ON THE IMPLEMENTATION OF ABSTRACT DATA TYPES BY PROGRAMMING LANGUAGE CONSTRUCTS

Axel Poigné
Dept. of Computing
Imperial College
London SW7 2BZ

Josef Voss
Abt. Informatik
Universitaet Dortmund
Postfach 500500
D-4600 Dortmund 50

ABSTRACT. Implementations of abstract data types are defined via an enrichment of the target type. We suggest to use an extended typed λ-calculus for such an enrichment in order to meet the conceptual requirement that an implementation has to bring us closer to a (functional) program. Composability of implementations is investigated, the main theorem being that composition of correct implementations is correct if terminating programs are implemented by terminating programs. Moreover we provide syntactical criteria to guarantee correctness of composition.

0. INTRODUCTION

The concept of abstract data types (ADTs) has pushed forward the investigations for a systematic and formal software design. The given problem is made precise as a set of data with operations on it. The way from the problem to a first exact description is often difficult and beyond formal methods. But a lot of work has been spent on ADTs and algebraic specifications and their relationship to programming languages in the last years. On the one hand the theory is involved in structuring large ADTs resp. specifications (parameterization), on the other hand the stepwise refinement of non-algorithmic specifications in direction of a higher order programming language (implementations) is investigated.

This paper is about implementation. There are two points of view how to deal with this subject: a purely semantical reasoning as in the embracing work of Lipeck [7], or a syntax-oriented reasoning on specifications, especially algebraic ones with an initial algebra semantics in mind. The latter approach is taken by several authors like Ehrig, Kreowski, ADJ-group, Ehrich,Ganzinger,... Their recent work investigates convenient correctness criteria, compatibility of parameterizations and implementations, and extensions to wider classes of specification techniques.

We join the latter approach and consider yet another notion of implementation of algebraic specifications. There are two conceptual requirements we attach importance to:

1. An implementation of SPEC0 by SPEC1 has to bring us closer to a program for SPEC0.
2. There has to be a natural way to compose implementations syntactically such that the correctness criteria are preserved.

It has been Ganzinger [6] who used the word 'program' for certain enrichments that may

be used for implementations. They were characterized by some semantical conditions but do not look like programs at all. We extend his approach and specify enrichments as programs where new sorts are introduced by domain equations, and new operators are introduced as λ-terms. In that, we restrict enrichments to special, often appearing patterns like products, sums, tree etc. for sorts, and to operator definitions using case-distinction and recursion.

In chapter 1 we introduce an extended typed λ-calculus over a base specification to denote our programs. Several properties of the calculus are investigated which will be used for our main theorem that composition of correct implementations is correct if terminating programs are implemented by terminating programs. The termination condition may be natural from a programmer's point of view but the difficulty of the proof seems to be rather surprising. This may cast some light on the difficulty to find sufficient but not constraining conditions to ensure correctness of composition of the more general notion of implementation used in abstract data type theory.

We assume that the reader is familiar with abstract data type theory. For the introduction of signatures, terms etc. we informally use sorted sets $M = (M_i \mid i \in I)$. A *signature* is a pair (S,Σ) with a set S of sorts and an $S^* \times S$ -sorted set Σ of operators. We use $\sigma: w \to s$ to denote operators with arity w and coarity s. Variables are created from a fixed countable set X by indexing. $x:s$ is a variable of sort s. If possible sort indices are omitted. $T_\Sigma(Y)$ denotes the S-sorted set of λ-*terms with variables* Y being defined as usual (Y is a S-sorted set of variables). Syntactical equality is denoted by $\equiv$. Substitution is defined as usual. A Σ-*equation* is a pair (t,t') with $t,t' \in T_\Sigma(Y)$ of the same type. A *specification* is a tripel (S,Σ,E) with a signature (S,Σ) and a set E of Σ-equations. The congruence on $T_\Sigma(Y)$ generated by the equations E is denoted by $=$.

We use the following notation, similar to CLEAR [2]:

```
ZFOUR0 = sorts Z4
         ops 0,1,2,3: → Z4
             +,-,*: Z4 Z4 → Z4
             pred, suc: Z4 → Z4
         eqns 0 * 1 = 0    0 + 1 = 1    0 - 1 = 3    pred(0) = 3    suc(0) = 1
              0 * 2 = 0    0 + 2 = 2    0 - 2 = 2          ⋮              ⋮
                   ⋮            ⋮            ⋮        pred(3) = 2    suc(3) = 0
              3 * 2 = 2    3 + 2 = 1    3 - 2 = 1
              3 * 3 = 1    3 + 3 = 2    3 - 3 = 0

ZFOUR1 = enrich ZFOUR0 by sorts -
                          opns -
                          eqns (z * 2)* 2 = 0
```

(These specifications of $\mathbb{Z}/4$ which will be used below, but do not constitute what we consider to be a typical specification). Apart from the notation we will as well use the nomenclature of [4].

1. PROGRAMS OVER A SPECIFICATION

1.1 MOTIVATION

Enrichments of specifications occur if we construct complex specifications out of smaller ones, or in implementations as studied by Ehrig et al. [3,4]: SPEC = (S,Σ,E) is extended to SPEC' = SPEC + (S',Σ',E') where the added part (S',Σ',E') need not to be a specification. The partition is used to structure the specification. Thus SPEC and SPEC' should depend on each other in an easy way. There are different notions to catch this semantically:

1. *Consistency*: No identifications of old constants: $t,t' \in T_\Sigma(\emptyset)$, $t =_{E+E'} t'$ $\Rightarrow$ $t =_E t'$.
2. *Completeness*: No new constants on old sorts: $t \in T_{\Sigma+\Sigma'}(\emptyset), s \in S \Rightarrow \exists t' \in T_\Sigma(\emptyset)$: $t =_{E+E'} t'$.
3. *Persistency*: 1. and 2. hold for terms with variables of S-sorts.

Consistency and completeness together guarantee the protection of the SPEC-part in the enrichment. Persistency is stronger in that the introduction of so-called derivors is allowed on S-sorts only. To check wether one of the conditions holds is difficult, in general undecidable.

One observes that over and over again the same constructs are used for enrichments. The new sorts represent lists, trees etc, the added equations are (often primitive) recursive schemes to define the new operators. The restriction to those standard constructs yields a syntactical notion of enrichment which is transparent and of course not exactly equivalent to the semantic ones above. The constructs to be permitted will be both, expressable in higher order programming languages and definable by algebraic specifications. One can state that the operators to be defined are recursive programs in an applicative language with recursive data structures, which uses the given specification, resp. the thereby defined ADT, as a kernel of basic data structures, and functional procedures on them. For new operators a λ-notation will be introduced which underlines the affinity to languages like LISP. The defining equations are replaced by rewrite rules of a typed λ-calculus.

We may lead a longer discussion about the definite choice of constructs to be used: What is typical for higher order PL's and algebraic specifications? What constructs are at least needed? Therefore the following choice is somewhat arbitrary. Some nice properties to be proved below may justify it.

The language Λ contains the following elements:

1. *Products*: In higher PL's products appear as records or classes. In algebraic specifications we write

```
PROD = enrich SPEC by sorts prod-a-b
                      ops  p: prod-a-b → a
                           q: prod-a-b → b
                        pair: a b → prod-a-b
                      eqns p(pair(x,y)) = x,  q(pair(x,y)) = y
                                pair(p(z),q(z)) = z
```

2. *Sums*: Variant records in PASCAL and subclasses in SIMULA correspond to sums. In a specification we write

SUM = enrich SPEC by sorts sum-a-b
ops inl: a → sum-a-b
inr: b → sum-a-b

Besides the embeddings we need a means to define functions on the sum by case distinction. In PASCAL we have the case statement. For specifications we use

SUMgh = enrich SUM by ops f: sum-a-b → c
eqns f(inl(x)) = g(x), f(inr(y)) = h(y)

where we presume that a,b,c, g:a → c, h:b → c are in SPEC.

3. *Recursive Types*: In PASCAL we can describe recursive data structures using recursive schemes of records and variant records. If a PL does not allow this a controlled use of pointers can help. As a means for the description of recursive types we introduce domain equations, for example

tree = 1 + (tree × entry × entry)

expr = term × operator × term
term = identifier + expr

Again entry, operator and identifier are given sorts. Senseless schemes like d = d × d are excluded.

4. *Recursion*: This is the essential construct which, in combination with the case distinction, allows to write non-trivial programs, but brings along the problems of non-termination. In specifications recursive schemes are those definition schemes which for each new operator symbol σ have one equation with $\sigma(x_0,\ldots,x_{n-1})$ on the left and an arbitrary right hand side. In our language we will use a fixpoint operator to denote recursive operator definitions.

1.2 RECURSIVE TYPES OVER BASE SORTS

For the set S of a given specification (S,Σ,E) we construct products, sums and recursive types as congruence classes of sorts terms over S:
Let DTn(S) denote sort terms with type variables $d_0,\ldots,d_{n-1}$ constructed as the smallest set such that

1. $S \subseteq DTo(S)$, $1 \in DTo(S)$
2. $DTn(S) \subseteq DTm(S)$ for $n \leq m$
3. $d_i \in DTi(S)$ $i \geq 1$
4. $t,t' \in DTn(S) \Rightarrow t + t',\ t \times t' \in DTn(S)$.

Now take the recursive type scheme

$$d_0 = t_0(d_0,\ldots,d_{n-1})$$
$$\vdots$$
$$d_{n-1} = t_{n-1}(d_0,\ldots,d_{n-1})$$

with n variables and n equations. We introduce names for the n solutions of this scheme by $D_n^i(t_0,\ldots,t_{n-1})$, $i \in \underline{n}:=\{0,\ldots,n-1\}$. We get arbitrarily nested schemes if we regard these solutions as new constants. Thus the defition of DTn(S) is completed by the line

5. $t_0,\ldots,t_{n-1} \in DTn(S) \Rightarrow D_n^i(t_0,\ldots,t_{n-1}) \in DTo(S)$

To obtain the $D_n^i(t_o,\ldots,t_{n-1})$ as the solution of the respective recursive scheme the type terms are to be factorized by the least equivalence relation $\sim$ containing

1. $D_n^i(t_o,\ldots,t_{n-1}) \sim t_i[d_o \leftarrow D_n^o(t_o,\ldots,t_{n-1}),\ldots,d_{n-1} \leftarrow D_n^{n-1}(t_o,\ldots,t_{n-1})]$ for $i \in \underline{n}$
2. $t_i \sim t_i'$, $i \in \underline{2} \Rightarrow t_o + t_1 \sim t_o' + t_1'$, $t_o \times t_1 \sim t_o' \times t_1'$
3. $t_i \sim t_i'$, $i \in \underline{n} \Rightarrow D_n^j(t_o,\ldots,t_{n-1}) \sim D_n^j(t_o',\ldots,t_{n-1}')$ for $j \in \underline{n}$

In fact, we state that $\sim$ is a congruence. Hence the operators $\_+\_$, $\_\times\_$ and $D_n^i(\ldots)$ are well defined on equivalence classes.

It is reasonable to restrict our attention to *acceptable types*, i.e. types whith non-empty solution ($d = d+d$ is not useful). We define acceptable types to be those types t such that $t \twoheadrightarrow 1$ with regard to (where $\twoheadrightarrow$ is the refl.,trans.,substitutive & compatible closure of $\rightarrow$).

$s \rightarrow 1$ for $s \in S$

$t \rightarrow 1$ or $t' \rightarrow 1 \Rightarrow t+t' \rightarrow 1$

$t \rightarrow 1$ and $t' \rightarrow 1 \Rightarrow t \times t' \rightarrow 1$

$D_n^i(t_o,\ldots,t_{n-1}) \rightarrow t_i[d_i \leftarrow D_n^i(t_o,\ldots t_{n-1}), i \in \underline{n}]$.

Remarks: 1. The whole calculus in the rest of the paper is essentially the same if we do not restrict to acceptable types, but the proof technique has to be extended sometimes. For instance the following property will be used:

2. The maximal decomposition of an acceptable type into products is finite.

Definition: Let $ATn(S) \subseteq DTn(S)$ denote the set of acceptable types with at most n variables.

The set of *base types* over S is given by $BType(S) := ATo(S)_{/\sim}$

The set of (*higher order*) *types* is defined to be the smallest set Type(S) with

1. $BType(S) \subseteq Type(S)$
2. $t,t' \in Type(S)$, t or $t' \notin BType(S) \Rightarrow t+t', t\times t' \in Type(S)$
3. $t,t' \in Type(S) \Rightarrow t \rightarrow t' \in Type(S)$.

1.3 THE PROGRAMMING LANGUAGE Λ

Compound operations will be denoted by Λ-terms. The set Type(S) is used to type the terms where S is a given set of sorts. The set FV(t) of free variables of a Λ-term t is defined simultaneously:

For a given signature (S,Σ) and a set of variable names X we define the language $\Lambda_\Sigma(X)$, or for short Λ, to be the smallest Type(S)-sorted set with

1. $x \in X$, $s \in Type(S) \Rightarrow x:s \in \Lambda_s$	$FV(x:s) := \{x:s\}$
2. $\emptyset \in \Lambda_1$	$FV(\emptyset) := \emptyset$
3. $t_i \in \Lambda_{s_i}$, $i \in \underline{n}$, $\sigma:s_o\ldots s_{n-1} \rightarrow s \in \Sigma$ $\Rightarrow \sigma(t_o,\ldots,t_{n-1}) \in \Lambda_s$	$FV(\sigma(t_o,\ldots,t_{n-1})) := \bigcup_{i \in \underline{n}} FV(t_i)$
4. $inl_{s,s'} \in \Lambda_{s \rightarrow s+s'}$, $inr_{s,s'} \in \Lambda_{s' \rightarrow s+s'}$	$FV(inl_{s,s'}) := FV(inr_{s,s'}) := \emptyset$

5.	$p_{s,s'} \in \Lambda_{s\times s' \to s'}$, $q_{s,s'} \in \Lambda_{s\times s' \to s'}$	$FV(p_{s,s'}) := FV(q_{s,s'}) := \emptyset$
6.	$t_i \in \Lambda_{s_i}$, $i \in \underline{2}$ => $\langle t_0,t_1\rangle \in \Lambda_{s_0 \times s_1}$	$FV(\langle t_0,t_1\rangle) := FV(t_0) \cup FV(t_1)$
7.	$t,t' \in \Lambda_{s''}$ => case x:s.t, y:s'.t' esac $\in \Lambda_{s+s' \to s''}$	FV(case x:s.t,y:s'.t' esac) := FV(t)\{x:s} $\cup$ FV(t')\{y:s'}
8.	$t \in \Lambda_{s'}$ => $\lambda x{:}s.t \in \Lambda_{s \to s'}$	$FV(\lambda x{:}s.t) := FV(t)\backslash\{x{:}s\}$
9.	$t \in \Lambda_{s \to s'}$, $t' \in \Lambda_s$ => $(t\ t') \in \Lambda_{s'}$	$FV(t\ t') := FV(t) \cup FV(t')$
10.	$Y_s \in \Lambda_{(s \to s) \to s}$	$FV(Y_s) := \emptyset$

Substitution is defined as usual in λ-calculus. We consider terms modulo α-conversion [1]. For convenience indices are omitted if the typing is obvious from the context. As standardization we assume that $\emptyset$ is the only term of type 1.

Examples: For better readability a more general notation is allowed in that we use many-fold sums and products, more than one parameter for abstractions and case-statements, omit brackets, and write t(t') instead of (t t').

We define some 'functions' on non-empty lists and trees over a sort entry:

list = entry + (entry × list)
tree = entry + (entry × tree) + (tree × entry) + (tree × entry × tree)

Attaching to the left side of a list is given by

latt ≡ λe,l.inr <e,l>

For attaching to the right side we write a recursive program

ratt ≡ Y(λf.λl,e.case ee.inr<ee,inl(e)>, ee,ll.inr<ee,f<ll,e>> esac (l))

Other functions on lists and trees are

conc ≡ Y(λf.λl,l'.case e.ratt(l,e), e,l".f(ratt(l,e),l") esac (l'))

inorder ≡ Y(λf.λb.case e.inl(e), e,b'.inr<e,f(b')>, b',e.ratt(f(b'),e), b',e,b".conc(f(b'),inr<e,f(b")>) esac (b))

If we add some syntactical sugar - for instance replacing fixpoint operators by recursive procedures and using type declarations - we would get a more or less standard procedural language. But for proof theoretic reasons we prefer the more clumsy Λ-notation.

1.4 REDUCTIONS ON Λ

We use a reduction system to define the operational semantics of our language. It should be remarked that an equivalent algebraic semantics can be defined [8]. Hence all arguments hold in a purely algebraic framework compatible with abstract data type theory. But in proofs we heavily rely on operational properties.

We define a special notion of reduction denoted by $Y\beta\eta E$. The equations E of the underlying specification (S,Σ,E) are understood as rewriting rules from left to right. We assume the following restrictions on E:

(E1) $(t,t') \in E$ => $FV(t') \subseteq FV(t)$

(E2) $\overline{E}$ is Church-Rosser ($\overline{E}$ trans.,refl.,substitutive & compatible closure of E).

(E3) $(t,t') \in E \Rightarrow x \in FV(t)$ appears only once in t

(E4) $(t,t') \in E \Rightarrow$ t is not a variable.

In addition we require that $T_\Sigma(\emptyset)_s$ is non empty for all $s \in S$.

Definition: $\rightarrow$ is the smallest (Type(S)-sorted) relation on Λ such that

E 1. $(t,t') \in E$, $t_i \in \Lambda_\Sigma(X)_{s_i}, i \in \underline{n} \Rightarrow t[x_i:s_i \leftarrow t_i, i\in\underline{n}] \rightarrow t'[x_i:s_i \leftarrow t_i, i\in\underline{n}]$

β
2. $(\lambda x:s.t)t' \rightarrow t[x:s \leftarrow t']$
3. case x:s.t,x':s'.t' esac (inl t") $\rightarrow$ t[x:s $\leftarrow$ t"]
   case x:s.t,x':s'.t' esac (inr t") $\rightarrow$ t[x':s' $\leftarrow$ t"]
4. p <t,t'> $\rightarrow$ t q <t,t'> $\rightarrow$ t'

Y 5. $(Yt) \rightarrow t(Yt)$

η
6. $\lambda x:s.(tx) \rightarrow t$ if $x:s \notin FV(t)$, $\lambda x:1.(t\emptyset) \rightarrow t$
7. <pt,qt> $\rightarrow$ t

We use $\twoheadrightarrow$ to denote the reflexive, transitive and compatible (with the structure) closure of $\rightarrow$. If we refer to a specific subset of rules we index by $\rightarrow_{Y\beta E}$, $\rightarrow_E$, $\rightarrow_6$,...

Example: According to the generalized notations for terms we have generalized reductions. For the constant T ≡ in2<e1,in4<in1(e3),e2,in1(e4)>> where e1,...,e4 are given constants of type entry, we reduce the term 'inorder(T)':

inorder(T) $\rightarrow$ (λf.λb.case...esac(b))(inorder)(T)

$\rightarrow$ λb.case e.in1(e), e,b".in2<e,inorder(b")>, b'e.ratt(inorder(b'),e), b',e,b".conc(inorder(b'),in2<e,inorder(b")> esac (b)(T)

$\twoheadrightarrow$ case...esac (in2<e1,in4<in1(e3),e2,in1(e4)>>)

$\twoheadrightarrow$ in2<e1,inorder(in4<in1(e3),e2,in1(e4)>)>

$\twoheadrightarrow$ in2<e1,conc(inorder(in1(e3),in2<e2,inorder(in1(e4))>)>

$\twoheadrightarrow$ in2<e1,conc(in1(e3),in2<e2,in1(e4)>)>

$\twoheadrightarrow$ in2<e1,in2<e3,in2<e2,in1e4)>>>

Remarks on the use of the equations E:

1. The essential use of the equations in the calculus is as 'stop-equations'. Besides β-reductions E-reductions are able to eliminate Y's and thereby stop a recursive calculation.
2. It is not realistic to require (E1)-(E4) to hold for all specifications. But we can take the following point of view: The use as stop-equations is a kind of error recovery. Like other authors we may distinguish a certain subset of E to be allowed for this purpose. Only these special equations may be used outside $T_\Sigma(X)$ in that arbitrary terms (especially those with Y's) are substituted for the variables. Then we require (E1)-(E4) only to hold for this subset.

As usual Church-Rosser property (CR), weakly Church-Rosser (WCR), finiteness of reductions are introduced. For illustration of such properties we use commuting diagram [1]. Unbroken lines stand for given reductions, dashed lines indicate when existence of reductions is claimed.

1.5 PROPERTIES OF THE CALCULUS

In this section we consider several properties of reductions which will be useful in proofs on composition of implementations. Because of lack of space only proof ideas can be given. For full proofs the reader is refered to [9] (or to [8] for an extended version).

Terms of special interest are those which are equivalent to a Y-free term, especially those of base types.

Definition: $t \in \Lambda_\Sigma(X)$ is called *terminating* :<=> $\exists t' \in \Lambda_\Sigma(X)$: no Y occurs in t' & $t \simeq t'$

Closed terms of base types are called *base constants* (BC).

($\simeq$ is the symmetric closure of $\twoheadrightarrow$)

Next we distinguish a class of Y-free BCs of a very simple form.

Definition: BCs only built up by inl,inr, <_,_>, $\emptyset$ and $T_\Sigma(\emptyset)$ are called *normal forms* (NF).

Facts: All base types have normal forms. Here we use that $T_\Sigma(\emptyset)_s$ is non empty, and the restriction to acceptable sort terms.

It is undecidable wether a term is terminating or not.

Next we will state some properties of the calculus in general, and then some special results about terminating BCs. It is essential for the proofs that there is no mixing-up of types, especially that a function space is not a product or a sum, and that the requirements (E1)-(E4) hold for equations. The proofs are restricted to terms of acceptable types but the proofs can be extended (compare [8]).

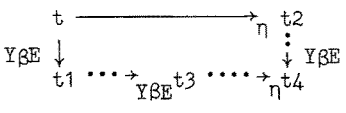
1.1 *Proposition*: β-reductions are finite. $Y\beta E$ is CR.

For the first statement we adapt the proof of Gandy [5]. For the Church-Rosser property use generalized (simultaneous) 1-step-reductions $\triangleright$ as in [1]. To avoid difficulties with E-reductions we generalize E-redices to maximal connected Σ-parts of a term and regard $\twoheadrightarrow_E$-reductions as one step.

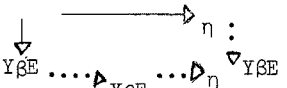
1.2 *Proposition*: η-reductions can be shifted to the end: $t \twoheadrightarrow_{Y\beta\eta E} t' \Rightarrow \exists t'': t \twoheadrightarrow_{Y\beta E} t'' \twoheadrightarrow_\eta t'$

Introduce a generalized 1-step-reduction $\triangleright_\eta$ .Then check $t \triangleright_\eta t' \to_{Y\beta E} t''$ implies $t \twoheadrightarrow_{Y\beta E} t''' \triangleright_\eta t''$ by case distinction on the origin of the $Y\beta E$-redex in t.

1.3 *Proposition*:



By a more complex redex marking we can show which yields the result by a diagram chase together with 1.1 and 1.2.



(1.4 *Proposition*: $\beta\eta$ is finite CR)

1.5 *Proposition*: t terminating => ∃t': t' Y-free and $t \twoheadrightarrow_{Y\beta E} t'$.

We have a Y-free tn and reductions $t \equiv t0 \twoheadleftarrow t1 \twoheadrightarrow t2 \twoheadleftarrow \ldots \twoheadrightarrow tn$. With 1.1 and 1.3 we stepwise construct shorter chains beginning at the right side. We do not need η-reductions as they (by 1.3 at the end of a reduction) cannot eliminate Y's.

1.6 *Proposition*: t terminating BC => ∃t': t' in NF and $t \twoheadrightarrow_{Y\beta E} t'$

Because of 1.5 it is sufficient to show that Y-free BCs are β-reducable to a term without Y, ,case,p,q. Now by a case distinction prove that a Y-free BC contains a β-redex as long as it contains a λ,case,p or q.

1.7 *Proposition*: t terminating BC => ∃t': t' in NF and $t \twoheadrightarrow_{Y\beta} t'' \twoheadrightarrow_{E} t'$.

We have to show that in the situation $t \to_E t1 \twoheadrightarrow_{Y\beta} t2 \twoheadrightarrow_E t3$ with t3 in NF, E-reductions can be shifted to the right. If t3 is in normal form then we can assume that Y-reductions are of such a form that they consist only of $\to_Y$ and $\to_\beta$ steps of maximal breadth. Such reductions treat syntactical equivalent subterms of t1 in the same way. Then the E- and the Y-reductions are exchangeable ($\twoheadrightarrow_E$ may become a $\to_E$-step).

(1.8 *Proposition*: t,t' in NF, $t \simeq t'$ => $t =_E t'$)

1.4 is used for the proof of 1.9 which we do not need for the following. But 1.8 proves that the enrichment of $T_\Sigma(X)$ to $\Lambda_\Sigma(X)$ is consistent. But there are new constants on S-sorts (at least Y(λx:s.x), hence the enrichment is not complete nor persistent.

2. IMPLEMENTATIONS

2.1 MOTIVATION AND DEFINITION

The notion of implementation makes the idea of stepwise refinement precise. Program development is the construction of a hierarchy of specification levels with decreasing abstractness. An implementation builds a bridge between two neighbouring levels with the aim to come closer to a program. If we have two specifications SPEC0 and SPEC1, an implementation of SPEC0 by SPEC1 should preserve correctness of SPEC0-programs. We might call this idea 'relative programming'; the program is developed on the SPEC0-level but run on the SPEC1-level. Implementations are given by SPEC1-data structures and SPEC1-programs implementing SPEC0-sorts and SPEC0-operators respectively. This proceeding seems to capture the task of a programmer who has to write a program realizing a data type.

Definition: An *implementation* of SPEC0 = (S0,Σ0,E0) by SPEC1 =(S1,Σ1,E1) is given by a pair of maps $I = (I_S: S0 \to ATo(S1), I_\Sigma: \Sigma 0 \to \Lambda_{\Sigma 1})$ s.t. $I_\Sigma(\sigma) \in \Lambda_{s_o \times \ldots \times s_{n-1} \to s}$ if $\sigma: s_o \ldots s_{n-1} \to s \in \Sigma$. We extend I_Σ to all terms in $T_\Sigma(X)$ by

$I_\Sigma(x:s) := x:\tilde{I}_S(s)$ $\qquad I_\Sigma(\sigma(t_0,\ldots,t_{n-1})) := T_\Sigma(\sigma) < I_\Sigma(t_o),\ldots,I_\Sigma(t_{n-1}) >$

where $\tilde{I}_S := \Pi \circ I_S$ with the factorization Π: ATo(S1) → BType(S1).

Example: Assume that we have a standard specification of stacks and arrays [3,4]. Stacks are implemented by arrays plus a pointer as follows

ARRAY impl STACK by

sorts stack = array × nat

ops push = λs:stack,n:nat.<add(p(s),suc(q(s)),n),suc(q(s))>
pop = λs:stack.<p(s),pred(q(s))>
empty = <nil,0>
top = λs:stack. p(s)[q(s)]

(the notation hopefully is self-explaining. Sorts and operators which are implemented identically are omitted).

The given syntactical definition has to be completed by semantical constraints which express the correctness of an implementation. We of course intend that the given example is correct. The example illustrates two features of the notion of correctness to be defined:

1. We allow manifold representation of data. An element of type stack may be represented by different elements of type array × nat, especially by arrays which differ in components above the pointer.
2. Not all elements of the implementing data type are used. In the example arrays with non-trivial entries under inder 0 are not used to represent stacks.

Definition: I is called *correct* iff

1. I is *consistent* :<=> $I_\Sigma(t) \simeq I_\Sigma(t')$ implies $t =_{E0} t'$ for all $t,t' \in T_{\Sigma 0}(\emptyset)_s, s \in S0$
2. I is *terminating*:<=> $I_\Sigma(t)$ is terminating for all $t \in T_{\Sigma 0}(\emptyset)$.

There is a close connection to the notion of correctness in the work of EKP [3], especially to their 'term version'. Consistency corresponds to their RI-correctness, and preservation of termination to OP-completeness. EKP add the requirement that the SPEC1-part remains unchanged in SORTIMPL. In our approach we have to examine what happens to T_Σ-terms in Λ_Σ. Property 1.9 guarantees that there are no additional identifications on $T_\Sigma(\emptyset)$-terms. On the other hand the only new terms on S-sorts that are not equivalent to $T_\Sigma(\emptyset)$-terms are non-terminating ones. But those we regard as error-programs which should not be used for implementations. In a more recent version [4] EKP restrict their SORTIMPL to special patterns which describe exactly those types over S which can be defined by recursive domain equation schemes in our approach. The equations EKP allow in their OPIMPL specification to implement operators are much more general than our recursive programs.

2.2 COMPOSABILITY

The most important property expected to hold for implementations is the composability of the single steps to one large implementation which then yields a computable program for every operator of the very first specification level.

We want to compose implementations I1 of SPEC0 by SPEC1 and I2 of SPEC1 by SPEC2.

Syntactically we intend the following: A $\Sigma 0$-operator $\sigma: w \to s$ has the SPEC1-implementation $I1(\sigma)$. Now replace all $\Sigma 1$-symbols in $I1(\sigma)$ by their SPEC2-implementations under I2. We obtain a SPEC2-program which is the implementation of σ in SPEC2.

For this purpose we have to extend I2 to all terms of $\Lambda_{\Sigma 1}(X)$. The composition of this extension with I1 then yields the implementation of SPEC0 by SPEC1.

Let $I = (I_S, I_\Sigma)$ be an implementation of SPEC0 by SPEC1. We extend I_S in the obvious way to I_S: ATo(S0) $\to$ ATo(S1). This defines a mapping I_S: BType(S0) $\to$ BType(S1) (using congruence properties) which finally extends to I_S^*:Type(S0) $\to$ Type(S1).
We extend I_Σ to I_Σ^*: $\Lambda_{\Sigma 1}(X) \to \Lambda_{\Sigma 2}(X)$ by

$I_\Sigma^*(x:s) := x:I_S^*(s) \qquad I_\Sigma^*(\emptyset) := \emptyset$
$I_\Sigma^*(\sigma(t_o,\ldots,t_{n-1})) := I_\Sigma(\sigma)\langle I_\Sigma^*(t_o),\ldots,I_\Sigma^*(t_{n-1})\rangle$
$I_\Sigma^*(p_{s,s'}) := p_{I_S^*(s),I_S^*(s')}$ and similar for q, inl, inr
$I_\Sigma^*(\lambda x{:}s.t) := \lambda x{:}I_S^*(s).I_\Sigma^*(t)$
and so on preserving the structure of programs.

Definition: For given implementations I1 of SPEC0 by SPEC1 and I2 of SPEC1 by SPEC2 the *syntactical composition* $I2.I1$ is defined by $I2.I1_S := I2_S^* \circ I1_S$ and $I2.I1_\Sigma := I2_\Sigma^* \circ I1_\Sigma$.

Fact: I2.I1 is an implementation of SPEC0 by SPEC2.

It should be noted that our notion of composition of implementations is different to that of EKP [4] as there SPEC0 is implemented using SPEC1 as a hidden part of the composed implementation while in our approach the intermediate specification disappears.

Questions: Is the composition of correct implementations correct again?
Do the consistency- and termination-conditions still hold if extended to all terminating BCs?
The answers are in general negative. The following examples shows that the composition is not necessary terminating:

The program p over stacks
$p \equiv Y(\lambda f.\lambda s.pop(push(empty,top(f(s)))))(empty)$
is a constant of type stack and is equivalent to empty. But its implementation $I_\Sigma^*(p)$ as a program over arrays with pointers is not terminating:
$I_\Sigma^*(p) \equiv Y(\lambda f.\lambda s.\langle add(p\langle nil,0\rangle,..\rangle)(\langle nil,0\rangle)$
$\twoheadrightarrow$ $p1 \equiv Y(\lambda f.\lambda s.\langle add(nil,suc(0),p(f(s))[q(f(s))]),0\rangle)$
$\twoheadrightarrow$ $\langle add(nil,suc(0),p(p1)[q(p1)]),0\rangle$
$\twoheadrightarrow$ ...
We never get rid of p1 and the Y's in it.

But we can prove the following

2.1 *Main Theorem*: If I1 and I2 are correct implementations and I2.I1 is terminating then I2.I1 is correct.

We outline the idea of the proof. We use $\simeq$ and $\twoheadrightarrow$ ambiguously for reductions with regard to E1 and E2-equations.

Let $t,t' \in T_{\Sigma 0}(\emptyset)$ with $I2.I1(t) \simeq I2.I1(t')$ both terminating. As I1 is correct, I1(t) and I1(t') are terminating BCs. We have to show that they are equivalent in $\Lambda_{\Sigma 1}(X)$. Then the consistence of I1 yields $t \simeq t'$. Therefore it is suffient to prove

Claim: If I2 is correct, $t,t' \in \Lambda_{\Sigma 1}(X)$ are terminating BCs, $I2(t) \simeq I2(t')$ both terminating, then $t \simeq t'$.

The proof of the claim takes several steps where we use the properties of the calculus stated in chapter 1. For convenience we use I instead of I2 (Remark: The claim states that correctness extends to terminating programs which are implemented by a terminating program with regard to an arbitrary correct implementation).

Step 1: We can assume that all terms are *well-formed* (wf) in that Y's only occur in the form Y(t) (idea: replace all Y's by $YY \equiv Y(\lambda F.\lambda f.f(F(f)))$).

Step 2: In terminating terms we can add arbitrary many Y-reductions:

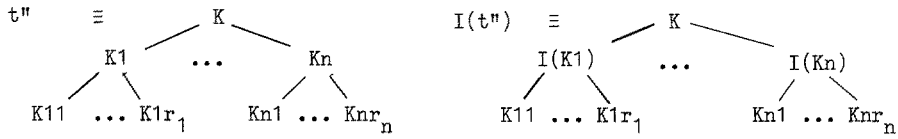
$$t \twoheadrightarrow t' \twoheadrightarrow t'' \quad (t' \twoheadrightarrow_Y t''') \qquad \text{where } t'' \text{ is Y-free.}$$

Step 3: Terminating computations with wf BCs have without restriction of generality the following form: $t \twoheadrightarrow_Y t' \twoheadrightarrow_\beta t'' \twoheadrightarrow_E t'''$ such that (i) t'' is maximal β-reduced, and t''' is in NF.

Step 4: We can synchronize the reduction of t and I(t) as follows:

$$\begin{array}{ccccccc} t & \twoheadrightarrow_Y & t' & \twoheadrightarrow_\beta & t'' & \twoheadrightarrow_{E1} & t''' \\ I(t) & \twoheadrightarrow_Y & I(t') & \twoheadrightarrow_\beta & I(t'') & \twoheadrightarrow_{Y\beta\eta E2} & t^{IV} \end{array} \qquad \text{with}$$

1. $t \twoheadrightarrow t'''$ is given as in step 3, t''' is in NF.
2. There are no Y-reductions of those Y's inherited from t in $I(t'') \twoheadrightarrow t^{IV}$.
3. t'' and I(t'') have the following form



where a) the Ki's only contain $\Sigma 1$-operators, K contains no $\Sigma 1$-operators (hence only Λ-operators inl,inr,<_,_>), and the Kij have a Λ-operator in the root.

b) The terms Kij are no more β-reducible (thus contain a Y)

c) Up to sort indices the terms K and Kij are the same in t'' and I(t'')

d) If we replace the subterms Kij of Ki by suitably typed variables, resulting term being $Ki(\vec{x})$, $Ki(\vec{x})$ is E1-reducible to a $T_{\Sigma 1}(\emptyset)$ term.

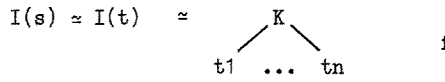
Step 5: In the above situation the terms $I(Ki(\vec{x}))$ terminate. The reason is that the Y's in the Kij's inherited from t'' need not to be reduced in $I(t'') \twoheadrightarrow t'$ and that

there are no β-redexes in the Kij's, hence the Kij's do not interfere with the computations on the I(Ki)'s. Therefore we may replace the Kij's by arbitrary subterms of appropiate sort.

Step 6: - The final argument:

We take the situation of the claim and construct for both s and t the synchronized computations. We have

I(s") and I(t") are reducible to the same term, and Ks ≡ Kt ≡: K (by step 4,3a). Hence



for some ti in NF (use 1.6).

Now replace, as sketched in step 5, the subtrees Ksij and Ktij by type matching terms of $T_{\Sigma 0}(\emptyset)$ (which exists for any sort by general assumtion on SPEC, compare section 1.4). We obtain terms $\bar{s}$ and $\bar{t}$ of the form

where $\overline{si}, \overline{ti} \in T_{\Sigma 0}(\emptyset)$. As the Ki(x) terminate we have $s \simeq \bar{s}$ and $t \simeq \bar{t}$. To show $s \simeq t$ we have to prove the equivalence of $\overline{si}$ and $\overline{ti}$:

$I(\overline{si}) \simeq I(Ksi) \simeq ti \simeq I(Kti) \simeq I(\overline{ti})$. The consistency of I gives us $\overline{si} \simeq \overline{ti}$.

This completes the proof of the main theorem.

Example: (Compare step 4) ZFOURO implements ZFOUR1 (cf. introduction). The constant $(Y(\lambda x.3)(1)*2)*2$ is reducible to 0 with regard to ZFOURO and ZFOUR1. But in ZFOUR1 we need no Y-reduction, in ZFOURO at least one. Thus additional Y-reductions may occur on the implementation level.

The result is not completely satisfactory so far. We would rather have a criterion that can be checked for a single implementation, and which guarantees correctness of composition. The proof of 2.1 gives a hint: It is sufficient to require that if a term t does not depend on one of its arguments x, then the implementation of t does not depend on x as well.

Definition: t is called I-*representative* of t' iff there exists a t" such that $t" \simeq t'$ and $t \simeq I(t")$.

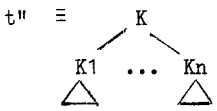
An implementation is called *strong* iff for some I-representative of $t' \in T_{\Sigma 0}(\emptyset)$ there exists a term t" with $t \simeq t"$ and $FV(t") = \emptyset$.

Remark: I is strong if SPECO has only equations with FV(t) = FV(t').

Proposition: Strongness is preserved by composition.

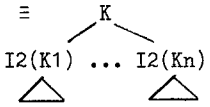
2.2 *Theorem*: If I2 is strong and correct, and I1 is correct then I2.I1 is correct.

We only have to show that I2.I1 is terminating. Take a terminating BC $t \in \Lambda_{\Sigma 1}(X)$. Then there exists a computation $t \twoheadrightarrow t'' \twoheadrightarrow_{E1} t'''$ such that t''' is in NF, and t'' has the form



with $Ki(\vec{x}) \twoheadrightarrow ti \in T_{\Sigma 1}(\emptyset)$

and a computation $I2(t) \twoheadrightarrow I2(t'') \equiv$



$I2(Ki(\vec{x}))$ is a I2-representative of ti. As I2 is strong I2(Ki) does not depend on its subtrees. Again we replace the subtrees by terms from $T_{\Sigma 1}(\emptyset)$. Let the resulting terms be $\overline{ti}$. As I2 is terminating $I2(\overline{ti})$ is terminating and equivalent to I2(Ki). Then the whole term I2(t) is terminating.

Example: The implementation of stacks by arrays is not strong as

t ≡ pop(push(empty,n)) = empty does not depend on n. But its implementation

I(t) ≡ <p<add(p<nil,0>,suc(q<nil,0>),n),... ≃ <add(nil,suc(0),n),0>

depends on n. We can do better and change the implementation of pop to

pop ≡ λs:stack.<add(p(s),q(s),0),pred(q(s))> ,

in that we erase the entry on the top and replace the pointer. Now the implementation is strong (assuming that all entries of the nil-array are 0's).

Remark: Strongness rules out a phenomenon well known in programming: If boolean expressions of a programming language are implemented evaluation strategies are used like "To evaluate 'x and y', first evaluate x. If x evaluates to 'false' then 'x and y' evaluates to false. If x evaluates to 'true' evaluate y". Different evaluation strategies of this kind yield different results with regard to non-termination. As evaluation strategies may be expressed by equations ("false and y = y") choosing different sets of equations to characterize the same (initial) algebra may change the intensional character with regard to 'infinitary' or 'non-terminating' terms. Strongness states that the intensional character of the equations is to be preserved to a certain extend.

CONCLUDING REMARKS AND OUTLOOK

1. An implementation step makes a part of a specification more computable. A typical situation is that SPEC1 is an enrichment of SPEC0, and the enrichment is to be implemented by programs. Considering parameterized data types may support an analysis of such a situation.

2. Other programming language constructs may be added. In [8] we add a fixed boolean sort and if-then-else-fi-constructs for any type with the restriction that any SPEC-term of sort boolean is equivalent to 'true' or 'false'. We then obtain similar results without any restrictions on the equations E of the base specification.

3.As already pointed out our notion of composition is different to that of EKP [4] which to our opinion is somewhat counter intuitive. For instance there the identical implementation of SPECO by SPECO not always is a unit with regard to composition.
4. At a first sight there seems to be little connection to the work of Lipeck [7] but in fact the extension of a data type by recursive data structures is 'conservative' in terms of [7(4.12)] which guarantees compatibility of construction and realization steps [7(4.11)]. Now the termination condition allows to reduce any terminating term to a normal form or, with other words, we prove that the respective functor is conservative.
5. The observation of 4. may indicate a more methodological aspect: The syntax of an implementation should be flexible to allow formalizations close to the given problem. This freedon has the consequence that correctnesss proofs (that functors are conservative) are more complicated. The situation is well known from programming languages.
6. (Added when preparing this version) There seems to be a close connection to [10] where recursive schemes are used as 'programs'. If we add (as in [8]) a fixed boolean sort and if_then_else_fi-operators our notion of programs seem to cover that of [10] (apart from the semantical side conditions). The in [10] indicated conditions for correctness of compositions seems to be a semantic counterpart to strongness. The connection needs further investigation.

REFERENCES

[1] Barendregt,H.: The Lambda Calculus, North-Holland 1981

[2] Burstall,R.M., Goguen,J.A.: Putting Theories Together to Make Specifications, Proc. of 1977 IJCAI MIT Cambridge, 1977

[3] Ehrig,H., Kreowski,H.J., Padawitz,P.: Algebraic Implementation of Abstract Data Types. Concept,Syntax,Semantics and Correctness, Proc. ICALP'80, LNCS 85, 1980

[4] Ehrig,H., Kreowski,H.J., Mahr,B., Padawitz,P.: Algebraic Implementation of Abstract Data Types, TCS 20, 1982

[5] Gandy,R.O.: Proofs of Strong Normalisation, In: To H.B.Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, Academic Press 1980

[6] Ganzinger,H.: Parameterized Specifications: Parameter Passing and Optimizing Implementation, Bericht Nr. TUM-I8110, TU Muenchen 1981

[7] Lipeck,U.: Ein algebraischer Kalkuel fuer einen strukturierten Entwurf von Datenabstraktionen, Dissertation, Ber. Nr.148, Abt. Informatik, Universitaet Dortmund, 1983

[8] Poigné,A., Voss,J.: Programs over Algebraic Specifications - On the Implementation of Abstract Data Types, Ber. Nr. 171, Abt. Informatik, Uni Dortmund, 1983

[9] Voss,J.: Programme ueber algebraischen Spezifikationen - Zur Implementierung von Abstrakten Datentypen, Diplomarbeit,Abt. Informatik, Uni Dortmund, 1983

[10] Blum,E.K., Parisi-Presicce,F.: Implementation of Data Types by Algebraic Methods, JCSS 27, 1983