JOURNAL OF COMPUTER AND SYSTEM SCIENCES 34, 340–376 (1987)

On the Implementation of Abstract Data Types by Programming Language Constructs*

AXEL POIGNÉ

Department of Computing, Imperial College, London SW7 2BZ, United Kingdom

AND

JOSEF VOSS

Abteilung Informatik, Universität Dortmund, Postfach 50 05 00, D-4600 Dortmund 50, West Germany

Received November 1985; revised April 1986

Implementations of abstract data types are defined via enrichments of a target type. We propose to use an extended typed λ-calculus for enrichments 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 result 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. The proof is based on strong normalization and Church–Rosser results of the extended λ-calculus which seem to be of interest in their own right.

INTRODUCTION

The theory of abstract data types (ADTs) has driven forward the investigation of a systematic and formal approach to software design. A lot of work has been spent on algebraic specifications of ADTs and their relationship to programming languages (e.g., [1, 8, 9, 5, 15]). On the one hand, the theory is involved in structuring large ADTs (resp. specifications) using parameterization techniques (as in [9, 11]); on the other hand, stepwise refinement via implementation is investigated (as in [8, 24]).

This paper is about implementation. There are two ways to approach the subject: semantically as in the comprehensive work of Lipeck [14], or by syntax-oriented reasoning on specifications, especially algebraic ones with initial algebra semantics in mind. The latter approach is, for instance, taken by Ehrig *et al.* in [9]. We follow the syntactic approach, but consider yet another notion of implementation of

* A short version of the paper appeared in the "Proceedings, CAAP'85, Berlin."

0022-0000/87 $3.00

algebraic specifications. There are two conceptual requirements we believe to be important:

1. An implementation should bring us closer to an (executable) program.
2. Implementations should naturally compose on a syntactic level so that correctness criteria are preserved.

Ganziger [11] has introduced the word "program" for certain enrichments used for implementations. Such "programs" are characterized by some semantic conditions but do not resemble programs. We somewhat follow his approach in that we specify enrichments using an abstract programming language over the target data type. The intuition is that sorts are implemented by recursive data structures over standard constructions such as products and sums and that operators are implemented by recursive procedures defined over the operators of the target type and the structure maps of products and sums.

An investigation of this kind presumes a concept of programming language which is defined relative to a basic data type. Several authors (such as [5, 15]) consider programming languages as a specific kind of abstract data types. We believe that such a view is inadequate, at least for our purposes. We are convinced that a programming language intends to denote entities of a certain mathematical universe by using a few orthogonal concepts such as products, sums, function spaces, and recursive domains to build data structures, and by using iteration, recursion, and the operators induced by the structure to define operations (these goals may often be obscured by pragmatic reasons like readability, efficiency, and implementability and by tradition). More specifically uniformity of the structure enhances reasoning in that only a few, but powerful, concepts with a rather clear intuitive meaning are used. In order to take advantage of the structural properties of a programming language we favor the idea that the specification of a programming language should distinguish between the specification of the basic data type(s) and the specification of the (universal) programming language constructs. We have chosen an extended typed λ-calculus defined relative to a basic data type as our concept of a programming language.

To anticipate objections we admit the operational character of the language, and we shall use operational arguments extensively. However, one should notice that the λ-calculus is the canonical initial model with regard to a suitable higher order specification of cartesian closed categories with structure (compare [17]). Hence it is as abstract as quotient term algebras, as used in initial algebra semantics. Moreover, as higher order specifications are generalized algebraic [6], one can prove with little effort that our notion of implementation corresponds to the algebraic one [18]. Nevertheless, we prefer the functional to the equivalent algebraic or combinatory style because we depend on the substantial existing body of knowledge about the reduction properties of functional languages like typed λ-calculi (even if a reduction theory of algebraic categorical languages begins to develop [7]).

Given a specification SPEC over the basic data type we will use Λ SPEC to denote the programming language over SPEC. An implementation

$$\text{SPEC } 0 \Rightarrow \Lambda \text{ SPEC } 1$$

then maps sorts to (recursive) data structures and operators to (recursive) procedures. We adopt the correctness criteria of [8] (RI-correctness and OP-completeness) in that no data are identified by an implementation and that data are implemented by "terminating" programs. We consider two closely related questions:

— Does a correct implementation SPEC $0 \Rightarrow \Lambda$ SPEC 1 extend to a correct implementation Λ SPEC $0 \Rightarrow \Lambda$ SPEC 1?

— Given a second implementation SPEC $1 \Rightarrow \Lambda$ SPEC 2, is the composition SPEC $0 \Rightarrow \Lambda$ SPEC $1 \Rightarrow \Lambda$ SPEC 2 a correct implementation?

According to our experience the behavior of programs should be independent of the implementation of the base type. Unfortunately our choice of implementation fails to satisfy this property, roughly because the operational character of equations is not preserved. In fact this phenomenon is well known to programmers: Assume that the boolean operator "p or q" is evaluated from left to right and that "p or q" is evaluated to "true" if p is evaluated to "true." This strategy is partly expressed by the equation "true or $q = q$." If we implement the or-operator by "p *or* q" satisfying the equations "true *or* true = true" and true *or* false = true," the implementation is correct with regard to the boolean constants "true" and "false." However "true or q" will terminate but not "true *or* q" if q is a non-terminating program. Hence the correctness criterion of preservation of termination is not preserved. The problems are caused by *operational* properties of equations which are not preserved by implementations.

Our main theorem states that correctness extends to programs of ground type if the extended implementation preserves termination (ground types in contrast to higher types, the definition of which includes "function spaces"). The termination condition may be natural from a programmers point of view, and the result could be expected. Surprisingly, at least to the authors, the proof is rather complex involving some machinery of the λ-calculus. This casts some light on the difficulty in finding sufficient but not overly restrictive conditions to ensure correctness of composition for the more general notion of implementation used for abstract data types.

The paper is organized as follows: In Section 1 we introduce and motivate the extended λ-calculus, properties of which are discussed in Section 2. The results provide a basis for the proof of our main theorem about composability of implementations given in Section 3, but are of interest in their own right.

We assume familiarity with [1, 8] from which we borrow some of our notation:

Specifications SPEC $= (S, \Sigma, E)$ consist of a set S of *sorts*, and $S^* \times S -$ $(S -)$ indexed sets Σ and E of *operators* and *equations*. The underlying signature (S, Σ) of SPEC is denoted by Σ SPEC. We use $\sigma : w \to s$ to specify arity and coarity of

an operator. $T_{\text{SPEC}}(X)$ denotes the free SPEC-algebra with generators being the S-sorted set X. We abbreviate $T_{\text{SPEC}} = T_{\text{SPEC}}(\varnothing)$.

$M[x/N]$ is used to state that "N is substituted for free occurences of x in M."

1. Programs over a Specification

1.1. *Motivation*

Enrichments of specifications are used to construct complex specifications out of smaller ones: SPEC $= (S, \Sigma, E)$ is extended to SPEC$'$ = SPEC + (S', Σ', E'), where (S', Σ', E') are additional sorts, operators, and equations. 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 capture this semantically:

1. *Consistency.* No identification of old constants, i.e.,

$$t =_{E+E'} t' \Rightarrow t =_E t' \qquad \text{for} \quad t, t' \in T_\Sigma.$$

2. *Completeness.* No new constants are added to old sorts, i.e.,

$$t \in T_{\Sigma+\Sigma',s}, \qquad s \in S \Rightarrow \exists t' \in T_\Sigma : t =_{E+E'} t'.$$

Consistency and completeness together guarantee the protection of the SPEC-part in the enrichment. To check whether one of the conditions holds is difficult, in general undecidable.

One observes that over and over again the same constructions 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 these standard constructs yields a syntactic notion of enrichment which is transparent but, of course, not exactly equivalent to the semantic ones above. The constructs we are going to consider are both expressable in higher order programming languages and definable by algebraic specifications. One may discuss a definite choice of such constructs: What is typical for programming languages or for algebraic specifications, or what constructs are essential? Therefore the following choice is somewhat arbitrary but, as we hope, reasonable. At least some nice properties to be proved below may justify our choice.

1. *Products.* In higher programming languages products appear as records or classes. In algebraic specifications we write

spec PROD is
SPEC with
sorts **prod-a-b**
ops p: **prod-a-b** → **a**
q: **prod-a-b** → **b**
pair: **a b** → **prod-a-b**

eqns $p(\text{pair}(x, y)) = x$
$q(\text{pair}(x, y)) = y$
$\text{pair}(p(z), q(z)) = z$

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

spec SUM is
SPEC with
sorts **sum-a-b**
ops u: **a** → **sum-a-b**
v: **b** → **sum-a-b**

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

spec SUM *gh* is
SUM with
ops f: **sum-a-b** → **c**
eqns $f(u(x)) = g(x)$
$f(v(y)) = h(y)$

where we assume that **a**, **b**, **c** are sorts in SPEC and that g: **a** → **c**, h: **b** → **c** are operators in SPEC.

3. *Recursive types.* In PASCAL we can describe recursive data structures using recursive schemes of records, variant records, and pointers. In other programming languages a controlled use of pointers may do. As a means for the description of recursive types we introduce domain equations, for example,

$$\text{tree} = 1 + (\text{tree} \times \text{entry} \times \text{tree})$$

$$\text{expr} = \text{term} \times \text{operator} \times \text{term}$$

$$\text{term} = \text{identifier} + \text{expr}.$$

Again entry, operator and identifier are supposed to be given sorts. In an algebraic framework one would, for instance, specify

spec TREE is
ENTRY with
sorts **tree**
ops empty: → **tree**
combine: **tree entry tree** → **tree**
+ "projections to the components of combine (t, e, t') and
definitions by cases"

4. *Recursion.* This is the essential construct which, in combination with the case distinction, allows non-trivial programs to be written, 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 $\sigma(x_0,..., x_{n-1})$ on the left-hand side and an arbitrary right-hand side. In our language we will use a fixpoint operator to denote recursive definitions.

Recursion makes little sense without a conditional. The sum structure offers a conditional via the boolean structure of $1+1$ and the case statement where 1 denotes the "one-point set." A conditional may as well be introduced explicitly using a boolean base sort. The very problem of both (and all other) approaches is to connect the boolean type with the boolean eventually occuring in the base specification. A fair solution would be to understand the booleans as a shared subtype but a theory of sharing subtypes is hardly developed and out of the scope of this paper. For reasons of tradition we prefer an explicit conditional.

1.2. *The Type Structure*

For the set S of a given specification (S, Σ, E) we construct products, sums, and recursive types as congruence classes of type terms over S.

Let $\mathrm{TT}\, n(S)$ denote type terms with type variables $\alpha_0,..., \alpha_{n-1}$ constructed as the smallest set such that

(a) $S \subseteq \mathrm{TT}\, o(S)$, $1 \in \mathrm{TT}\, o(S)$

(b) $\mathrm{TT}\, m(S) \subseteq \mathrm{TT}\, n(S)$ for $m \leqq n$

(c) $\alpha_i \in \mathrm{TT}\, i(S)$ for $i \in \mathbb{N}_0$

(d) $\tau, \tau' \in \mathrm{TT}\, n(S) \Rightarrow \tau + \tau', \tau \times \tau' \in \mathrm{TT}\, n(S)$.

Now take the recursive-type scheme

$$\begin{aligned} \alpha_0 &= \tau_0(\alpha_0,..., \alpha_{n-1}) \\ &\vdots \\ \alpha_{n-1} &= \tau_{n-1}(\alpha_0,..., \alpha_{n-1}) \end{aligned}$$

with n variables and n equations. We introduce names for the n solutions of this scheme by $D_n^i(\tau_0,..., \tau_{n-1})$, $i \in \mathbf{n} := \{0,..., n-1\}$. We get arbitrarily nested schemes if we regard these solutions as new constants. Thus the definition of $\mathrm{TT}\, n(S)$ is completed by the line

(e) $\tau_0,..., \tau_{n-1} \in \mathrm{TT}\, n(S) \Rightarrow D_n^i(\tau_0,..., \tau_{n-1}) \in \mathrm{TT}\, o(S)$.

EXAMPLE. The definition of expr and term is translated to

$$t_0(\alpha_0, \alpha_1) \equiv \alpha_1 \times \text{operator} \times \alpha_1$$

$$t_1(\alpha_0, \alpha_1) \equiv \text{identifier} + \alpha_0$$

expr is represented by $D_2^0(t_0, t_1)$, term by $D_2^1(t_0, t_1)$.

The property to be a solution of the recursive-type scheme is satisfied if we identify

$$D_2^0(t_0, t_1) \text{ and } D_2^1(t_0, t_1) \times \text{operator} \times D_2^1(t_0, t_1)$$

(resp. $D_2^1(t_0, t_1)$ and identifier $+ D_2^0(t_0, t_1)$).
To obtain $D_n^i(\tau_0,..., \tau_{n-1})$ as solution of the respective recursive scheme the type terms are to be factorized by the least equivalence relation $\sim$ containing

$$D_n^i(\tau_0,..., \tau_{n-1}) \sim \tau_i[\alpha_0/D_n^0(\tau_0,..., \tau_{n-1}),..., \alpha_{n-1}/D_1^{n-1}(\tau_0,..., \tau_{n-1})] \quad \text{for } i \in \mathbf{n}$$

$$\tau_i \sim \tau_i', i \in \mathbf{2} \Rightarrow \tau_0 + \tau_1 \sim \tau_0' + \tau_1'$$

$$\tau_i \sim \tau_i', i \in \mathbf{2} \Rightarrow \tau_0 \times \tau_1 \sim \tau_0' \times \tau_1'$$

$$\tau_i \sim \tau_i', i \in \mathbf{n} \Rightarrow D_n^j(\tau_0,..., \tau_{n-1}) \sim D_n^j(\tau_0',..., \tau_{n-1}') \quad \text{for } j \in \mathbf{n}.$$

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

DEFINITION. The set of *ground* or *data types* over S is given by $\mathrm{DType}(S) := \mathrm{TT}\, o(S)/\sim$. The set of (*higher*) *types* is defined as the smallest set $\mathrm{Type}(S)$ with

1. $\mathrm{DType}(S) \subseteq \mathrm{Type}(S)$
2. $\tau, \tau' \in \mathrm{Type}(S)$, $\tau \notin \mathrm{DType}(S)$ or $\tau' \notin \mathrm{DType}(S) \Rightarrow \tau + \tau'$, $\tau \times \tau' \in \mathrm{Type}(S)$
3. $\tau, \tau' \in \mathrm{Type}(S) \Rightarrow \tau \to \tau' \in \mathrm{Type}(S)$.

We refer to types of the form $\tau \to \tau'$ as *functional types.*

Remark. Our results do not hold for recursive types involving function spaces as we depend on normalization properties.

1.3. *The Programming Language* Λ

As our language is an extension of typed λ-calculus the standard conventions apply. The terms are typed by the set $\mathrm{Type}(S)$ with $\mathrm{SIG} = (S, \Sigma)$ being a signature. X is an enumerable set of variable names, typically x, y, z,.... Variables are of the form $x : \tau$ with x being a variable name and τ being a type. For convenience we often omit the type index if the typing can be recovered from the context.

DEFINITION. Let $\mathrm{SIG} = (S, \Sigma)$ be a signature. The language $\Lambda\,\mathrm{SIG}$ (for short, Λ) is the smallest $\mathrm{Type}(S)$-sorted set such that

(a) $x : \tau \in \Lambda_\tau$ if $x \in X$ and $\tau \in \mathrm{Type}(S)$ — *variables*

(b) $\sigma \in \Lambda_s$ if $\sigma : \to s \in \Sigma$; — *base operators*
$M_i \in \Lambda_{s_i}$, $i \in \mathbf{n} \Rightarrow \sigma(M_0,..., M_{n-1}) \in \Lambda_s$ if $\sigma : s_0 \cdots s_{n-1} \to s \in \Sigma$

(c) $M \in \Lambda_{\tau \to \tau'}$, $N \in \Lambda_\tau \Rightarrow (MN) \in \Lambda_\tau$; — *application*
$M \in \Lambda_{\tau'} \Rightarrow (\lambda x : \tau \cdot M) \in \Lambda_{\tau \to \tau'}$ — *abstraction*

(d)	$\diamond \in \Lambda_1$	*null tuple*
	$p_{\tau,\tau'} \in \Lambda_{\tau\times\tau'\to\tau}$, $q_{\tau,\tau'} \in \Lambda_{\tau\times\tau'\to\tau'}$	*projections*
	$M \in \Lambda_\tau$, $N \in \Lambda_{\tau'} \Rightarrow \langle M, N\rangle \in \Lambda_{\tau\times\tau'}$	*pairing*
(e)	$M \in \Lambda_\tau \Rightarrow u_{\tau,\tau'} * M \in \Lambda_{\tau+\tau'}$	*injections*
	$N \in \Lambda_{\tau'} \Rightarrow v_{\tau,\tau'} * N \in \Lambda_{\tau+\tau'}$	
	$C \in \Lambda_{\tau+\tau'}$, $M \in \Lambda_{\tau\to\tau''}$, $N \in \Lambda_{\tau'\to\tau''}$	
	$\Rightarrow$ case C, M, N esac $\in \Lambda_{\tau''}$	*cases*
(f)	$B \in \Lambda_{\text{bool}}$, $M, N \in \Lambda_\tau \Rightarrow$ if B then M else N fi $\in \Lambda_\tau$	*conditional*
	$M \in \Lambda_{\tau\to\tau} \Rightarrow Y(M) \in \Lambda_\tau$	*fixpoint operator.*

For convenience we often omit the subscripts of projections and injections. Free and bound variables are defined as usual. We use $\mathrm{FV}(M)$ to denote the set of variables occuring free in M. Substitution is also defined in the usual way preserving the structure. We use $M[x/N]$ as notation for "N is substituted for free occurences of x in M." As in [2] we assume that substitution does not affect the binding structure, i.e., we consider terms modulo α-conversion (justified by an obvious extension of Barendregt's argument). The set of closed terms of type τ is denoted by $\Lambda\,\mathrm{SIG}^0_\tau$.

We define *reduction* on SIG-terms by

(β) $(\lambda x : \tau \cdot M)N \to M[x : \tau/N]$

(η) $\lambda x : \tau \cdot (Mx) \to M$ if $x : \tau \notin \mathrm{FV}(M)$

(τ) $M \to \diamond$ for $M \in \Lambda\,\mathrm{SIG}_1$

(π) $p\langle M, N\rangle \to M$, $q\langle M, N\rangle \to N$

(δ) $\langle pL, qL\rangle \to L$

(ρ) case $u * L$, M, N esac $\to ML$; case $v * L$, M, N esac $\to NL$

(ζ) if true then M else N fi $\to M$; if false then M else N fi $\to N$

(Y) $Y(M) \to M(Y(M))$.

Reduction is compatible with the structure in that computations may proceed in subterms (captured by a lengthy list of obvious axioms and rules).

We use $M \twoheadrightarrow N$ as notation for the reflexive and transitive closure of reduction.

We refer to *conversion* if, additionally, symmetry of reduction is assumed. We use $M = N$ as notation for conversion.

Let $\mathrm{SPEC} = (S, \Sigma, E)$ be a specification of a *rewrite system* in that equations in E are understood as rewrite rules from left to right. We assume the following fairly standard restrictions to hold:

(E1) $\mathrm{FV}(t') \subseteq \mathrm{FV}(t)$ for $(t, t') \in E$

(E2) $\bar{E}$ is *Church–Rosser*

(E3) E is *left-linear*, i.e., variables occur only once in t for $(t, t') \in E$

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

(E5) $\bar{E}$ is strongly normalizing,

where $\bar{E}$ denotes the smallest relation such that

(i) $E \subseteq \bar{E}$

(ii) $(t,t') \in E,\ (t_i,t_i') \in \bar{E}\ (i \in \mathbf{n}) \Rightarrow (t[x_0/t_0,\ldots,x_{n-1}/t_{n-1}],\ t'[x_0/t_0',\ldots,\ t_{n-1}']) \in \bar{E}$.

The rewrite rules are added to the SIG-calculus by

(E) $t[\bar{x}/\bar{M}] \to t'[\bar{x}/\bar{M}]$ if $(t, t') \in E$;

$\bar{x}$ (resp. $\bar{M}$) denotes a tuple of variables (resp. terms) the ith component of which are x_i (resp. M_i). Substitution is simultaneous.

We use Λ SPEC to denote the language based on the specification SPEC.

Throughout the paper we use the

GENERAL ASSUMPTION. Specifications SPEC only have *non-trivial* sorts, i.e., $T_{\Sigma\,\mathrm{SPEC},s} \neq \varnothing$ for all sorts s, and they are BOOL-*constrained*, i.e., $T_{\mathrm{SPEC,bool}} = \{[\mathrm{true}], [\mathrm{false}]\}$.

Notation. We index reductions to state which axioms are allowed in a computation. A "—" indicates that rules are not used: For instance, $M \twoheadrightarrow_{\beta\eta} N$ stand for "$M \twoheadrightarrow N$" but only the β- and η-rule are used for reduction, $M \twoheadrightarrow_{-\eta} N$ states that all rules except for the η-rule are used. A sequence $(M_0,\ldots, M_n)$ such that $M_i \to M_{i+1}$, $i \in \mathbf{n}$, is called a *computation sequence.*

To recall the definitions we say that a binary relation $X < Y$ is *strongly Church–Rosser* if $X < Y$ and $X < Z$ implies existence of a W such that $Y < W$ and $Z < W$. The relation is *Church–Rosser* if its reflexive and transitive closure is strongly Church–Rosser. If a relation is strongly Church–Rosser it is Church–Rosser [2].

A computation sequence $X_0 < X_1 < \cdots < X_n$ is *finitary* if X_n is in *normal form*, i.e., $X_n \not< X$ for all X. We say that $<$ *strongly normalizes* if no infinitary computation $X_0 < X_1 < \cdots < X_n < \cdots$ exists.

Remark. Our extension of the standard typed λ-calculus is a higher type specification in the sense of [17]. There (structured) parameter lists

$$\lambda \langle x_0 : \tau_0,\ldots, x_{n-1} : \tau_{n-1} \rangle \cdot M$$

are used. Then projections can be replaced by terms $\lambda \langle x : \tau, y : \tau' \rangle \cdot x$ and $\lambda \langle x : \tau, y : \tau' \rangle \cdot y$. One can prove the equivalence of the calculi. For better readability we will use parameter lists as well as manifold products and sums in our examples. Nevertheless the calculus introduced is more convenient for proofs.

EXAMPLES. We define some functions on non-empty lists and trees over a data-type entry. The recursive types are represented by type equations:

$$\mathrm{list} = \mathrm{entry} + (\mathrm{entry} \times \mathrm{list})$$

$$\mathrm{tree} = \mathrm{entry} + (\mathrm{entry} \times \mathrm{tree}) + (\mathrm{tree} \times \mathrm{entry}) + (\mathrm{tree} \times \mathrm{entry} \times \mathrm{tree})$$

$$\mathrm{latt} \equiv \lambda \langle e : \mathrm{entry}, l : \mathrm{list} \rangle \cdot v * \langle e, l \rangle.$$

To attach an entry to the right-hand side of a list we write a recursive program

$$\begin{aligned}\text{ratt} \equiv Y(&\lambda f : \text{entry} \times \text{list} \to \text{list}.\\ &\lambda \langle e : \text{entry}, l : \text{list}\rangle \cdot \text{case } l,\quad \lambda ee : \text{entry} \cdot v * \langle ee, u * e\rangle,\\ &\qquad\qquad \lambda \langle ee : \text{entry}, ll : \text{list}\rangle \cdot v * \langle ee, f\ \langle e, ll\rangle\rangle\\ &\quad \text{esac})\end{aligned}$$

Concatenation of lists is given by

$$\begin{aligned}\text{conc} \equiv Y(&\lambda f : \text{list} \times \text{list} \to \text{list}\\ &\lambda \langle l1 : \text{list}, l2 : \text{list}\rangle \cdot \text{case } l2,\quad \lambda e : \text{entry} \cdot \text{ratt}(e, l1),\\ &\qquad\qquad \lambda \langle e : \text{entry}, l : \text{list}\rangle \cdot f\ \langle \text{ratt}\langle e, l1\rangle, l\rangle\\ &\quad \text{esac})\end{aligned}$$

and the inorder representation of a tree is given by

$$\begin{aligned}\text{inorder} \equiv Y(&\lambda f : \text{tree} \to \text{list}\\ &\lambda t : \text{tree} \cdot \text{case } t,\quad \lambda e : \text{entry} \cdot u * e,\\ &\qquad \lambda \langle e : \text{entry}, t2 : \text{tree}\rangle \cdot v * \langle e, f\ t1\rangle,\\ &\qquad \lambda \langle t1 : \text{tree}, e : \text{entry}\rangle \cdot \text{ratt}\ \langle e, f\ t1\rangle,\\ &\qquad \lambda \langle t1 : \text{tree}, e : \text{entry}, t2 : \text{tree}\rangle.\\ &\qquad\qquad \text{conc}\ \langle f\ t1, v * \langle e, f\ t2\rangle\rangle\\ &\quad \text{esac})\end{aligned}$$

Simultaneous recursion is possible. For expr and term we define operations which count the occurences of operators and identifiers

$$\begin{aligned}\text{count} \equiv Y(&\lambda \langle f : \text{expr} \to \text{nat}, g : \text{term} \to \text{nat}\rangle.\\ &\langle \lambda \langle \text{t1} : \text{term}, \text{op} : \text{operator}, \text{t2} : \text{term}\rangle \cdot (g\ t1) + (g\ t2) + 1,\\ &\lambda x : \text{term} \cdot \text{case } x, \lambda i : \text{identifier} \cdot 1, f\ \text{esac}\end{aligned}$$

The following is an example for reduction:
For the constant of type tree

$$T \equiv u2 * \langle e1, u4 * \langle u1 * e3, e2, u1 * e4\rangle\rangle$$

where $e1$, $e2$, $e3$, and $e4$ are given constants of type entry, and $ui *\_$ is used for the ith injection in the sum.

We reduce the term

$$\text{inorder}(T) \to (\lambda f : \textbf{tree} \to \textbf{list} \cdot \lambda b : \textbf{tree} \cdot \text{case} \cdots \text{esac}(b))\ \text{inorder}\ T$$

$$\to \lambda b : \textbf{tree} \cdot \text{case}\ b,\ \lambda e : \textbf{entry} \cdot u * e,$$
$$\lambda e : \textbf{entry}, b : \textbf{tree} \cdot v * \langle e, \text{inorder}\ b \rangle,$$
$$\lambda b : \textbf{tree}, e : \textbf{entry} \cdot \text{ratt}\ \langle \text{inorder}\ b, e \rangle,$$
$$\lambda b1 : \textbf{tree}, e : \textbf{entry}, \text{b2} : \textbf{tree} \cdot \text{conc}\ \langle \text{inorder}\ b1, v * \langle e, \text{inorder b2} \rangle \rangle$$
$$\text{esac}\ T$$
$$\to \text{case}\ u2 * \langle e1, u4 * \langle u1 * e3, e2, u1 * e4 \rangle \rangle, \ldots\ \text{esac}$$
$$\to\!\to v * \langle e1, \text{inorder}\ u4 * \langle u1 * e3, e2, u1 * e4 \rangle \rangle$$
$$\to\!\to v * \langle e1, \text{conc}\ \langle \text{inorder}\ u1 * e3, v * \langle e2, \text{inorder}\ u1 * e4 \rangle \rangle \rangle$$
$$\to\!\to v * \langle e1, \text{conc}\ \langle u * e3, v * \langle e2, u * e4 \rangle \rangle \rangle$$
$$\to\!\to v * \langle e1, v * \langle e3, v * \langle e2, u * e4 \rangle \rangle \rangle$$

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 functional language. We prefer the more clumsy λ-notation for proof-theoretic reasons.

We complete the section with a few remarks:

— One should observe that the extension of a specification SPEC to the language Λ SPEC is not complete as the fixpoint operator adds data to the base sorts. In general, the extension is not even consistent as terms of the base specification may be identified. If we, for instance, have a specification of booleans using the standard axioms of Boolean algebras and the fixpoint property of Y (not), we can compute

$$\text{true} = (Y\ \text{not})\ \text{or not}\ (Y\ \text{not}) = (Y\ \text{not})\ \text{or}\ (Y\ \text{not}) = Y\ \text{not}$$
$$\text{false} = (Y\ \text{not})\ \text{and not}\ (Y\ \text{not}) = (Y\ \text{not})\ \text{and}\ (Y\ \text{not}) = Y\ \text{not}.$$

Hence true = false in Λ BOOL but true $\neq$ false in BOOL.

— The essential use of the rewrite rules of the specification is as "stop-equations." Besides β-reduction E-reductions are able to eliminate Y's and thereby stop a recursive calculation.

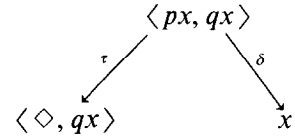
It is not realistic to require (E1)–(E4) to hold for all specifications. But we can take the following point of view: The use of stop-equations is a kind of error recovery mechanism. Like other authors we may distinguish a certain subset of E to be chosen for this purpose. Only these special equations may be used in the calculus in that arbitrary terms (especially those containing Y's) are substituted for variables while others are restricted in that only terms of $T_{\text{SPEC}}(X)$ are substituted. Formally then a subsort technique as in [16] may be used.

The assumption of strong normalization (E5) is not essential for our main result (compare [19, 23], where different proof techniques are used). But viewing equations as an operational device, it seems to be a fair assumption that infinitary computations are only caused by recursive programs. At least, this is a common assumption for programming languages.

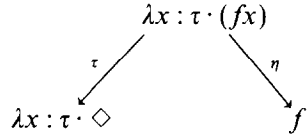
— The "extensionality" axioms (η), (τ), and (δ) are, in general, not considered in reduction systems for typed λ-calculi. Whatever the reasons may be, certainly proofs of the Church–Rosser property are more complicated in the presence of extensionality axioms. Nevertheless we believe that exclusion of extensionality axioms from the very beginning is unnatural as then terms are not convertible, which intuitively should be so, e.g., $\lambda x : s \cdot \sigma x$ and σ, $Y(\lambda x : 1 \times 1 \cdot x)$ and $\langle \diamond, \diamond \rangle$ (moreover, the equivalence to cartesian closure would no longer hold).

1.4. *Properties of the Calculus*

Unfortunately the calculus is not even *weakly Church–Rosser* (i.e., if $M \to P$ and $M \to Q$, then there exists a R such that $P \twoheadrightarrow R$ and $Q \twoheadrightarrow R$). There are several conflicting reductions because of the extensionality axiom for the "one-point" type 1:



where x is a variable of type $1 \times \tau$ (one may consider $Y(\lambda x : 1 \times \tau \cdot x)$ as well);



where f is a variable of type $\tau \to 1$. If we replace f by $Y(\lambda f : \tau \to 1 \cdot f)$ the example shows that the normal order or leftmost-outermost reduction is not correct in that a normal form is not necessarily computed if a normal form exists.

The conflicts can be resolved by adding suitable reductions which are—somewhat unexpectedly—quite complex.

We define 1-*types* and terms $@\tau$ of 1-type τ inductively by

(i) 1 is a 1-type, $@1 = \diamond$

(ii) $\tau \to \tau'$ is a 1-type if τ' is a 1-type, $@\tau \to \tau' = \lambda x : \tau \cdot @\tau$

(iii) $\tau \times \tau'$ is a 1-type if τ and τ' are 1-types, $@\tau \times \tau' = \langle @\tau, @\tau' \rangle$.

We then add the reduction rules

(τ) $M \to @\tau$ if M is of 1-type τ.

(δ) $\langle @\tau, qM\rangle \to M$, $\langle pM, @\tau'\rangle \to M$ if M is of type $\tau \times \tau'$ with either τ or τ' being a 1-type.

Remark. The rules do not cause new identifications of terms with regard to conversion.

We state some results for which the proofs are given in Section 2.

PROPOSITION. *$M \to {}_{Y}N$ is strongly normalizing.*

PROPOSITION. *$M \to N$ is Church–Rosser.*

COROLLARY. *$M = N$ iff $M \twoheadrightarrow L$ and $N \twoheadrightarrow L$ for some L.*

We say that SPEC is *consistent* if $t = t'$ (in SPEC) implies that $t =_E t'$ for $t, t' \in T_{\text{SPEC}}$.

PROPOSITION. *SPEC is consistent.*

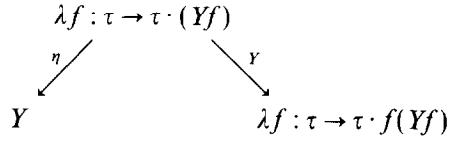
Proof. Use the Church–Rosser property and

$$t \to_E M \qquad \text{if} \quad t \to M \text{ for } t \in T_{\text{SPEC}}(X).$$

Remark. The reduction properties depend on the specific way the additional operators are introduced. If, for instance, the fixpoint operator is introduced as a constant, e.g.,

$$Y \in \Lambda_{(\tau \to \tau) \to \tau}$$

additional conflicts are caused



which can be resolved by the additional reduction rule

$$Y \to \lambda f : \tau \to \tau \cdot f(Yf) \qquad \text{if} \quad Y \text{ is of type } (\tau \to \tau) \to \tau.$$

2. Properties of Programs

We discuss several properties of reduction and conversion which will be used in the proofs on composition of implementations. We believe that some of the results are of interest in their own right. We first check

2.1. *Strong Normalization*

Strong normalization of a variant of the calculus is introduced for purposes which are motivated by the arguments in the next section.

We extend SPEC by a constant $\perp$ for each type. For base types $\perp_s \equiv c_s$, where $c_s \in T_{\text{SPEC}}$ (such a constant exists due to our general assumption).

Additional reduction rules are

($\perp$) $\perp M \to \perp$, $p\perp \to \perp$, $q\perp \to \perp$, case $\perp$, M, N esac $\to \perp$.

2.1. PROPOSITION. *$M \twoheadrightarrow_{\_\gamma} N$ is strongly normalizing.*

We use an extension of Tait's proof technique [22] following the proofs given in [21, 10]. The key idea is to replace "basic types" by "non-functional types" to which we refer as *structured types.* A similar proof for a standard typed λ-calculus (i.e., without recursive data structure and without the fixpoint operator, conditional and rewriting on the base specification) seems to be given in [20, 12] (we have no access to these papers; we have found a reference in [7]).

The proof is split into a definition and several lemmas. We define the notion of stability for terms and prove that each stable term is strongly normalizable (SN) and that each term is stable.

DEFINITION OF STABILITY. (1) If M is of type τ with τ being a structured type, then M is stable provided that M is SN and that

(a) M' and M'' are stable if M reduces to $\langle M', M'' \rangle$, or

(b) M' is stable if M reduces to u^*M' or to v^*M'.

(2) A term M of type $\tau \to \tau'$ is stable if MN is stable of type τ' for all stable N of type τ.

Remark. To avoid additional case distinctions we assume without restriction of generality that the terms we consider in this section are not of 1-type. Terms of 1-type are trivially stable and SN.

2.2. LEMMA. (i) *Every stable term M of type τ is SN.*

(ii) *A term $xM_1 \cdots M_n$ of type τ is stable whenever the M_i's are SN (x being a variable).*

Proof. By induction on τ (the proof is essentially given in the above references but is included for sake of completeness).

(a) τ is a structured type:

- Each stable term of type τ is SN by definition.
- $xM_1 \cdots M_n$ is SN as the computation can only proceed in the M_i's.

Clearly,

- $xM_1 \cdots M_n$ cannot reduce to a term of the form $\langle M', M'' \rangle$, u^*M', or v^*M'.

(b) Let M be a stable term of type $\tau \to \tau'$, and let x be a variable of type τ. By inductive assumption ($n = 0$) x is stable, so Mx is stable (by (2)), and by inductive assumption Mx and thus M is SN.

Let $xM_1 \cdots M_n$ be a term of type $\tau \to \tau'$ with all the M_i SN, and let N be a stable term of type τ. It follows from the inductive assumption that N is SN as well. Then $xM_1 \cdots M_n N$ is SN as the computations can only occur in the components. Again applying the induction hypothesis $xM_1 \cdots M_n N$ is stable, and so is $xM_1 \cdots M_n$ as the term cannot be reduced to a term of the form $\langle M', M'' \rangle$, u^*M', or v^*M'.

In the proofs to come we refer to the

2.3. FACT. *If M is stable and reduces to N, then N is stable.*

Proof. Obvious for terms of structured type. If M is of function type consider suitable stable terms $M_1, \ldots, M_n$ such that $MM_1 \cdots M_n$ is of structured type.

2.4. LEMMA. (i) *The constants $\diamond$ and $\perp$ are stable*

(ii) *$Y(M)$ is stable if M is stable.*

Proof. (i) $\diamond$ is in normal form and not of the form $\langle M', M'' \rangle$, u^*M', or v^*M': If $\perp$ is of structured type the same argument applies. Otherwise one uses that $\perp M$ reduces to $\perp$.

(ii) Let $M_1, \ldots, M_n$ be stable terms of suitable type such that $Y(M)M_1 \cdots M_n$ is of structured type.

As fixpoint reduction is excluded the computations must proceed within the terms $M, M_1, \ldots, M_n$. By 2.2 these terms are SN and hence so is $Y(M)M_1 \cdots M_n$. Since the term cannot be reduced to a term of the form $\langle M', M'' \rangle$, u^*M', or v^*M', we conclude that $Y(M)M_1 \cdots M_n$ is stable. By definition of stability then $Y(M)$ is stable.

The arguments are typical in that they always apply if the term of interest does not interact with its context. Hence, in proofs to come, we focus our interest on such interactions.

2.5. LEMMA. *$\sigma(M_0, \ldots, M_{n-1})$ is stable if the terms $M_0, \ldots, M_{n-1}$ are stable.*

Proof. For notational convenience we use $\bar{M}$ for a tuple of (argument) terms and $\bar{x}$ for a tuple of variables.
If the computation only proceeds within $\bar{M}$, then $\sigma(\bar{M})$ is SN if $\bar{M}$ is so. Due to typing SN implies stability for base sorts. Thus an infinite computation must start with

$$M \equiv \sigma(\bar{x}_0)[\bar{x}_0/\bar{M}_0] \xrightarrow[-YE]{}\!\!\!\!\!\rightarrow \sigma(\bar{x}_0)[\bar{x}_0/\langle \bar{t}_0 \rangle][\bar{x}_1/\bar{M}_1] \xrightarrow[E]{} t_1[\bar{x}_1/\bar{M}_1] \xrightarrow[-YE]{}\!\!\!\!\!\rightarrow \cdots,$$

where $\bar{M}_0 \twoheadrightarrow_{-YE} \bar{t}_0[\bar{x}_1/\bar{M}_1]$ and $\bar{t}_0$ is a list of terms of $T_{\mathrm{SPEC}}(\{\bar{x}_1\})$, $\bar{x}_1$ being the only variables occuring in $\bar{t}_0$ and t_1. Because condition (E1) for the rewriting system stability (= SN) of $\bar{M}_0$ implies that of $\bar{M}_1$. We cannot iterate the argument as the ith component of $\bar{M}_1$ may occur more than once in $t_1[\bar{x}_1/\bar{M}_1]$, and as further computations may differently proceed in the different components. To cope with the problem we index all but one occurence of the components of $\bar{x}_1$ in t_1, with the resulting term being t_1', and obtain a new list of (indexed) variables, each of them occuring only once in t_1'. Then

$$M \equiv \sigma(\bar{x}_0)[\bar{x}_0/\bar{M}_0] \xrightarrow[-YE]{}\!\!\!\!\twoheadrightarrow \sigma(\bar{x}_0)[\bar{x}_0/\bar{t}_0][\bar{x}_1'/\bar{M}_1'] \xrightarrow[E]{} t_1'[\bar{x}_1'/\bar{M}_1'] \equiv t_1[\bar{x}_1/\bar{M}_1],$$

where $\bar{M}_1$ is obtained by substituting the ith component of $\bar{M}_1$ for the (indexed) ith component of $\bar{x}_1$ in $\bar{x}_1'$ ($\bar{x}_1$ is a sublist of x_1').

Then the infinitary computation must continue as follows

$$t_1'[\bar{x}_1'/\bar{M}_1'] \xrightarrow[-YE]{}\!\!\!\!\twoheadrightarrow t_1'[\bar{x}_1'/\bar{t}_2][\bar{x}_2/\bar{M}_2] \xrightarrow[E]{} t_2[\bar{x}_2/\bar{M}_2] \xrightarrow[-YE]{}\!\!\!\!\twoheadrightarrow \cdots.$$

We apply the same indexing procedure as above, and iterate the argument.

The computation defines terms of the form

$$t_i'[\bar{x}_i'/\bar{t}_i'] \cdots [\bar{x}_n'/\bar{t}_n']$$

($t_0 \equiv \sigma(\bar{x}_0)$) such that

$$t_i'[\bar{x}_i'/\bar{t}_i'] \cdots [\bar{x}_n'/\bar{t}_n'] \xrightarrow[E]{} t_{i+1}'[\bar{x}_{i+1}'/\bar{t}_{i+1}'] \cdots [\bar{x}_n'/\bar{t}_n'].$$

The terms $\bar{t}_n'$ are generated from $\bar{M}_0$,

$$\bar{M}_0 \xrightarrow[-YE]{}\!\!\!\!\twoheadrightarrow \bar{t}_0[\bar{x}_1'/\bar{M}_1'] \xrightarrow[-YE]{}\!\!\!\!\twoheadrightarrow \bar{t}_0[\bar{x}_1'/\bar{t}_1'][\bar{x}_2'/\bar{M}_2'] \xrightarrow[-YE]{}\!\!\!\!\twoheadrightarrow \cdots,$$

hence the creation of terms is bounded, say by n, as $\bar{M}_0$ is SN. We obtain a sequence

$$\begin{aligned}
&\sigma(\bar{x}_0)[\bar{x}_0/\bar{t}_0][\bar{x}_1'/\bar{t}_1'] \cdots [\bar{x}_n'/\bar{t}_n'] \\
&\quad \xrightarrow[E]{} t_1'[\bar{x}_1'/\bar{t}_1'] \cdots [\bar{x}_n'/\bar{t}_n'] \xrightarrow[E]{} t_2'[\bar{x}_2'/\bar{t}_2'] \cdots [\bar{x}_n'/\bar{t}_n'] \\
&\quad \xrightarrow[E]{}\!\!\!\!\twoheadrightarrow t_n'[\bar{x}_n'/\bar{t}_n'] \xrightarrow[E]{} t_{n+1}' \xrightarrow[E]{} t_{n+2}' \xrightarrow[E]{} \cdots,
\end{aligned}$$

in contradiction to the strong normalization of rewriting.

2.6. LEMMA. *The projections $p : \tau \times \tau' \to \tau$, $q : \tau \times \tau' \to \tau'$ are stable. The terms case L, M, N esac, $\langle M, N \rangle$, u^*M, v^*N and if B then M else N if are stable if B, L, M, and N are stable.*

Proof. (a) We check that pM is stable for stable arguments M. An interaction with the environment takes place if M reduces to $\perp$ or to $\langle M', M''\rangle$. In the first case a reduction of $pMM_1 \cdots M_n$ with the M_is being stable such that $pMM_1 \cdots M_n$ is of structured type yields $\perp$ (by a straightforward argument). Otherwise, if an infinite reduction is caused, it must be of the form

$$pMM_1 \cdots M_n \xrightarrow[-Y]{} p\langle M', M''\rangle M_1' \cdots M_n' \xrightarrow[-Y]{} M'M_1' \cdots M_n'.$$

Stability of M implies that of M' and M'' by definition. We conclude that $M'M_1' \cdots M_n'$ is stable (using 2.3), hence SN (in contradiction to the assumption), and if $M'M_1' \cdots M_n'$ reduces to a term of the form $\langle L', L''\rangle$, u^*L', or v^*L' then the terms L' and L'' are stable.

(b) We consider case L, M, N esac, and assume that L reduces to u^*L' or to $\perp$. The argument is basically the same as in (a), using that L' is stable by definition of stability.

(c) One problematic case is that $\langle M, N\rangle$ reduces to $\langle pL, qL\rangle$ and to L. But stability of M implies that pL is stable, hence SN. Thus L must be SN. If L reduces to $\langle L', L''\rangle$ then M reduces to L' which proves stability of L'. L cannot reduce to u^*L' or to v^*L' because of typing.

The other interesting case is that $\langle M, N\rangle$ reduces to $\langle \diamond, qL\rangle \to L$. L is SN as qL is stable. If L reduces to $\langle L', L''\rangle$ L'' is stable as qL is so. Moreover, L' reduces to $\diamond$.

(d) The other terms are handled in what is now the obvious way.

2.7. LEMMA. *If $M[x/N]$ is stable, then $(\lambda x : \tau \cdot M)N$ is stable, provided that N is stable if x does not occur free in M.*

Proof. We sketch the proof given in [21]. Again we provide enough M_is to obtain a term $M[x/N]\, M_1 \cdots M_n$ of structured type. Clearly, $(\lambda x : \tau \cdot M)\, NM_1 \cdots M_n$ has the same properties as $M[x/N]\, M_1 \cdots M_n$ which is SN except that an infinite reduction may be caused by application of the β-rule. Either the infinite reduction is caused by reduction of M, but then $M[x/N]$ has an infinite reduction in contradiction to $M[x/N]\, M_1 \cdots M_n$ being SN, or $\lambda x : \tau \cdot M$ reduces to $\lambda x : \tau \cdot M'x$ and to M' (with x not free in M') causing the infinite reduction. But then $M[x/N]$ reduces to $M'N$ starting an infinite reduction.

The proof of 2.1 is completed by

2.8. LEMMA. *Every instance M' of a term M is stable, where M' is an* instance *of M if M' is obtained by correct simultaneous substitution of stable expressions for free variables in M.*

Proof. By induction on the structure of terms. If $M = x$ is a free variable the statement is immediate.

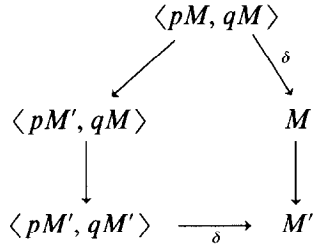
If $M = y$ is a non-free occurence of a variable or if M is a constant, stability follows from the previous lemmata.

For all terms, except for $M = \lambda x : \tau \cdot N$, the obvious inductive argument is to be applied.

If $M = \lambda x : \tau \cdot N$, let N' be an instance of N. Then $N'[x/L]$ is also an instance of N, and stable by the induction hypothesis. Since $(\lambda x : \tau \cdot N')L$ reduces to $N'[x/L]$ stability of M follows from 2.7.

2.2. *The Church–Rosser Property*

A proof of the Church–Rosser property cannot be obtained by the standard technique of "simultaneous computation of independent redexes" [2] as the δ-rule may need some adjustment steps



Instead we make a profit of the strong normalization property which, combined with the weak Chruch–Rosser property yields the Church–Rosser property.

We now check that the calculus is weakly Church–Rosser.

2.9. LEMMA. *$M[x/N] \twoheadrightarrow M'[x/N']$ if $M \to M'$ and $N \to N'$.*

Proof. We use a case distinction along the lines of Lemma 3.2.4 in [2].

(a) $(\lambda y : \tau \cdot P)\, Q \to_\beta P[y/Q]$:

$$(\lambda y : \tau \cdot P)\, Q[x/N] \equiv (\lambda y : \tau \cdot P[x/N])\, Q[x/N]$$
$$\twoheadrightarrow (\lambda y : \tau \cdot P[x/N'])\, Q[x/N'] \qquad \text{by induction hypothesis}$$
$$\xrightarrow[\beta]{} P[x/N'][y/Q[x/N']]$$
$$\equiv P[y/Q][x/N'].$$

The last identity is that of the substitution lemma of [2, 2.1.16], which canonically extends to cope with the additional operators.

(b) $\lambda y : \tau \cdot (Py) \to_\eta P$:

$$\lambda y : \tau \cdot (Py)[x/N] \equiv \lambda y : \tau \cdot (P[x/N]\, y)$$
$$\twoheadrightarrow \lambda y : \tau \cdot (P[x/N']\, y) \qquad \text{by induction hypothesis}$$
$$\xrightarrow[\eta]{} P[x/N']$$

(c) $p\langle P, Q\rangle \to_{\pi} P$:

$$p\langle P, Q\rangle [x/N] \equiv p\langle P[x/N], Q[x/N]\rangle$$
$$\twoheadrightarrow p\langle P[x/N'], Q[x/N']\rangle \qquad \text{by induction hypothesis}$$
$$\xrightarrow[\pi]{} P[x/N'].$$

The same scheme of argumentation applies to the other reduction rules except for

(d) $t[\bar{x}/\bar{P}] \to_{E} t'[\bar{x}/\bar{P}]$:

As $\bar{x}$ is the list of variables occuring in t and t' we have

$$t[\bar{x}/\bar{P}][x/N] \equiv t[\bar{x}/\bar{P}[x/N]] \xrightarrow[E]{} t'[\bar{x}/\bar{P}[x/N]].$$

A simple induction on the structure of terms proves that

$$t[\bar{x}/\bar{P}] \twoheadrightarrow t[\bar{x}/\bar{Q}] \qquad \text{if} \quad \bar{P} \to \bar{Q}$$

Hence

$$t'[\bar{x}/\bar{P}[x/N]] \twoheadrightarrow t'[\bar{x}/\bar{P}[x/N']] \qquad \text{by induction hypothesis}$$
$$\equiv t'[\bar{x}/\bar{P}][x/N'].$$

2.10. PROPOSITION. *$M \to N$ is weakly Church–Rosser.*

Proof. Assume that $M \to P$ and $M \to Q$. We use induction on $M \to P$: We assume without restriction of generality that M is not of a 1-type τ as then $M \to @\tau$.

(a) $(\lambda x : \tau \cdot K)L \to_{\beta} K[x/L]$:

The second computation may take place either on K or on L

$$\begin{array}{ccc} (\lambda x:\tau\cdot K)L & \longrightarrow & (\lambda x:\tau\cdot K')L \\ {\scriptstyle\beta}\downarrow & & \downarrow{\scriptstyle\beta} \\ K[x/L] & \xrightarrow[2.9]{}\!\!\!\!\to & K'[x/L] \end{array} \qquad \begin{array}{ccc} (\lambda x:\tau\cdot K)L & \longrightarrow & (\lambda x:\tau\cdot K)L' \\ {\scriptstyle\beta}\downarrow & & \downarrow{\scriptstyle\beta} \\ K[x/L] & \xrightarrow[2.9]{}\!\!\!\!\to & K[x/L'] \end{array}$$

(b) $\lambda x : \tau \cdot (M'x) \to_{\eta} M'$:

The second computation may take place within M', say

$$\lambda x : \tau \cdot (M'x) \to \lambda x : \tau \cdot (M''x) \xrightarrow[\eta]{} M'',$$

or is a β-reduction where $M' \equiv \lambda y : \tau \cdot M''$. Then

$$\lambda x : \tau \cdot ((\lambda y : \tau \cdot M'')x) \xrightarrow[\beta]{} \lambda x : \tau \cdot M''[y/x] \equiv M'$$

(we consider terms modulo α-conversion). $M'x$ cannot be of a 1-type as then $\lambda x : \tau \cdot (M'x)$ is of a 1-type.

(c) $p\langle M', N'\rangle \to_\pi M'$: If the second computation proceeds within M' or N', then

$$p\langle M', N'\rangle \to p\langle M'', N'\rangle \to M''$$
$$p\langle M', N'\rangle \to p\langle M', N''\rangle \to M'$$

Otherwise

$$p\langle @\tau, qM\rangle \xrightarrow[\delta]{} pM \xrightarrow[\tau]{} @\tau.$$

$$p\langle pM, @\tau'\rangle \xrightarrow[\delta]{} pM.$$

The other cases are similarly straightforward except for

(d) $\langle pM', qM'\rangle \to M'$: We only consider the interesting cases.

$$\begin{array}{ccccc} \langle pM', qM'\rangle & \to & \langle pM'', qM'\rangle & \to & \langle pM'', qM''\rangle \\ \downarrow{\scriptstyle\delta} & & & & \downarrow{\scriptstyle\delta} \\ M' & & \longrightarrow & & M'' \end{array}$$

The context $\langle p\_\ q\_\rangle$ interacts with M' only if $M' \equiv \langle N', N''\rangle$

$$\begin{array}{ccccc} \langle pM', qM'\rangle & \to & \langle N', q\langle N', N''\rangle\rangle \\ \downarrow & & \downarrow \\ M' & \equiv & \langle N', N''\rangle \end{array}$$

or if $pM \to @\tau$ or $qM \to @\tau'$. Then the newly added δ-rules are applied. The other δ-rules are handled similarly.

The proof of the Church–Rosser properties requires some preparation. We define another calculus with the same terms and reductions except for the fixpoints; fixpoint operators are indexed by natural numbers, Y^i with $i \in \mathbb{N}_0$, and fixpoint reduction is defined by

$$Y^{i+1}(M) \xrightarrow{*} M(Y^i(M)). \qquad (Y^*)$$

(We use the superscript * to refer to this calculus.) The index indicates the number of "recursive calls" to be evaluated. The same technique has been used in [3] and seems to originate in [13]. As to be expected

2.11 LEMMA. *$M \to {}^* N$ is SN and WCR.*

Proof. We just extend the proofs of 2.1 and 2.10. For stability assume that M in $Y^{i+1}(M)$ is stable. An infinite computation starting in $Y^{i+1}(M)$ may have the form

$$Y^{i+1}(M) \xrightarrow{*} Y^{i+1}(M') \xrightarrow[Y^*]{*} M'(Y^i(M')) \xrightarrow{*} \cdots.$$

But M' is stable and, using induction on i, we can assume that Y^i is stable. Then $M'(Y^i(M'))$ is stable by the definition of stability, hence it is SN by 2.2. It is a straightforward exercise to establish the WCR—property of the calculus.

2.12. FACT. *If $M \to^* N$ then $|M| \to |N|$, where $|M|$ is obtained from M by stripping off the indices.*

2.13. LEMMA. *For any computation $M_0 \to M_1 \to \cdots \to M_{n-1}$ there exists an indexed computation $\bar{M}_0 \to^* \bar{M}_1 \to^* \cdots \to^* \bar{M}_{n-1}$ such that $|\bar{M}_i| \equiv M_i$ for $i \in \mathbf{n}$.*

Proof. Index all Y's in M_0 by n.

2.14. PROPOSITION. *$M \to N$ is Church–Rosser (CR).*

Proof. Let

$$M \equiv P_0 \to P_1 \to \cdots \to P_m \equiv P \qquad \text{and} \qquad M \equiv Q_0 \to Q_1 \to \cdots \to Q_n \equiv Q.$$

By 2.13 there exist indexed computations

$$\bar{P}_0 \overset{*}{\to} \bar{P}_1 \overset{*}{\to} \cdots \overset{*}{\to} \bar{P}_m \qquad \text{and} \qquad \bar{Q}_0 \overset{*}{\to} \bar{Q}_1 \overset{*}{\to} \cdots \overset{*}{\to} \bar{Q}_n$$

such that $|\bar{P}_i| \equiv P_i$ and $|\bar{Q}_j| \equiv Q_j$.

We can assume that $\bar{P}_0 \equiv \bar{Q}_0$, otherwise increase the indexes suitably. As $\to^*$ is CR there are computations $\bar{P}_m \to^* R$ and $\bar{Q}_n \to^* R$. Then 2.12 yields the result.

Remark. We conjecture that the calculus is Church–Rosser if the strong normalization condition on the SPEC (E5) has been dropped. But different proof techniques are to be used then. The counter example of Klop [2, 15.3] does not apply due to typing.

2.3. *Normal Forms*

We explore the structure of normal forms. Unfortunately, the normal forms have an unpleasant structure in the general case as only the β-, η-, and Y-reduction do not depend on a specific form of the arguments. The situation is somewhat better for closed terms of a ground type.

2.15. LEMMA. *$\langle t \rangle \twoheadrightarrow$ true or $\langle t \rangle \twoheadrightarrow$ false if $t \in T_{\text{SPEC,bool}}$.*

Proof. By our general assumption $t =_E \text{true}$ or $t =_E \text{false}$. The Church–Rosser property then yields the statement.

2.16. *Let M be a closed Λ SPEC-term of ground type in normal form. Then M has the form $\diamond$, u^*M, v^*N, $\langle M, N \rangle$ with M, N being in normal form or $M \equiv \langle t \rangle$ for some $t \in T_{\text{SPEC}}$.*

Proof. We use induction over terms. The inductive assumption is that the statement holds for all terms of length smaller than that of M. The critical cases are

the conditional and the application. For $M \equiv$ if B then M' else M'' *fi* B must be in normal form, hence by the inductive assumption, typing, and 2.15 either $B \equiv$ true or $B \equiv$ false. We conclude that the term is not reduced.

If $M \equiv M'N$ we use induction over M':

(a) $M' \equiv p$. By the inductive assumption $N \equiv \langle N_0, N_1 \rangle$. Hence M is not reduced. The other cases are handled similarly.

3. Implementation of Abstract Data Types

3.1. *Motivation and Definition*

The notion of implementation states the idea of stepwise refinement more precisely. Program development consists of a hierarchy of specification levels with decreasing abstractness. An implementation builds a bridge between two neighbouring levels with the aim of approaching executable programs. If we have two specifications SPEC 0 and SPEC 1, an implementation of SPEC 0 by SPEC 1 should preserve correctness of SPEC 0-programs. We might call this idea "relativised programming"; the program is developed on the SPEC 0 level but runs on the SPEC 1 level. SPEC 0 sorts and operators are implemented by SPEC 1 data structures and programs. The proceeding captures the task of a programmer implementing a data type by a programming language. The most important property expected to hold for implementations is the composability of the single steps to one large implementation which at last yields a computable program for the very first specification level.

Definition. An *implementation* of SPEC $0 = (S0, \Sigma 0, E0)$ by SPEC $1 = (S1, \Sigma 1, E1)$ is given by a pair of maps $\mathrm{IMPL} = (\mathrm{IMPL}_S, \mathrm{IMPL}_\Sigma)$ such that

$$\mathrm{IMPL}_S : S0 \to D\,\mathrm{Type}(S1)$$

$$\mathrm{IMPL} : \Sigma 0 \to \Lambda\,\mathrm{SPEC}\ 1$$

such that $\mathrm{IMPL}_\Sigma(\sigma) \in \Lambda\ \mathrm{IMPL}_S(s_0) \times \cdots \times \mathrm{IMPL}_S(s_{n-1}) \to \mathrm{IMPL}_S(s)$ if $\sigma : s_0 \cdots s_{n-1} \to s \in \Sigma$. The notation is ambiguously used for the *representation* mappings

$$\mathrm{IMPL}_S : S0 \to \mathrm{TT}\ o(S1), \qquad \mathrm{IMPL}_\Sigma : \Sigma 0 \to \Lambda\,\mathrm{SIG}\ 1$$

which choose elements of the respective equivalence classes.

We extend IMPL to all terms in $T_\Sigma(X)$:

$$\mathrm{IMPL}(x : s) = x : \mathrm{IMPL}_S(s)$$

$$\mathrm{IMPL}(\sigma(t_0, \ldots, t_{n-1})) = \mathrm{IMPL}_\Sigma(\sigma)\ \langle \mathrm{IMPL}(t_0), \ldots, \mathrm{IMPL}(t_{n-1}) \rangle.$$

Notation. We use the scheme

SPEC 1 impl SPEC 0 by
def dom ⋯
ops ⋯ "auxiliary (recursive) definitions"
sortimpl ⋯ "implementation of sorts"
opimpl ⋯ "implementation of operators"

Sorts and operators of SPEC 0 which occur as well in SPEC 1 are only listed if they are redefined.

EXAMPLES. (i) We implement a (obvious) specification of natural numbers by booleans.

BOOL impl NAT by

def dom nat $= 1 + \text{nat}$
sortimpl **nat** $=$ nat
opimpl $0 = u^* \diamond$
$\text{suc} = \lambda x : \text{nat} \cdot v^* x$
$\text{pred} = \lambda x : \text{nat} \cdot \text{case } x, \lambda x : 1 \cdot \diamond, \lambda y : \text{nat} \cdot y \text{ esac}$
$\text{iszero} = \lambda x : \text{nat} \cdot \text{case } x, \lambda x : 1 \cdot \text{true}, \lambda y : \text{nat} \cdot \text{false esac}$

(ii) We implement stacks by arrays with a pointer.

ARRAY impl STACK by

sortimpl **stack** $=$ **array** $\times$ **nat**
opimpl $\text{push} = \lambda \langle s : \textbf{array} \times \textbf{nat}, d : \textbf{data} \rangle \cdot \langle \text{update}(p\ s, q\ s, d), \text{suc}(q\ s) \rangle$
$\text{pop} = \lambda s : \textbf{array} \times \textbf{nat} \cdot \langle p\ s, \text{pred}(q\ s) \rangle$
$\text{top} = \lambda s : \textbf{array} \times \textbf{nat} \cdot \text{get} \langle p\ s, q\ s \rangle$
$\text{empty} = \langle \text{new}, 0 \rangle$

We allow manifold representation of data; an element of type **stack**, for instance, may be represented by different elements of type **array** $\times$ **nat**, especially by arrays which differ in components above the pointer. Moreover, not all the elements of the implementing data structure are used for representation; arrays with non-trivial entries under 0 are not used to represent stacks.

Similarly to [8] we impose semantic constraints on implementations: correct implementations preserve distinctness of data and implement data by finitary structures. Distinctness of data is preserved if terms of $T_{\Sigma 0}$ the implementations of which

are equal in Λ SPEC 1 are already equivalent in SPEC 0. We call this property consistency.

DEFINITION. An implementation IMPL : SPEC $0 \Rightarrow \Lambda$ SPEC 1 is *consistent* if

$$\mathrm{IMPL}(t) = \mathrm{IMPL}(t') \quad \text{implies} \quad t =_{E0} t' \text{ for all } t, t' \in T_{\Sigma 0}$$

(observe that $[t] = [t']$ in $T_{\mathrm{SPEC}\,0}$ if $t =_{E0} t'$).

Elements of $T_{\mathrm{SPEC}\,0}$ should be considered as finitary data. These data are implemented by (recursive) data structures (i.e., terms of ground type) which are potentially infinitary in that fixpoint operators can be used for the representation of operators. We claim that finitary data (structures) should be implemented by finitary data structures. As infinitary data structures correspond to non-terminating programs in our setting, we state the

DEFINITION. A SIG-term M of ground type is called *terminating* if it has a normal form. IMPL *preserves termination* if IMPL(t) is terminating for all $t \in T_{\Sigma 0}$.

At last, the

DEFINITION. IMPL *preserves booleans* if IMPL(true) = true and IMPL(false) = false.

Is used to guarantee the correct interaction with the programming language. We combine these concepts in the

DEFINITION. An implementation IMPL : SPEC $0 \Rightarrow \Lambda$ SPEC 1 is *correct* if IMPL is consistent and preserves termination and booleans.

There is a close connection to the notion of correctness in [8]. Consistency corresponds to RI-correctness, and the termination condition to OP-completeness. For the latter one should observe that OP-completeness states that data are implemented by terms which consist only of operators of SPEC 1 and of the sort implementing operators which exactly correspond to our recursive data structures. Recursive definitions of the SPEC 0-operators thus must unfold finitely.

The rationale of both these notions of implementations is that the implemented operators should operate on implemented data exactly as specified. Hence the equational structure is only preserved as far as closed equations (i.e., equations without variables) are concerned. It will turn out that the implicit assumption that the other equations can be neglected is somewhat fictitious: the operational character of equations is denied which cause problems when composition of implementations are considered. The point will be substantited below.

The implementations given so far are correct. In the case of the stack implementation the termination condition is trivially satisfied. For consistency one checks by induction that a stack contains the same data as the array up to the pointer.

3.2. *Composition of Implementations*

We want to compose implementations IMPL 1 : SPEC 0 $\Rightarrow \Lambda$ SPEC 1 and IMPL 2 : SPEC 1 $\Rightarrow \Lambda$ SPEC 2. Syntactically we intend the following: A SPEC 0-operator $\sigma : w \to s$ has the SPEC 1-implementation IMPL 1(σ). Now replace all SPEC 1-operators $\sigma' : w' \to s'$ in IMPL 1(σ) by their implementations IMPL 2(σ') over SPEC 2. We obtain a SPEC 2-program which is the implementation of σ over SPEC 2. For this purpose we have to extend the implementation IMPL 2 : SPEC 1 $\Rightarrow \Lambda$ SPEC 2 to programs, PIMPL 2 : Λ SPEC 1 $\Rightarrow \Lambda$ SPEC 2. The composition of the implementation is then given by PIMPL 2 * IMPL 1.

DEFINITION. Given an implementation IMPL : SPEC 0 $\Rightarrow \Lambda$ SPEC 1 an *extended implementation*

$$\text{PIMPL} : \Lambda \text{ SPEC } 0 \Rightarrow \Lambda \text{ SPEC } 1$$

consists of a sort implementation $\text{PIMPL}_S : \text{TYPE}(S0) \to \text{TYPE}(S1)$ given by

$$\text{PIMPL}_S(s) = \text{IMPL}_S(s) \qquad \text{for} \quad s \in S0$$

$$\text{PIMPL}_S(\tau \diamond \tau') = \text{PIMPL}_S(\tau) \diamond \text{PIMPL}_S(\tau') \qquad \text{for} \quad \diamond \in \{\times, +, \to\}$$

$$\text{PIMPL}_S(1) = 1$$

$$\text{PIMPL}_S(D_n^i(\tau_0, \ldots, \tau_{n-1})) = D_n^i(\text{PIMPL}_S(\tau_0), \ldots, \text{PIMPL}_S(\tau_{n-1}))$$

(for convenience we identify equivalence classes and typical elements), and a term implementation PIMPL : Λ SIG 0 $\to \Lambda$ SIG 1 given by

$$\text{PIMPL}(x : \tau) = x : \text{PIMPL}_S(\tau)$$

$$\text{PIMPL}(\sigma(\tau_0, \ldots, \tau_{n-1})) = \text{IMPL}_\Sigma(\sigma)\langle \text{PIMPL}(\tau_0), \ldots, \text{PIMPL}(\tau_{n-1})\rangle \qquad \text{for} \quad \sigma : w \to s \in \Sigma$$

$$\text{PIMPL}(MN) = \text{PIMPL}(M)\ \text{PIMPL}(N)$$

$$\text{PIMPL}(\lambda x : \tau \cdot M) = \lambda x : \text{PIMPL}_S(\tau) \cdot \text{PIMPL}(M)$$

$$\text{PIMPL}(\diamond) = \diamond$$

$$\text{PIMPL}(p_{\tau,\tau'}) = p_{\text{IMPL}_S(\tau),\text{IMPL}_S(\tau')}\ \text{PIMPL}(q_{\tau,\tau'}) = q_{\text{IMPL}_S(\tau),\text{IMPL}_S(\tau')}$$

$$\text{PIMPL}(\langle M, N\rangle) = \langle \text{PIMPL}(M), \text{PIMPL}(N)\rangle$$

and so along the structure.

The *composition* IMPL 2 * IMPL 1 : SPEC 0 $\Rightarrow$ SPEC 2 of two implementations IMPL 1 : SPEC 0 $\Rightarrow$ SPEC 1 and IMPL 2 : SPEC 1 $\Rightarrow$ SPEC 2 is defined by

$$\text{IMPL } 2 * \text{IMPL } 1_S = \text{IMPL } 2_S \circ \text{IMPL } 1_S$$

$$\text{IMPL } 2 * \text{IMPL } 1 = \text{PIMPL } 2 \circ \text{IMPL } 1.$$

Remark. Our notion of composition is different to that of [8] as the intermediate specification SPEC 1 is not added to the composed implementation. This is due to the uniformity of the language used for implementation which does not require intermediate specification. Our definition models the notion of composition used for implementations of programming languages.

Now two questions are closely related:

— Is the composition of correct implementations correct again?

— Do consistency and termination conditions still hold for extended implementations?

DEFINITION. An extended implementation PIMPL : Λ SPEC 0 $\Rightarrow$ Λ SPEC 1

(i) is *consistent* if for all terminating terms M, N of closed Λ SPEC 0-terms of the same ground type $M = M'$ if PIMPL(M) = PIMPL(N)

(ii) *preserves termination* if for each closed SPEC-term M of ground type PIMPL(M) terminates if M terminates.

(iii) is *correct* if PIMPL is consistent and preserves termination and booleans.

Remark. The restriction to terms of ground types is necessary as for other types consistency is not guaranteed. If we consider the specification

spec NAT is
BOOL with
sorts **nat**
ops 0: $\rightarrow$ **nat**
  suc : **nat** $\rightarrow$ **nat**
  add : **nat nat** $\rightarrow$ **nat**
  pred : **nat** $\rightarrow$ **nat**
  iszero : **nat** $\rightarrow$ **bool**
eqns add $(x, 0) = x$
  add$(x, \text{suc}(y)) = \text{suc}(\text{add}(x, y))$
  pred$(0) = 0$
  pred$(\text{suc}(x)) = x$
  iszero$(0) = \text{true}$
  iszero$(\text{suc}(x)) = \text{false}$

and define

$$\text{radd} \equiv Y(\lambda f : \mathbf{nat} \times \mathbf{nat} \rightarrow \mathbf{nat} \cdot \lambda x : \mathbf{nat} \times \mathbf{nat} \cdot$$
$$\text{if iszero}(q\ x) \text{ then } p\ x \text{ else suc}(f \langle p\ x, \text{pred}(q\ x) \rangle) \text{ fi})$$

and implement NAT by NAT such that IMPL(add) = radd (a correct implementation) then we obtain PIMPL(radd) = PIMPL(add) but radd $\neq$ add in NAT.

We shall consider some examples which demonstrate that correctness of implementations does not extend to the programming language.

3.1. EXAMPLE. We consider enriched versions of the most simple booleans

spec BOOL is
sorts **bool**
ops true: → **bool**
false: → **bool**

spec BOOL 0 is
BOOL with
ops _ and _: **bool bool** → **bool**
_ or _ : **bool bool** → **bool**

eqns	true and true = true	true or true = true
	false and true = false	false or true = true
	true and false = false	true or false = true
	false and false = false	false or false = false

and

spec BOOL 1 is	spec BOOL 2 is
BOOL 0 with	BOOL 0 with
eqns true or b = true	eqns false and b = false
b or true = true	b and false = false

Trivially $T_{\text{BOOL 0}} = T_{\text{BOOL 1}} = T_{\text{BOOL 2}}$.

We use the obvious implementation IMPL : BOOL 1 ⇒ BOOL 2, which is correct:

(i) PIMPL is not consistent: Consider the BOOL 1-programs

$$P0 \equiv Y(\lambda f : \textbf{bool} \to \textbf{bool} \cdot \lambda b : \textbf{bool} \cdot (\text{false and } (f\ b)))$$

$$P1 \equiv Y(\lambda f : \textbf{bool} \to \textbf{bool} \cdot \lambda b : \textbf{bool} \cdot ((f\ b) \text{ and false}))$$

$P0 \neq P1$ as the only terms equivalent to $P0$ are of the form

$$\lambda b : \textbf{bool} \cdot (\text{false and } \cdots \text{ (false and}(P0))\cdots)$$

and similarly for $P1$. Then

$$\text{PIMPL}(P0 \text{ true}) = \text{false} = \text{PIMPL}(P1 \text{ true}) \text{ but } (P0 \text{ true}) \neq (P1 \text{ true}).$$

(ii) PIMPL does not preserve termination: Consider the program

$$Y(\lambda f : \textbf{bool} \to \textbf{bool} \cdot \lambda b : \textbf{bool} \cdot (\text{true or } (f\ b))).$$

Even if the counterexample may look somewhat artificial it is quite well known if implementation of programming languages are considered: A standard strategy is to evaluate the first argument of a logical operator, and to evaluate a second argument only if further evidence is needed. Such a strategy is encoded in our set of equations with variables in the case of BOOL 1 and another strategy in BOOL 2. The strategy does not cause any harm as long as finitary programs are considered. But it is quite well known that a change of evaluation strategy may give different results in the case of infinitary programs. As our definition of correctness of an implementation only refers to terminating programs the counterexample above should not be too surprising.

EXAMPLE. Consider the implementation of stacks by means of arrays indexed with natural numbers. We take the STACK program

$$P \equiv Y(\lambda f : \textbf{stack} \to \textbf{stack} \cdot \lambda s : \textbf{stack} \cdot \text{pop}(\text{push}(\text{empty}, \text{top}(f\,s)))) \text{ empty}$$

which denotes a constant of type **stack**, and which is equivalent to empty. But the implementation PIMPL(P) is not terminating,

$$\begin{aligned}
\text{PIMPL}(P) \equiv\ & Y(\lambda f : \textbf{array} \times \textbf{nat} \to \textbf{array} \times \textbf{nat} \cdot \lambda s : \textbf{array} \times \textbf{nat}. \\
& \quad \langle p \text{ update } (\ldots), \text{pred}(q \text{ update } (\ldots)) \rangle \\
\to\!\!\to\ & P' \equiv Y(\lambda f : \cdots \lambda s : \cdots \langle \text{update } \langle \text{new}, \text{suc } 0, \text{get } \langle p(f\,s), q(f\,s) \rangle, 0 \rangle) \\
\to\!\!\to\ & \langle \text{update } \langle \text{new}, \text{suc } 0, \text{get} \langle p\ P', q\ P' \rangle, 0 \rangle \\
\to\!\!\to\ & \cdots
\end{aligned}$$

We never get rid of P' and the Y's in it.

Obviously, all counterexamples follow the same pattern: Given an equation of the form

$$t[x_0, \ldots, x_{n-1}, y_0, \ldots, y_{m-1}] = t'[x_0, \ldots, x_{n-1}],$$

where all the variables occuring in a term are listed, a recursive program P is written which contains a subterm

$$t[x_0/M_0, \ldots, x_{n-1}/M_{n-1}, y_0/N_0, \ldots, y_{m-1}/N_{m-1}]$$

such that P is recursively called in some N_i. If the equation holds, the program terminates; otherwise it does not. As implementations do not preserve the "operational" properties of such equations, extended implementations may fail to be correct. Hence one may ask what happens if terminating programs are implemented by terminating programs, avoiding the problems caused by non-termination.

3.2. PROPOSITION. *Let* IMPL : SPEC 0 $\Rightarrow$ A SPEC 1 *be a correct implementation, and let* M, N *be closed* A SPEC 0*-terms of the same ground type such that* M, N,

PIMPL(M), *and* PIMPL(N) *are terminating. Then* $M = N$ *if* PIMPL(M) = PIMPL(N).

As a consequence we obtain our main

THEOREM. (1) *Let* IMPL : SPEC 0 $\Rightarrow$ Λ SPEC 1 *be a correct implementation. If* PIMPL : Λ SPEC 0 $\Rightarrow$ Λ SPEC 1 *preserves termination then* PIMPL *is correct.*

(2) *Let* IMPL 1 : SPEC 0 $\Rightarrow$ Λ SPEC 1 *and* IMPL 2 : SPEC 1 $\Rightarrow$ Λ SPEC 2 *be correct implementations. If the composition* IMPL 2 * IMPL 1 *preserves termination, then* IMPL 2 * IMPL 1 *is correct.*

(3) *For implementations such that the extended implementation preserves termination, composition of correct implementations is correct and associative*

The next section will essentially consist of the proof of 3.2.

The termination condition for extended implementations appears to be natural. It states independency of the operational properties encoded in the equations of a specification, and thus provides a kind of additional condition for correctness of implementations which unfortunately is language dependent.

3.3. *The Proof of the Main Theorem*

We assume that IMPL : SPEC 0 $\Rightarrow$ Λ SPEC 1 is a correct implementation. The main difficulty in the proof of 3.2 is that E-reductions are not preserved by implementations, i.e., $M \to_E N$ does not imply that PIMPL(M) = PIMPL(N). However, we are able to encompass the problem in specific situations. In order to sketch the proof idea we look at the rather simple example where Λ BOOL 2 implements BOOL 1 (compare Example 3.1). Then

$$M \equiv (\text{true or } (Y(\lambda x : \textbf{bool} \cdot \text{false and } x))) \text{ and false}$$

$$\xrightarrow[E]{} \text{true and false} \xrightarrow[E]{} \text{false}$$

in Λ BOOL 1 but

$$(\text{true or } (Y(\lambda x : \textbf{bool} \cdot \text{false and } x))) \text{ and false} \xrightarrow[E]{} \text{false}$$

in Λ BOOL 2. The recursive procedure $Y(\lambda x : \textbf{bool} \cdot \text{false and } x)$ can be replaced by an arbitrary term of type **bool**, for instance true, without affecting the computations:

$$M' \equiv (\text{true or true}) \text{ and false} \xrightarrow[E]{} \text{true and false} \xrightarrow[E]{} \text{false} \qquad (\text{in } \Lambda \text{ BOOL 1})$$

and

$$(\text{true or true}) \text{ and false} \xrightarrow[E]{} \text{false} \qquad (\text{in } \Lambda \text{ BOOL 2}).$$

We conclude that $M = M'$ and PIMPL(M) = PIMPL(M').

Let us assume that there exists another Λ BOOL 1-term N such that $N \twoheadrightarrow_E L$, L being in normal form, such that $\mathrm{PIMPL}(M) = \mathrm{PIMPL}(N)$. Assume that we again replace subterms of N, which do not contribute to the computation of normal forms, by boolean constants, obtaining a term N'. As above, $N = N'$ and $\mathrm{PIMPL}(N) = \mathrm{PIMPL}(N')$ should hold. But then $\mathrm{PIMPL}(M') = \mathrm{IMPL}(M')$ and $\mathrm{PIMPL}(N') = \mathrm{IMPL}(N')$ as M' and N' only contain boolean operators. Consistency of the implementation guarantees $M' = N'$, hence $M = M' = N' = N$.

The hope is that the idea carries over to a more general case in that the computation of a normal form can be rearranged delaying E-reductions (i.e., the normal form can be obtained by computations of the form $L \twoheadrightarrow_{-E} L' \twoheadrightarrow_E L''$). Then we could reduce the general case to the one discussed above as implementations preserve $-E$-redexes.

Unfortunately, the example is too simple minded:

(A) (true or ($Y(\lambda x$: **bool** · false and $x)$)) and true $\rightarrow_E$ true and true $\rightarrow_E$ true in Λ BOOL 1 but

(true or ($Y(\lambda x$: **bool** · false and $x)$)) and true

$\xrightarrow[Y]{}$ (true or ((λx : **bool** · false and x)($Y(\cdots)$)) and true

$\xrightarrow[\beta]{}$ (true or (false and ($Y(\cdots)$))) and true

$\xrightarrow[E]{}$ (true or false) and true

$\xrightarrow[E]{}$ true and true

$\xrightarrow[E]{}$ true in Λ BOOL 2.

Here $-E$-reductions have to take place on the program structure inherited from M to obtain the normal form on the implementation level. Hence there is some difficulty to determine subterms which do not contribute to the computation of the normal form.

(B) In general, E-reductions cannot be delayed:
if true or ($Y(\lambda x$: **bool** · false and $x)$) then true else false fi
$\rightarrow_E$ if true then true else false fi $\rightarrow_\zeta$ true.

To cope with (A) the remedy is at hand: If we compute
(true or ($Y(\lambda x$: **bool** · false and $x)$)) and true

$\xrightarrow[Y]{}$ (true or ((λx : **bool** · false and x)($Y(\cdots)$))) and true

$\xrightarrow[\beta]{}$ (true or (false and ($Y(\cdots)$))) and true $\equiv M'$ (in Λ BOOL 1)

we have the computations

$$M' \xrightarrow[E]{} \text{true and true} \xrightarrow[E]{} \text{true} \qquad (\text{in } \Lambda \text{ BOOL } 1)$$

but we do not refer to the recursive program $Y(\cdots)$ in

$$M' \xrightarrow[E]{} (\text{true or false}) \text{ and true} \xrightarrow[E]{} \text{true and true} \xrightarrow[E]{} \text{true} \qquad (\text{in } \Lambda \text{ BOOL } 2).$$

This observation suggests the following procedure: If no further recursive calls are necessary to compute the normal forms we replace the Y's by $\perp$'s of suitable type, obtaining a term $[[M]]$ from which the same normal forms can be computed (Y does not occur in a normal form by 2.16). Therefore $[[M]] = M$ in $\Lambda_\perp$ SPEC 0. If we then compute the $-E$-normal form, e.g.,

$$(\text{true or } (\perp(\lambda x : \textbf{bool} \cdot \text{false and } x))) \text{ and true} \xrightarrow[\perp]{} (\text{true or true}) \text{ and true}$$

(assuming that $\perp_s \equiv \text{true}$) we obtain a term which is *sufficiently normal* in that all its subterms are of the form $\diamond$, u^*M, v^*N, $\langle M, N\rangle$ or t with $t \in T_{\text{SPEC }0}$. But

3.3. LEMMA. *$M = N$ if $\text{PIMPL}(M) = \text{PIMPL}(N)$ for sufficiently normal terms M, N.*

Proof. By induction over the structure of M. If $M \equiv t$ then $N \equiv t'$ because of typing. As $\text{IMPL}(t) \equiv \text{PIMPL}(t) = \text{PIMPL}(t') \equiv \text{IMPL}(t')$ consistency of IMPL yields $t = t'$. Because of $\text{PIMPL}(\diamond) = \diamond$, $\text{PIMPL}(u^*M) = u^*\text{PIMPL}(M)$ etc., the inductive assumption can be used straightforwardly.

In order to ensure that all the necessary recursive calls are executed we take advantage of the indexed fixpoint operators. Because of 2.12 and 2.13 any computation in Λ SPEC 0 can be viewed as a computation in Λ SPEC 0*. We index all Y's in a Λ SPEC 0-term M by the number of recursive calls needed to compute the normal forms of M and $\text{PIMPL}(M)$. Then we replace the Y^i's by terms R_i, inductively defined by

$$R_0 := \perp$$

$$R_{i+1} := \lambda f : \tau \to \tau \cdot f(R_i f)$$

of suitable type, the resulting term being denoted by $[[M]]$.

3.4. LEMMA. *Let M be a Λ SPEC*-term such that no fixpoint operator occurs in its normal form N. Then $[[M]] \twoheadrightarrow_{-Y} N \twoheadleftarrow^* M$.*

Proof. We first check that $[[M]] \twoheadrightarrow_{-Y} [[N]]$ if $M \to^* N$: We have to simulate Y-reductions,

$$R_{i+1}M \xrightarrow[\beta]{} M(R_iM) \qquad \text{if} \quad Y^{i+1}M \xrightarrow[Y]{*} M(Y^iM)$$

Let $M \twoheadrightarrow^* N$ be a computation such that N is in normal form. Then $[[M]] \twoheadrightarrow [[N]]$, but $[[N]] \equiv N$ as no Y^i occurs in N.

The results obtained so far almost prove 3.2 except for the difficulties caused by the conditional (compare (B) bove). Even if we apply the outlined strategy and compute

$[[$if (true or $(Y^1(\lambda x : \mathbf{bool} \cdot$ false and $x))$ then true else false fi$]]$

$$\begin{aligned}
&= \text{ if (true or}((\lambda x : \mathbf{bool} \to \mathbf{bool} \cdot f(\bot f))(\lambda x : \mathbf{bool} \cdot \text{false and } x))) \\
&\qquad \text{then true else false fi} \\
&\xrightarrow[\beta]{} \text{ if (true or } ((\lambda x : \mathbf{bool} \cdot \text{false and } x)(\bot(\lambda x : \mathbf{bool} \cdot \text{false and } x)))) \\
&\qquad \text{then true else false fi} \\
&\xrightarrow[\beta]{} \text{ if (true or (false and } (\bot(\lambda x : \mathbf{bool} \cdot \text{false and } x)))) \\
&\qquad \text{then true else false fi} \\
&\xrightarrow[\bot]{} \text{ if (true or (false and true)) then true else false fi}
\end{aligned}$$

equational rewriting is necessary to create the ζ-redex. A reassuring observation is the

3.5. LEMMA. PIMPL$(\langle t \rangle) \twoheadrightarrow$ true/false *if* $\langle t \rangle \twoheadrightarrow$ true/false *for* $t \in T_{\text{SPEC 0,bool}}$.

Proof. PIMPL$(\langle t \rangle)$ = IMPL$(\langle t \rangle) \twoheadrightarrow$ true/false implies that $\langle t \rangle \twoheadrightarrow$ true/false because of consistency of IMPL: IMPL$(\langle t \rangle)$ reduces to true or false as IMPL$(\langle t \rangle)$ is a closed terminating term of ground type (the implementation preserves termination, then we use the normal form lemmas 2.16 and 2.15). Assume that $\langle t \rangle \twoheadrightarrow$ true. If IMPL$(\langle t \rangle)$ reduces to false, $\langle t \rangle$ must reduce to false as well. Hence true = false in SPEC 0 in contradiction to the consistency result in Section 1.4 and the general assumption.

The result suggests defining a restricted reduction system replacing E-reductions by

(e) $\langle t \rangle \to_e$ true/false if $\langle t \rangle \to_E$ true/false for $t \in T_{\text{SPEC,bool}}$.

We use $M \to^e N$ to denote this notion of reduction.

3.6. LEMMA. *$M \to^e N$ is Church–Rosser; $M \twoheadrightarrow N$ if $M \to^e N$.*

Proof. Observe that the rewrite system satisfies (E1)–(E5). The Church–Rosser

property of the rewriting system follows from the consistency result and the general assumption.

3.7. PROPOSITION. *Let M be a closed Λ* SPEC 0*-term of ground type such that M and* PIMPL(M) *are terminating. Then there exists a closed $\Lambda_\perp$* SPEC 0*-term M' of ground type such that $M = M'$ and* PIMPL(M) = PIMPL(M') *and such that no fixpoint operator occurs in M'.*

Proof. We extend the calculus Λ SPEC 1* with indexed fixpoint operators by adding (unindexed) fixpoint operators Y with the standard Y-reductions, and denote the resulting calculus by Λ SPEC $1^{(*)}$. Clearly, statements 2.12, 2.13, and 3.4 hold analoguously.

Let n be the number of recursive call at least needed to compute the normal forms of M and PIMPL(M). We index all Y's occuring in M with n, obtaining M'. Then M' and PIMPL(M') have the same normal forms (in Λ SPEC 0* and Λ SPEC $1^{(*)}$, where PIMPL(Y^i) = Y^i) as M and PIMPL(M) (in Λ SPEC 0 and Λ SPEC 1), because no fixpoint operator occurs in both the normal forms (due to the normal form lemma 2.16. Observe that PIMPL(M) is a closed term of ground type as base types are implemented by ground types). We apply 3.4. Forgetting the indexes yields the result.

3.8. LEMMA. *Let M be a closed $\Lambda_\perp$* SPEC*-term of ground type such that no fixpoint operator occurs in M. Then the normal form of M with regard to $\rightarrow^{c}$-reductions is sufficiently normal.*

Proof. Along the lines of 2.16, terms of the form "if B then M' else M'' fi" cannot occur as $B = \langle t \rangle$ with $t \in T_{\text{SPEC,bool}}$ by inductive assumption. But then $B \rightarrow^{c}$ true/false.

We put all these statements together to prove

3.2. PROPOSITION. *Let* IMPL : SPEC $0 \Rightarrow \Lambda$ SPEC 1 *be a correct implementation, and let M, N be closed* SPEC 0*-terms of the same ground type such that M, N,* PIMPL(M), *and* PIMPL(N) *are terminating. Then $M = N$ if* PIMPL(M) = PIMPL(N).

Proof. By 3.6, 3.7, and 3.8 there exist sufficiently normal terms M' and N' such that $M = M'$, $N = N'$, PIMPL(M) = PIMPL(M'), and PIMPL(N) = PIMPL(N'). 3.3 ensures that $M' = N'$ if PIMPL(M') = PIMPL(M) = PIMPL(N) = PIMPL(N'). Hence $M = M' = N' = N$ (observe that this statement essentially depends on the Church–Rosser property of the calculus).

3.4. *Sufficient Conditions for the Correctness of Compositions*

The above results are not completely satisfactory. We would rather have a criterion which guarantees the correctness of extended implementation, but which is easily checked. The analyis of examples which fail to preserve termination give

some hints: It is sufficient to require that the implementation of a term $t \in T_{\text{SPEC }0}(X)$ does not depend on a variable $x \in X$ if the term itself does not depend on x.

DEFINITION. Let IMPL : SPEC $0 \Rightarrow \Lambda$ SPEC 1 be an implementation.

(i) Λ SPEC 1-term N is called IMPL-*representative* of a Λ SPEC 0-term M if there exists a term L such that $M = L$ and $N = \text{PIMPL}(L)$.

(ii) IMPL is called a *strong* implementation if for all $t \in T_{\Sigma\,\text{SPEC }0}$ and for all IMPL-representatives N of t there exists a Λ SPEC 1-term L such that $\text{FV}(L) = \varnothing$.

3.9. THEOREM. *Let* IMPL : SPEC $0 \Rightarrow \Lambda$ SPEC 1 *be a strong and correct implementation. Then the extended implementation* PIMPL : Λ SPEC $0 \Rightarrow \Lambda$ SPEC 1 *is a correct implementation.*

The proof is given by the following lemmas which analyze the use of equations. We assume that IMPL : SPEC $0 \Rightarrow \Lambda$ SPEC 1 is a strong correct implementation.

3.10. LEMMA. *Let* $M \twoheadrightarrow_E N$ *be of ground type with* N *being in normal form. Then* PIMPL(M) *terminates.* PIMPL(M) $\twoheadrightarrow$ true/false *if* $M \twoheadrightarrow_E$ true/false.

Proof. We determine a *prefix* of M by

$$\# \sigma M \# = \# \sigma M \#$$

$$\# \langle M, N \rangle \# = \langle \# M \#, \# N \# \rangle$$

$$\# u^* M \# = u^* \# M \#$$

$$\# v^* N \# = v^* \# N \#$$

otherwise

$$\# M \# = x \qquad \text{where } x \text{ is a "new" variable of suitable type.}$$

Then $M \equiv M[\bar{x}/\bar{M}]$ for some tuple $\bar{M}$ of terms, where $\bar{x}$ is the list of variables occuring in M. Due to typing, any M_i must be of the form PQ with $P \notin \Sigma$. E-reductions do not change the structure of PQ, hence E-reductions either take place within $\# M \#$ or within $\bar{M}$. Thus computation sequences are of the form

$$\# M \# [\bar{x}/\bar{M}] \xrightarrow[E]{} M_1[\bar{x}_1/\bar{M}_1] \xrightarrow[E]{} \cdots \xrightarrow[E]{} M_n[\bar{x}/\bar{M}_n] \equiv N.$$

As N is in normal form no subterms PQ with $P \notin \Sigma$ occur in N, thus $M_n \equiv N$. We conclude that

$$\# M \# \xrightarrow[E]{} M_1 \xrightarrow[E]{} \cdots \xrightarrow[E]{} M_n \equiv N.$$

Assume that M is of base type. Strongness of the implementation implies that there exists a term L such that $\text{PIMPL}(\# M \#) = L$ and $\text{FV}(L) = \varnothing$. Then

$\mathrm{PIMPL}(M) = \mathrm{PIMPL}(\#\ M\ \#\ [\bar{x}/\bar{M}]) = \mathrm{PIMPL}(\#\ M\ \#)[\bar{x}/\mathrm{PIMPL}(\bar{M})] =_{2.9} L[\bar{x}/\mathrm{PIMPL}(\bar{M})] = L$.

If M is of type **bool**, the normal form of $\mathrm{PIMPL}(M)$ is true or false (by the normal form lemma 2.16 as L is of type **bool** and closed). Then the argument of 3.5 is to be applied.

For terms of ground type structural induction works in an obvious way.

3.11. LEMMA. *Let M be a terminating term of ground type. Then* $\mathrm{PIMPL}(M)$ *terminates.*

Proof. We reduce M to obtain the $-E$-normal form L (which exists as the empty set of rewrite rules satisfies (E1)–(E5)). Implementations preserve $-E$-redexes, thus $\mathrm{PIMPL}(M) \to {}_{-E}\mathrm{PIMPL}(L)$. The Church–Rosser property ensures that the normal form N of M (with regard to $\to$) can be computed from L.

The computations must start with a sequence of E-reductions $L \twoheadrightarrow_E P \twoheadrightarrow N$. If $P \equiv N$, $\mathrm{PIMPL}(M)$ terminates because of 3.10. Otherwise the reduction must have the form $L \twoheadrightarrow_E P \twoheadrightarrow_\zeta Q \twoheadrightarrow N$ as E-reductions create $-E$-redexes only if some boolean subterm B reduces to true or false where B occurs in a subterm "if B then L' else L'' fi" of L. We reduce such a B to true (resp. false) obtaining a computation $L \twoheadrightarrow_E P' \twoheadrightarrow_\zeta Q' \twoheadrightarrow N$. By 3.10, $\mathrm{PIMPL}(L) \twoheadrightarrow \mathrm{PIMPL}(P') \twoheadrightarrow_\zeta \mathrm{PIMPL}(Q')$. The procedure is iterated. The iteration terminates because of strong normalization of the indexed calculus Λ SPEC 0*.

An easy criterion for strongness is

3.12. OBSERVATION. *Let* $(t, t') \in E0$ *such that* $\mathrm{FV}(t) \supset \mathrm{FV}(t')$. *Such an equation is called* critical. *If* $\mathrm{PIMPL}(t) = \mathrm{PIMPL}(t')$ *for all critical equations, then* IMPL *is strong.*

Remark. Strongness is not equivalent to preservation of termination as preservation is only assumed for closed terms. If termination is preserved for all terms of ground type, strongness is an equivalent condition.

EXAMPLE. The implementation of STACK by ARRAY is not strong:

$$t \equiv \mathrm{pop}\ \langle \mathrm{push}\ \langle \mathrm{empty}, n \rangle \rangle = \mathrm{empty}$$

does not depend on n, but

$$\mathrm{IMPL}(t) \twoheadrightarrow \langle \mathrm{update}\ \langle \mathrm{new}, \mathrm{suc}\ 0, n \rangle, 0 \rangle$$

depends on n. If we modify the definition of the pop by

$$\mathrm{pop} = \lambda s : \mathbf{array} \times \mathbf{nat} \cdot \langle \mathrm{update}\ \langle ps, qs, 0 \rangle, \mathrm{pred}(qs) \rangle$$

the last entry is erased and the pointer decreased by one, yielding a strong implementation (we assume that the array is initialized by 0, i.e., $\mathrm{get}(\mathrm{new}, i) = 0$).

4. Concluding Remarks and Outlook

1. An implementation step makes a part of a specification more executable. Typically SPEC 1 is an enrichment of SPEC 0, and the enrichment is to be implemented by programs. Parameterized data types may support an analysis of such a situation.

2. We have pointed out that our notion of composition of implementations is different from that of [8] which adds the equations of the intermediate specification to the implementation. Such an approach appears to be somewhat counter intuitive as then the identical implementation of a specification by itself is in general not a unit. Computational experience suggests that we sacrifice preservation of termination (OP-completeness).

3. At first glance there seems to be little connection to the work of Lipeck [14]. But the extension of a data type by recursive data structures is "conservative" in terms of [14, 4.12] which guarantees composability. The termination condition allows reduction of any terminating term to a normal form or, in other words, we prove "conservativeness."

4. The observation of 3 indicates a more methodological aspect: The syntax of a specification should be flexible to enable formalizations closely related to given problems. As a consequence, correctness proofs ("conservativeness") are more complicated. This phenomenon is well known from programming languages.

5. Our approach seems to be closely related to that of [4], where recursive schemes are used as programs. The conditions for correctness of composition given there seem to be a semantic counterpart to our notion of strongness.

6. Several problems remain to be discussed:

— The correctness criteria should be extended to higher types. Quite clearly, the semantics of conversion must then be replaced by full abstraction.

— We can relax our notion of language and implementation in that we

- introduce more general recursive type equations, polymorphism etc.,
- implement base sorts by arbitrary types
- extend the implementation to arbitrary types and higher type operators

adjusting the correctness criteria. Most of the extensions appear to be non-trivial.

— A characterization of specifications (resp. data types) which can be implemented correctly.

Acknowledgment

We thank Samson Abramsky for valuable comments on the final version.

REFERENCES

1. J. A. GOGUEN, J. W. THATCHER, AND E. G. WAGNER, An initial algebra approach to the specification, correctness and implementation of abstract data types, *in* "Current Trends in Programming Methodology, IV: Data Structuring" (R. Yeh, Ed.), Prentice–Hall, Englewood Cliffs, NJ, 1978.
2. H. BARENDREGT, "The Lambda Calculus," 2nd ed., North-Holland, Amsterdam, 1984.
3. G. BERRY, P. L. CURIEN, AND J.-J. LEVY, Full abstraction for sequential languages: The state of the art, Rapport de Recherche No. 197, INRIA, Paris 1983.
4. E. K. BLUM AND F. PARISI-PRESICCE, Implementation of data types by algebraic methods, *J. Comput. System Sci.* **27** (1983), 304–330.
5. M. BROY AND M. WIRSING, Programming languages as abstract data types, *in* "Proceedings, 5ème Coloque de Lille, 1980."
6. J. W. CARTMELL, "Generalized Algebraic Theories and Contextual Categories," Ph. D. thesis, University of Oxford, Oxford, 1978.
7. P.-L. CURIEN, "Categorical Combinators, Sequential Algorithms and Functional Programming," Pitman, London 1986.
8. H. EHRIG, H.-J. KREOWSKI, B. MAHR, AND P. PADAWITZ, Algebraic implementation of abstract data types, *Theoret. Comput. Sci.* **20** (1982).
9. H. EHRIG, H.-J. KREOWSKI, J. W. THATCHER, E. G. WAGNER, AND J. B. WRIGHT, Parameter passing in algebraic specification languages, *Theoret. Comput. Sci.* 1984; *in* "Proceedings ICALP'80," Lect. Notes in Comput. Sci. Vol. 95, Springer, New York, 1980.
10. S. FORTUNE, D. LEIVANT, AND M. O'DONNELL, The expressiveness of simple and second order type structures, *J. Assoc. Comput. Mach.* **30** (1983).
11. H. GANZINGER, "Parameterized Specifications: Parameter Passing and Optimizing Implementation," Bericht Nr. TUM I8110, Tech. Univ., Munich, 1981.
12. J. LAMBECK AND P. SCOTT, "Introduction to Higher Order Categorical Logic, Cambridge Univ. Press, New York, 1986.
13. J.-J. LEVY, An algebraic interpretation of the λ-β-κ-calculus and an application of a labeled lambda-calculus, *Theoret. Comput. Sci.* **2** (1976).
14. U. LIPECK, "Ein algebraischer Kalkuel fuer einen strukturierten Entwurf von Datenabstraktionen," Dissertation, Ber. Nr. 148, Abt. Informatik, Universität Dortmund, 1983.
15. C. PAIR, Abstract data types and algebraic semantics of programming languages, *Theoret. Comput Sci.* **18** (1982).
16. A. POIGNÉ, Another look at parameterization using algebraic specifications with subsorts, Proc. MFCS'84, LNCS 176, 1984.
17. A. POIGNÉ, On specifications, theories and models with higher types, *Inform. and Control* **68** (1986), 1–46.
18. A. POIGNÉ AND J. VOSS, "Programs over Algebraic Specifications—On the Implementation of Abstract Data Types," Ber. Nr. 171, Abt. Informatik, Universität Dortmund, 1983.
19. A. POIGNÉ AND J. VOSS, On the implementation of abstract data types by programming language constructs, *in* "Proceedings CAAP'85," Lect. Notes in Comput. Sci. Vol. 185, Springer, New York, 1985.
20. G. POTTINGER, The Church–Rosser theorem for the typed λ-calculus with extensional pairing, preprint, Carnegie–Mellon University, Pittsburgh, 1979.
21. S. STENLUND, "Combinators, Lambda Terms, and Proof Theory," Reidel, Dordrecht, 1972.
22. W. W. TAIT, Intensional interpretation of functionals of finite type, *J. Symbolic Logic* **32**, 1967.
23. J. VOSS, "Programme über algebraischen Spezifikationen—Zur Implementierung von Abstrakten Datentypen, Diplomarbeit," Abt. Informatik, Universität Dortmund, 1983.
24. D. T. SANNELLA AND M. WIRSING, A kernel language for algebraic specification and implementation, *in* "Proceedings Intl. Conference Foundations of Computation Theory, Lect. Notes in Comput. Sci. Vol. 158, Springer, New York, 1983.