}^*$ we can define the lexicographic order $\sqsubseteq^{lex}$ on occurrences by:

$$o \sqsubseteq^{lex} o' \text{ iff } \begin{cases} \text{true} & \text{if } o = o''i \text{ and } o' = o''j \\ & \quad \text{with } i \leq j \\ o \sqsubseteq^p o' & \text{else} \end{cases}$$

(subtree substitution)

for $o \in occ(t)$ with $u \vdash t|o : s$ and term $u \vdash t' : s$

$$\overline{(t|_m o \mapsto t')}(o') := \begin{cases} \overline{t'}(o'') & \text{if } o' = oo'' \\ \overline{t}(o') & \text{else} \end{cases}$$

(constructor substitution)

for $o \in cocc(t)$ with $t@o = f : (w, s)$ and term $w \vdash t' : s$

$$\overline{(t|_f o \mapsto t')}(o') := \begin{cases} \overline{t'}(o'') & \text{if } o' = oo'' \text{ with } o'' \in cocc(t') \\ \overline{t}(oio_2) & \text{if } o' = oo_1o_2 \text{ with } t'@o_1 = x_i \\ \overline{t}(o') & \text{else} \end{cases}$$

Figure 5: substitutions at occurrences

5.2 Tree Representation

Given a S-sorted tree $t \in Tm_\Sigma$ the representation of a tree is a partial function $\overline{t} : \mathcal{O} \rightharpoonup |\Sigma| \cap X$. The representation is defined by $\overline{ft'}(\epsilon) = f$, $\overline{x}(\epsilon) = x$ and $\overline{ft'}(io)$ is defined as $\overline{t'_i}(o)$ iff $i \in [|t'|]$. We write $t@o := \overline{t}(o)$ as short form for the constructor or variable occuring at place o.

We define the domain of a representation as $occ(t) := \{o \in \mathcal{O} \,|\, t@o \text{ is defined }\}$. The constructor occurrences of a term are defined by $cocc(t) := \{o \in \mathcal{O} \,|\, t@o \in |\Sigma|\}$. A representations of a term string $t \in T_\Sigma$ is defined by setting $\overline{t}(io) := \overline{t_i}(o)$ iff $i \in [|t|]$.

There is a one-to-one correspondence between trees and their representation. So in the following we define operations on trees by explaining the operation on representations. The notion of a subterm at an occurrence $t|o$ is defined by the representation $\overline{t|o}(o') = \overline{t}(oo')$.

5.3 Substitutions

Given a term $u \vdash t : v$ we define two notions of sort compatible substitution at place $o \in \mathcal{O}$ (see figure 5)

5.3.1 Substitution at Occurrence Sets

Given a term t and a set of occurrences $O \subseteq occ(t)$ we say $t|O : s$ is well sorted iff for all occurrences $o \in O$ the induced subterms $t|o$ have sort s. Obviously we can extend the notion of subtree substitution on well sorted occurrence sets. Especially if we denote the set of occurrences of variable $x_i : s$ by $occ(x_i, t) := \{o \in \mathcal{O} \,|\, t|o = x_i\}$. We introduce the short form $(t|_m x_i \mapsto t') := (t|_m occ(x_i, t) \mapsto t')$.

Given a set of constructor occurrences $O \subseteq cocc(t)$ we say $t@O : (u, s)$ is a sorting of set O iff for all occurrences $o \in O$ the constructors $t@o$ coincide on their sorting (u, s). Obviously we can extend the notion of constructor substitution on occurrence sets with sorting. Especially if we denote the set of occurrences of constructor $f \in |\Sigma|$ by $occ(f, t) := \{o \in \mathcal{O} \,|\, t@o = f\}$. We introduce the short form $(t|_f f \mapsto t') := (t|_f occ(f, t) \mapsto t')$.

5.3.2 Evaluation Order

The above definitions of substitutions $(t|O \mapsto t')$ on occurrence sets O are only well defined if we assume a certain order in the evaluation of substitutions on occurrences $o \in \mathcal{O}$. In any case of two occurrences $o, o' \in O$ with $o \sqsubseteq^p o'$ substitution on o' has to be done before substitution on o. This is necessary because substitution on o might change the representation on all superfixes (for example o'). So we define that substitution on occurrence sets have to be evaluated in a bottom up manner.

5.3.3 Term Transformation at an Occurrence

Instead of substitution at an occurrence $o \in occ(t)$ in a term t we can apply a term transformation g :

$T_\Sigma \rightharpoonup T_\Sigma$ at the occurrence. Given term $u \vdash t : v$ the function g must satisfy certain sort restrictions. In case of a subterm sorted like $t|o : s$ we need a restriction $g : T_\Sigma(u,s) \to T_\Sigma(u,s)$ and we denote the application as $(t|_m o \mapsto g)$. In case of constructor substitution and the generator $t@o = f : (w,s)$ we need a restriction $g : T_\Sigma(u,w) \to T_\Sigma(u,s)$ and denote the application as $(t|_f o \mapsto g)$.

5.4 Family Substitutions

Given a term $u \vdash t : v$, a family of term transforms $\vec{f} := \{f_i : T_\Sigma \to T_\Sigma\}_{i \in I}$ and a set of occurrences $O \subseteq occ(t)$. We want to apply at each occurrence $o \in O$ a certain transform f_i. To this end we define partial functions called either occurrence index function $oindx : \mathcal{O} \rightharpoonup I$ or constructor index function $cindx : |\Sigma| \rightharpoonup I$.

5.4.1 M-Family Substitution

The tuple $(\vec{f}, oindx)$ is a m-compatible family iff for each occurrence $o \in O$ the index $oindx(o) = i \in I$ is defined and the associated transform is sorted as $f_i : T_\Sigma(u,s) \to T_\Sigma(u,s)$ in case of $t|o : s$. This leads to the m-family substitution $(t|_m O \mapsto (\vec{f}, oindx))$. In case of the equation $O = \{o \in \mathcal{O} \mid oindx(o) \text{ defined }\}$ we denote in short $(t|_m(\vec{f}, oindx))$.

5.4.2 F-Family Substitution

We restrict the occurrence set O to constructor occurrences $O \subseteq cocc(t)$. The tuple $(\vec{f}, cindx)$ is a f-compatible family iff for each occurrence $o \in O$ the index $cindx(t@o) = i \in I$ is defined and the associated transform is sorted as $f_i : T_\Sigma(u,w) \to T_\Sigma(u,s)$ in case of sorting $t@o : (w,s)$. This leads to the f-family substitution $(t|_f O \mapsto (\vec{f}, cindx))$.

6 Representation of Iteration Schemes

Along with the definition of an adt come some program schemes to define functions on the values of the adt. We will look at maps and folds. Fold functions for simple cases as the data types `List` or `Tree` are well known in the functional programming community. But the general construction of a fold associated to any adt was yet an open problem. Here we will define such a fold as a term transform on pre-values which can be polymorphically typed.

In the above section we defined the notion of term transformations in free theories. Here we will extend this notion to adt's. First we need the right notion of occurrence in adt's. We will relate value occurrences to type occurrences (section 6.2). This will eventually lead to a representation of map by a m-family substitution and to the representation of fold by a f-family substitution (section 6.3). Well-sorting preservation of the substitutions will imply well-typedness of the iteration schemes.

The full paper[Weh98] contains a zoo of program schemes on adt's following a more traditional presentation in form of typed combinators.

6.1 Algebraic Functions

We deal with well-typed functions $f : Val_\mathcal{A} \rightharpoonup Val_\mathcal{A}$ of algebraic values. A function is well-typed (in short: $f : \tau \to \tau'$) if its well defined as a function of sets $f : Val_\mathcal{A}(\epsilon,\tau) \to Val_\mathcal{A}(\epsilon,\tau')$. A function $f : \tau \to \tau$ with $n \vdash \tau\tau'$ is polymorphic iff the restrictions $f\tau'' : \tau \bullet \tau'' \to \tau' \bullet \tau''$ for types $\tau'' : n$ are well defined. Any value $\tau \vdash v : \tau'$ with $n \vdash \tau\tau'$ induces a polymorphic function $\underline{v} : \tau \to \tau'$ by composition from the left $\underline{v}(v') := v \bullet v'$

6.2 Associated Occurrences

Given a typed value $v : \tau$ we explain how an occurrence in the type $o \in occ(\tau)$ induces a set of occurrences $O \in occ(v)$ in the value. To this end we define to a occurrence set $O \subseteq \mathcal{O}$ and a natural number $j \in \mathbb{N}$ the set $j.O := \{jo \mid o \in O\}$. We define :

$$
\begin{aligned}
&\{v||o\} \subseteq occ(v) \text{ where} \\
&\quad \{v||\epsilon\} := \{\epsilon\} \\
&\quad \{Cv||io\} := \textstyle\bigcup_{j \in [|v|]} \bigcup_{o' \in occ(\alpha_i,(\tau_C)_j)} j.\{v_j||o'o\}
\end{aligned}
$$

$m_{\tau''}\vec{f} : \tau'' \bullet \tau \to \tau'' \bullet \tau'$ map
for a family of functions $\vec{f} := \{f_i : \tau_i \to \tau'_i\}_{i\in[n]}$ and a type $n \vdash \tau''$
$f_{T,\tau}\vec{f} : T\underline{n} \to \tau$ general fold
for a family of polymorphic functions $\vec{f} := \{f_C : \tau_C^r \to \tau\}_{C\in\mathcal{D}_T}$ with $n \vdash \tau$
on reduced argument types $\tau_C^r := (\tau_C|_f T \mapsto \tau)$

Figure 6: program schemes on algebraic data types

$$\{v@@o\} \subseteq cocc(v) \text{ where}$$
$$\{Cv@@\epsilon\} := \{\epsilon\} \cup \bigcup_{j\in[|v|]} \bigcup_{o'\in occ(t(C),(\tau_C)_j)} j.\{v_j@@o'o\}$$
$$\{Cv@@io\} := \bigcup_{j\in[|v|]} \bigcup_{o'\in occ(\alpha_i,(\tau_C)_j)} j.\{v_j@@o'o\}$$

Proposition 2 *(Occurrence Separation) Different occurrences $o \neq o'$ induce disjoint value occurrence sets. That is $\{v||o\} \cap \{v||o'\} = \emptyset$ and $\{v@@o\} \cap \{v@@o'\} = \emptyset$*

6.2.1 Well-Typed Occurrences

Given a typed value $v : \tau$ and a type occurrence $o \in occ(\tau)$ the the set of associated value occurrences is well typed $\{v||o\} : \tau|o$ that is for each value occurrence $o' \in \{v||o\}$ the sub-value $v|o'$ has type $\tau|o$. Obviously we can extend the notion of value occurrences $\{v||O\}$ associated to a set of type occurrences $O \subseteq occ(\tau)$. The set of occurrences O has to coincide on the induced types that is all occurrences $o \in O$ induce the same type $\tau|o$.

6.2.2 Well-Sorted Occurrences

We denote the sort of a data constructor by $C :^s T$ in case of $t(C) = T$. Given a typed value $v : \tau$ and a constructor occurrence $o \in cocc(\tau)$ then the associated data constructors at the occurrences are well sorted $\{v@@o\} :^s \tau@o$ that is for each value occurrence $o' \in \{v@@o\}$ the associated constructor $v@o'$ has sort $\tau@o$.

6.3 First order Schemes

With the above connection of occurrences and adt's we are able to state the definition of program schemes for adt's.

6.3.1 Map

We give a definition of a map $m_{\tau''}$ for a type $n \vdash \tau''$ and a family of functions $\vec{f} := \{f_i : \tau_i \to \tau'_i\}_{i\in[n]}$. The short forms $\tau = \tau_1 \ldots \tau_n$ and $\tau' = \tau'_1 \ldots \tau'_n$ are used. The operation on a value $v : \tau'' \bullet \tau$ is defined by :

$$m_{\tau''}\vec{f}v := (v|_m(\vec{f}, oindx))$$

where we apply the notion of m-family substitution from paragraph 5.4.1. In order to apply this notion we have to define an occurrence index function $oindx : O \to [n]$. We define the sets $O_i := \{v||occ(\alpha_i, \tau'')\}$ for $i \in [n]$ containing the value associated to the type occurrences of type variable α_i. Finally we define $O := O_1 \cup \cdots \cup O_n$ with a canonical occurrence index $oindx(o) = i$ iff $o \in O_i$. The occurrence index is well defined because occurrence separation implies disjointness $i \neq i' \Rightarrow O_i \cap O_{i'} = \emptyset$.

In order to prove well-typedness of map we have to interpret the operations of map on explicitly typed values. Well-typedness of the occurrence sets O_i is interpreted as well-$Ty_{\mathcal{A}}$-sortedness. Well-sort preservation of m-family substitution gives explicit typability of map results. By application of erasure we get the implicitly typed result $m_{\tau''}\vec{f}v : \tau'' \bullet \tau'$.

6.3.2 Fold

We give a definition of a general fold $f_{T,\tau}$ with arity $n = a(T)$ a type $n \vdash \tau : 1$ and a family of polymorphic functions $\vec{f} := \{f_C : \tau_C^r \to \tau\}_{C \in \mathcal{D}_T}$. The argument type associated to a data constructor in set $\mathcal{D}_T := \{C \in \mathcal{C} \,|\, t(C) = T\}$ is give by the specification through $\tau_C := ag(C)$. The reduced type τ_C^r is defined by a constructor substitution (see 5.3.1)$\tau_C^r := (\tau_C|_f T \mapsto \tau)$. Given a typed value $v : T\tau'$ we define the result a fold application by :

$$f_{T,\tau}\vec{f}v := (v|_f\{v@@\epsilon\} \mapsto (\vec{f}, id_{\mathcal{D}_T}))$$

where we apply the notion of f-family substitution from paragraph 5.4.2.

Applying well-sortedness of the occurrence set $\{v@@\epsilon\}$ and well-sort preservation of f-family substitution implies well-typedness of result $f_{T,\tau}\vec{f}v : \tau \bullet \tau'$.

7 n^{th}-Order Iteration Schemes

7.1 Out most Constraints

We define a sorting function $srt_k : G\mathcal{S}_n \to G\mathcal{S}_{k-1}$ for $k \in [n]$. by setting $srt_n := st_n$ and $srt_k := st_k \circ srt_{k+1}$. Given a generator $f \in G\mathcal{S}_n$ we define the associated out most constraint as $D_f := agt_1 \circ str_2(f)$ and the number of used variables in that constraint is $n_f := agt_0 \circ str_1(f) \in \mathbb{N}$ so we have $D_f \in n_f \downarrow T_{\Sigma\mathcal{S}}$. To each value $v \in Val_{\mathcal{S}}$ exists a typing $t \vdash v : t'$. Recall that t is a sequence of values of ascending order and t' is a sequence of values of descending order. So any typing has form $t \vdash v : t''D$ for a value of order zero $D \in T_{\Sigma\mathcal{S}}$. We will denote the above fact as $v :: D$ and say value v is out most constraint by D.

7.2 Constrained Functions

We say a function $f : Val_{\mathcal{S}} \rightharpoonup Val_{\mathcal{S}}$ is out most constrained like $f :: D \to D'$ if each out most constrained closed value $\epsilon \vdash v :: D$ is mapped to an out most constrained value $\epsilon \vdash fv :: D'$. If we know the used variables $n \vdash DD'$ in the constraints we call the function f outer poly constrained if it is outer constrained $fD'' : D \bullet D'' \to D' \bullet D''$ for each constraint $D'' : n$.

7.3 Associated Occurrences

Each value $v \in Val_{\mathcal{S}}$ is a pre-value $v \in T_{\Sigma\mathcal{S}^{Pre}}$ so we define the set of value occurrences associated to a occurrence in an out most constraint.

$\{v|^n o\} \subseteq occ(v)$ where
$\quad \{v|^n \epsilon\} := \{\epsilon\}$
$\quad \{fv|^n io\} :=$ is defined iff $i \in [n_f]$
$\qquad \bigcup_{j \in [|v|]} \bigcup_{o' \in occ(x_i,(D_f)_j)} j.\{v_j|^n o' o\}$

$\{v@^n o\} \subseteq cocc(v)$ where
$\quad \{fv@^n \epsilon\} := \{\epsilon\} \cup$
$\qquad \bigcup_{j \in [|v|]} \bigcup_{o' \in occ(srt_1(f),(D_f)_j)} j.\{v_j@^n o' o\}$

$\quad \{fv@^n io\} :=$ is defined iff $i \in [n_f]$
$\qquad \bigcup_{j \in [|v|]} \bigcup_{o' \in occ(x_i,(D_f)_j)} j.\{v_j@^n o' o\}$

7.3.1 Well-Constrained Occurrences

Given an out most constrained value $v :: D$ and an occurrence o which is a place in the constraint $o \in occ(D)$. In this case the sub-values at the occurrences $o' \in \{v|^n o\}$ are out most constraint such as $(v|o') :: (D|o)$. We naturally extend the notion $\{v|^n O\}$ of value occurrences associated to a set of constraint occurrences $O \subseteq occ(D)$ iff for each two occurrences $o_1, o_2 \in O$ induce the same constraints $D|o_1 = D|o_2$.

7.3.2 Well-Sorted Occurrences

Given an out most constrained value $v :: D$ and a constructor $f \in G\mathcal{S}_n$. We call the constructor $E := srt_1(f) \in G\mathcal{S}_0 = |\Sigma\mathcal{S}|$ the outer sort of the constructor f (in short $f ::^s E$). Given a constructor occurrence $o \in cocc(D)$ then the associated constructor occurrences $o' \in \{v@^n o\}$ are outer sorted such as $(v@o') ::^s (D@o)$.

7.4 n^{th}-Order Map

A general map $m_{D''}$ can be defined for a outer constraint $D'' \in n \downarrow T_{\Sigma\mathcal{S}}$ and a family of out most constrained functions $\vec{f} := \{f_i :: D_i \to D'_i\}_{i \in [n]}$ We get a out most constrained function $m_{D''}\vec{f} : D'' \bullet D \to$

$D'' \bullet D'$ which operates on values $v :: D \bullet D'$ as follows by m-family substitution (see 5.4.1) :

$$m_{D''} \vec{f} v := (v|_m(\vec{f}, oindx))$$

We define occurrence sets associated to the variables occuring in constraint D'' by setting $O_i := \{v|^n occ(x_i, D'')\}$. So we get the canonical occurrence index function $oindx : O_1 \cup \cdots \cup O_n \to [n]$.

7.5 n^{th}-Order Fold

To define a general fold we need as ingredients a constraint constructor $E : n \in \Sigma\mathcal{S}$ an outer constraint $D \in T_{\Sigma\mathcal{S}}(n, 1)$ and a family of outer poly constrained functions $\vec{g} := \{g_T :: D^r_T \to D\}_{T \in G\mathcal{S}^E_1}$ where $G\mathcal{S}^E_1 := \{T \in G\mathcal{S}_1 \mid st_1(T) = E\}$ and $D_T = agt_1(T)$ for $T \in G\mathcal{S}_1$. Further the reduced constraint D^r_T is given by definition $D^r_T := (D_T|_f T \mapsto D)$. We get a outer poly constrained function $f_{E,D}\vec{f} : E\underline{n} \to D$ which operation of a value $v :: ED'$ is given by f-family substitution (see 5.4.2) :

$$f_{E,D}\vec{g}v := (v|_f \{v@^n \epsilon\} \mapsto (\vec{g}, srt_2))$$

The sorting function $srt_2 : G\mathcal{S}_n \to G\mathcal{S}_1$ is used as the constructor index. We know for each constructor occurrence $o' \in \{v@^n \epsilon\}$ the outer sorting $v@o' = fv' :^s E$. This sorting implies $srt_2(f) \in G\mathcal{S}^E_1$ which guarantees well-definedness.

8 Conclusion

8.1 Related Work

The ideas of this paper are clearly initiated on the functional programming side by the work of Bird [Bir87]. The generalisation in notions of cata-, ana-, hylo- and para-morphisms in [MFP91] make the connection to category theory and domain theory explicit. While the above work promotes the discipline of algebraic programming encouraging to write programs using only the provided iteration schemes, functorial ML [BJM96] introduces categorical notions into languages such as ML. In functorial ML general algorithms for fold and map are introduced but just for regular adt's. We were able to state general algorithms working also with non-regular adt's.

The strand of work on polytypic programming [Mee96] emphasises that some functions can be written for all types. Map and fold are examples for polytypic functions. The results of this paper can be seen as first examples of higher order polytypic programming.

On the mathematical side the paper was inspired by Baez and Dolan's program of higher dimensional algebra [BD95]. The foundation of this work is a hierarchy of weak n-categories, the definition of which is current research [BD98, HMP98, Bat98]. The free theories used in this paper are instances of the topological motivated notion of operads which are central structures in above work. The formal connection with these structures is current research of the present author.

8.2 Results

We presented the novel programming construct of higher order algebraic data types. This new construct is presented as the hierarchy of n^{th}-order adt's. At level zero live signatures and free theories. At the first level live the adt's known from ML and Haskell. At the second level we introduced constraint adt's which are an instance of constraint type systems [SOW98]. Associated with the hierarchy of adt's we were able to present a type inference algorithm together with a minimal typing property. Further we defined the general map and fold algorithms living in this hierarchy. In order to define these algorithms the non-regular fold has been invented.

8.3 Future Work

The above results raise naturally a range of questions about extending the new programming construct introduced here : How to extend adt's to allow use of function types in specifications ? How to mix higher order functions and adt's (that is the introduction of lambda abstraction) ? Introduction of first-order (synonym predicative) poly-n-morphism with associated let-constructs would also be nice. Finally the introduction of impredicative polymorphism in the style of Läufer, Odersky [OL96] or Jones [Jon97] would lead to a nice general picture of typefull programming.

Acknowledgements

I want to thank my colleagues who helped to improve the presentation and my poor English. Thanks to Daniele Turi, Martin Sulzmann, Alex Simpson, Ralf Schweimeier, Samson Abramsky, Martin Odersky and Julo Chroboczek.

References

[Bat98] M.A. Batanin. Monoidal globular categories as a natural environment for the theory of weak n-categories. *Adv. Math.*, 136:39–103, 1998.

[BD95] J. Baez and J. Dolan. Higher-dimensional algebra and topological quantum field theory. *Jour. Math. Phys.*, 36:6073–6105, 1995.

[BD98] J. Baez and J. Dolan. Higher dimensional algebra iii: m-categories and the algebra of opetopes. *Adv. Math.*, 135:145–206, 1998.

[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.

[HMP98] C. Hermida, M. Makkai, and J. Power. Higher dimensional multigraphs. In *LICS '98 Proceedings*, volume 13, pages 207–213, Indianapolis, Indiana, June 1998. IEEE, ieee press.

[Jon97] Mark P. Jones. First-class polymoprhism with type inference. In *POPL*, pages 483–496. acm press, 1997.

[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.

[OL96] Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *Proc. 23rd ACM Symposium on Principles of Programming Languages*, pages 65–67, January 1996.

[SOW98] Martin Sulzmann, Martin Odersky, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, pages 75–103, to appear 1998.

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

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