On the Theory of Specification, Implementation, and Parametrization of Abstract Data Types

H.-D. EHRICH

Universitat Dortmund, Dortmund, West Germany

ABSTRACT. In the framework of a category **spec** of equational specifications of abstract data types, implementations are defined to be certain pairs of morphisms with a common target This concept covers, among others, arbitrary recursion schemes for defining the derived operations It is shown that for given single steps of a multilevel implementation, there is always a multilevel implementation composed of these steps, but there is no effective construction of this overall implementation Some suggestions are given for practical composition of implementations utilizing pushouts Parametric specifications and parameter assignments are defined to be special morphisms in **spec**, and parameter substitution is made precise by means of pushouts Since actual parameters can again be parametric, parameter substitution can be iterated. This iteration is shown to be associative While the subject is being treated on a syntactical level in terms of specifications, the initial algebra approach is adopted as providing an appropriate semantics for specifications, and the effects of the present concepts and results on the initial algebras are studied

Categories and Subject Descriptors: D.3 1 **[Programming Languages]** Formal Definitions and Theory, D 3 3 **[Programming Languages]** Language Constructs—*abstract data types*, F.3.1 **[Logics and Meanings of Programs]** Specifying and Verifying and Reasoning about Programs—*specification techniques*, F 3 2 **[Logics and Meanings of Programs]** Semantics of Programming Languages—*algebraic approaches to semantics*

General Terms: Languages, Theory

Additional Key Words and Phrases: abstract implementation, algebraic specification, algebras, category theory, data types, equational specification, implementation, parametrization, parametrized specification

1. *Introduction*

Equational specifications of data abstractions and abstract data types are considered to be a promising design tool in software engineering [14, 16, 28]. The theoretical basis of this method has been investigated by several authors [4–11, 14, 20, 25–27] utilizing initial algebras or algebraic theories.

In this paper we concentrate on two concepts that are central to the theory of abstract data types, namely, implementations and parametric specifications. There are several approaches to make the notion of implementation precise: as a relationship between algebras [14] or between specifications and algebras [6, 7, 11]. A more general functional approach is described in [20]. The approaches to parametrization [15, 25] model the idea of a type constructor: a parametric data type is considered to be a functor sending parameter types to a resultant type.

We consider the concepts of implementation and parametrization on a syntactical level, that is, as a relationship between specifications. Our choice of terminology and notation may need some justification, since it is not yet widely used in computer science. We use the language (but no deeper results) of category theory because we

Author's address Abteilung Informatik, Universitat Dortmund, Postfach 500500, 4600 Dortmund 50, West Germany

feel it is most appropriate for the material presented here. A great deal of the literature cited in this paper makes more or less explicit use of categorical concepts, and we are also encouraged by the availability of excellent introductory texts in category theory [1, 12]. The language of categories provides a clean and perspicacious way to deal with objects, together with certain relationships (morphisms) between them, in a closed and consistent system of notation. In comparison with more conventional notations, it helps to reduce the amount of notational detail, thus achieving more concise and elegant formulations. We found it especially advantageous to utilize the categorical concept of a pushout, both for implementation composition and parameter substitution.

Our mathematical framework is a category named **spec**: the objects are specifications, and the morphisms are certain pairs of mappings on the sorts and the operations. Associated with each specification there is a category of algebras (satisfying the specification), and there is an initial algebra in this category determined uniquely up to isomorphism [11–13]. This initial algebra serves as a semantics of the specification. While we develop our notions on a purely syntactical level, we discuss semantical issues by considering the effects on the associated initial algebras.

In Section 2 we give a brief exposition of the general algebraic background used in the paper. For more details we refer to the literature [1, 12, 13, 19]. In Sections 3 and 4 we develop our specific mathematical machinery, that is, the category **spec** of specifications. The notions of sufficient completeness and consistency in [14] are reflected by special morphisms.

In Section 4 we show that **spec** has pushouts (in fact, **spec** is cocomplete), and we prove some useful theorems about what properties of morphisms carry over to opposite sides of a pushout. The main idea in utilizing the categorical pushout construction is that pushouts describe, roughly speaking, how to build a new object by combining two objects while identifying certain parts of them. A simple instance of a pushout in the category of sets is the union of two sets. Pushouts have been used in the algebraic theory of graph grammars [21, 22] to describe the substitution of subgraphs by graphs and to handle at the same time the connections to the rest graph. In this paper we have two applications for pushouts: they describe how to glue implementation steps together and how to substitute actual parameters into parametric specifications.

In Section 5 we define implementations to be certain pairs of morphisms with a common target. This definition is general enough to include arbitrary recursion schemes for defining the derived operations used for implementation. Our main results concern the composability of implementations. We show that the single steps of a multilevel implementation can always be composed, but there is no general effective composition method. We give some hints and suggestions for a practical composition method utilizing pushouts, where the single nonsystematic step consists of finding a canonical term algebra [6, 11]. Special cases of interest, called full implementations, can be composed in nearly the same way, but there is one more nonsystematic step consisting of, roughly speaking, the completion of partially defined operations.

Parametric specifications are defined in Section 6 to be certain morphisms in **spec**, embedding a formal part into a specification. This notion coincides with that in [25] for the special case where the parameter conditions are equations. For parameter assignments we allow for a great deal of flexibility by requiring that they are morphisms in **spec**. Thus actual parameters may have more operations than prescribed by the formal parameter, and different formal sorts and operations can be assigned the same sort and operation, respectively. For example, from **array of key**

and entry we can get **array of nat and int**, **array of int and (stack of entry)**, **array of int and int** with the same **int**, etc.

Parameter substitution is made precise by means of pushouts. Since it should again be possible to have parametric specifications as actual parameters, parameter substitution can be iterated. We show that this iteration is associative.

A given parametric specification defines a category of possible parameters that may be substituted for the formal part. Our pushout approach results in a functor from this parameter category to **spec**. Thus, we are not so much concerned with type constructors as in [15, 25] but with "specification constructors" as proposed in [3].

2. *Algebraic Background*

Let **set** be the category of sets with functions as morphisms. If $S \in |\mathbf{set}|$ (the class of objects in **set**), we denote by $\mathbf{set}_S$ the comma category whose objects are functions $A: \bar{A} \to S$, S fixed, and morphisms $f: A \to B$ are functions $f: \bar{A} \to \bar{B}$ such that $A = fB$ (cf. Figure 1). Morphism composition is written from left to right, that is, xfg instead of the conventional $g(f(x))$.

If we call the elements of S *sorts*, we may view objects in $\mathbf{set}_S$ to be "S-sorted sets," that is, sets with a sort in S attached to each element. The objects of $\mathbf{set}_S$ are in bijective correspondence with S-indexed families of disjoint sets; for $A: \bar{A} \to S$ define A_s to be the set of all $a \in A$ such that $aA = s$. Many authors use S-indexed families without the disjointness requirement. Morphisms in $\mathbf{set}_S$ are mappings leaving the sort fixed.

Let $S \in |\mathbf{set}|$ be a set of sorts. By $S^*(S^+)$ we denote the set of (nonempty) words over S. An object $\Omega \in |\mathbf{set}_{S^+}|$ is called a *signature* over S. If $x \in S^*$ and $s \in S$, an element $(\omega \mapsto xs) \in \Omega$ is called an *operation (-symbol)*, xs its *index*, x its *domain*, and s its *codomain*. Goguen et al. [11] use "type," "arity," and "sort" for our "index," "domain," and "codomain," respectively.

Given a signature Ω over S we define an endofunctor

$$\hat{\Omega}: \mathbf{set}_S \to \mathbf{set}_S,$$

as follows: Each S-sorted set X of "variables" (or "constants") is sent to the S-sorted set $X\hat{\Omega} := \{([x_1 \ldots, x_p]\omega \mapsto s_{p+1}) \mid \omega \in \Omega,\ x_i \in X,\ \omega\Omega = s_1 \cdots s_p s_{p+1},\ x_i X = s_i \text{ for } 1 \le i \le p\}$ of all simple or atomic formal expressions consisting of an operation symbol applied to an appropriate p-tuple of variables. The source set of $X\hat{\Omega}$ is a set of strings over an alphabet consisting of X, Ω, and the symbols [,], and ,. Morphisms $f: X \to Y \in \mathbf{set}_S$ are sent to the corresponding variable substitutions, that is, $f\hat{\Omega}: [x_1, \ldots, x_p]\omega \mapsto [x_1 f, \ldots, x_p f]\omega$.

Definition 2.1. An Ω-*algebra* is a pair $\mathcal{A} = (A, \delta)$, where $A \in |\mathbf{set}_S|$ is an S-sorted *carrier set* and $\delta: A\hat{\Omega} \to A \in \mathbf{set}_S$ is the *operational structure*.

δ describes the operations of the algebra by associating a value with each formal expression $[a_1, \ldots, a_p]\omega$. Thus, for fixed operation symbol ω we have the associated operation

$$\delta_\omega: A_{s_1} \times \cdots \times A_{s_p} \to A_{s_{p+1}}: (a_1, \ldots, a_p) \mapsto ([a_1, \ldots, a_p]\omega)\delta,$$

where A_s is the subset of all elements in A of sort s.

Definition 2.2. **Ω-alg** is the category of all Ω-algebras. The morphisms $f: (A, \delta) \to (A', \delta')$ are mappings of the carriers $f: A \to A'$ such that $\delta f = (f\hat{\Omega})\delta'$ (see Figure 2).

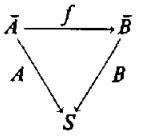
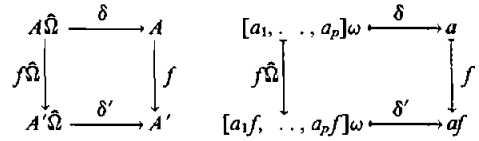


FIGURE 1

FIGURE 2



Let Ω_0, Ω_1 be signatures over sort sets S_0 and S_1, respectively. If $S_0 \subset S_1$ and $\Omega_0 \subset \Omega_1$, there is a forgetful functor

$$U : \Omega_1\text{-alg} \to \Omega_0\text{-alg},$$

sending each Ω_1-algebra $\mathscr{A} = (A_1, \delta_1)$ to the Ω_0-algebra $\mathscr{A}U = (A_0, \delta_0)$, where A_0 is the sorted subset of all elements in A_1 with a sort in S_0, and δ_0 is the restriction of δ_1 to operation symbols in Ω_0 and carrier elements in A_0. Algebra morphisms $f : \mathscr{A} \to \mathscr{A}' \in \Omega_1\text{-alg}$ are sent to $fU : \mathscr{A}U \to \mathscr{A}'U \in \Omega_0\text{-alg}$, where fU is the restriction of f to A_0. If $\mathscr{A} \in \Omega_1\text{-alg}$, $\mathscr{A}U$ is called the *reduct* of $\mathscr{A}$ induced by S_0 and Ω_0 or the (S_0, Ω_0)-reduct of $\mathscr{A}$.

A signature Ω over a sort set S determines an endofunctor

$$T_\Omega : \mathbf{set}_S \to \mathbf{set}_S,$$

associating the sorted set XT_Ω of all Ω-terms over variables X with each sorted set X of variables. Formally, XT_Ω is the least sorted set Y containing X and being closed with respect to an application of $\hat{\Omega}$, that is, the least Y such that (1) $X \subset Y$ and (2) $W \subset Y \Rightarrow W\hat{\Omega} \subset Y$. Morphisms $f : X \to X'$ are sent to fT_Ω given by (1) $x(fT_\Omega) = xf$ and (2) $[t_1, \ldots, t_p]\omega(fT_\Omega) = [t_1(fT_\Omega), \ldots, t_p(fT_\Omega)]\omega$; that is, fT_Ω is the variable substitution corresponding to f.

It is well known that for each $X \in |\mathbf{set}_S|$, $(XT_\Omega, X\gamma)$, where $X\gamma : [t_1, \ldots, t_p]\omega \mapsto [t_1, \ldots, t_p]\omega$, is a free Ω-algebra over X, that is, for each Ω-algebra (A, δ) and each mapping $g : X \to A$ there is a unique Ω-algebra morphism $g^\# : (XT_\Omega, X\gamma) \to (A, \delta)$ extending g such that $g = \eta g^\#$, where $\eta : X \hookrightarrow XT_\Omega$ is the inclusion of generators.

Definition 2.3. The mapping $X\alpha : XT_\Omega \to [[X \to A] \to A] : t \mapsto [g \mapsto tg^\#]$ is called the *evaluation* of terms over X in (A, δ).

We are especially interested in equationally defined Ω-algebras. Let $E \subset YT_\Omega \times YT_\Omega$, given $Y \in |\mathbf{set}_S|$. E is an S-sorted set, and its elements are called Ω-*equations* (or simply equations if Ω is clear from the context). For each equation $e = (t, t')$ let Y_e be the S-sorted set of variables occurring in e (i.e., in t or t'). An Ω-algebra (A, δ) is said to *satisfy* equations E iff $t(Y_e\alpha) = t'(Y_e\alpha)$ for each equation $e = (t, t') \in E$. Equations $(t, t') \in E$ are conventionally denoted by $t = t'$.

Definition 2.4. An ΩE-*algebra* is an Ω-algebra satisfying the given set E of equations. $\Omega\mathbf{E}$-**alg** denotes the full subcategory of all ΩE-algebras in Ω-**alg**.

For each Ω-algebra $\mathscr{A} = (A, \delta)$, a given set E of equations determines a congruence c_E generated by the relation r_E:

$$ar_E b: \quad a = tg^\# \wedge b = t'g^\# \qquad \text{for some} \quad (t = t') \in E \quad \text{and some} \quad g : Y \to A.$$

The quotient algebra $(XT_\Omega^E, X\gamma^E) := (XT_\Omega, X\gamma)/c_E$ is known to be a free ΩE-algebra over X [19]. Thus $(\varnothing T_\Omega^E, \varnothing\gamma^E)$ is an initial object in **ΩE-alg**: since there is a unique mapping from $\varnothing$ to each X, there is a unique morphism from this initial algebra to each ΩE-algebra.

Initial objects in categories are unique up to isomorphism. Therefore isomorphism classes of initial algebras are useful semantic domains for interpreting syntactic structures [11, 13]. We will not differentiate between isomorphic algebras. By "the" initial algebra we thus mean its isomorphism class or any representative of it.

3. *Specifications*

Definition 3.1. A *specification* is a triple $D = (S, \Omega, E)$, where S is a set of sorts, Ω is a signature over S, and E is an S-sorted set of Ω-equations.

As shown in the previous section, with each specification D and each S-sorted set X of variables the free algebra over X can be associated uniquely (up to isomorphism). The initial algebra, that is, the free algebra over $\varnothing$, is called **init** D.

We give some examples using the notation of Guttag [14]. Thus $\omega: s_1 \times s_2 \times \cdots \times s_p \to s_{p+1}$ means that $s_1 s_2 \cdots s_p s_{p+1}$ is the index of the operation ω. Signatures and equations are separated by horizontal lines.

Example 3.2

$$\begin{array}{lll} D_b & \mathit{true}: \to \mathbf{bool} & \\ & \mathit{false}: \to \mathbf{bool} & \end{array} \qquad \begin{array}{ll} D_n & 0: \to \mathbf{nat} \\ & \mathit{succ}: \mathbf{nat} \to \mathbf{nat} \end{array}$$

These are very basic specifications without equations. Clearly, **init** D_b is a two-element set, and **init** D_n is isomorphic to the set of natural numbers generated by the constant 0 and the successor function.

Example 3.3. D_{nb} is obtained by taking D_b and D_n and adding the following items:

$$\begin{array}{c} \mathit{eq}: \mathbf{nat} \times \mathbf{nat} \to \mathbf{bool} \\ \mathit{if\text{-}then\text{-}else\text{-}fi}: \mathbf{bool} \times \mathbf{nat} \times \mathbf{nat} \to \mathbf{nat} \end{array}$$

$$\begin{array}{rll} \mathit{eq}(0, 0) = \mathit{true} & & \\ \mathit{eq}(0, \mathit{succ}(n)) = \mathit{false} & & \mathit{if\ true\ then\ n\ else\ m\ fi} = n \\ \mathit{eq}(\mathit{succ}(n), 0) = \mathit{false} & & \mathit{if\ false\ then\ n\ else\ m\ fi} = m \\ \mathit{eq}(\mathit{succ}(n), \mathit{succ}(m)) = \mathit{eq}(n, m) & & \end{array}$$

The initial algebra **init** D_{nb} has as reducts **init** D_b and **init** D_n, connected by an equality test and a branching operation.

Example 3.4

$$\begin{array}{ll} D_a & \mathit{new}: \to \mathbf{array} \\ & \mathit{store}: \mathbf{array} \times \mathbf{nat} \times \mathbf{nat} \to \mathbf{array} \\ & \mathit{read}: \mathbf{array} \times \mathbf{nat} \to \mathbf{nat} \end{array}$$

$$\begin{array}{l} \mathit{read}(\mathit{new}, n) = 0 \\ \mathit{read}(\mathit{store}(a, n, m), p) = \mathit{if\ eq}(n, p)\ \mathit{then\ m\ else\ read}(a, p)\ \mathit{fi} \end{array}$$

Specification D_{nb} must be added to obtain a complete specification. **init** D_a behaves like an array with indexes and entries in **init** D_{nb} . Actually, D_a specifies an array whose entries are **stacks of nat**, but only the top element of the stack can be accessed. Thus D_a does not specify **array of nat** according to correctness criteria imposed by Goguen et al. [11]. However, it serves its purpose as a useful example in our context.

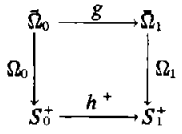


FIGURE 3

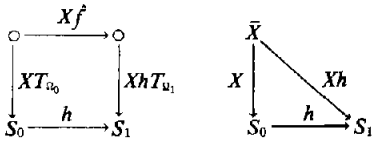


FIGURE 4

Example 3.5

$$\begin{aligned} D_s \quad & create: \rightarrow \textbf{stack} \\ & push: \textbf{stack} \times \textbf{array} \rightarrow \textbf{stack} \\ & pop: \textbf{stack} \rightarrow \textbf{stack} \\ & top: \textbf{stack} \rightarrow \textbf{array} \end{aligned}$$

$$\begin{aligned} pop(create) &= create & top(create) &= new \\ pop(push(s, a)) &= s & top(push(s, a)) &= a \end{aligned}$$

Specification D_a has to be added in order to complete this specification. In the first equations, errors are usually introduced. We avoid doing so in order to keep the examples small and complete. Equational specification of error handling is treated in [11], and [10] contains a semantic approach to errors. **init** D_s is a stack whose entries are taken from **init** D_a.

Relationships between different specifications are generally given by relationships among their sorts, signatures, and equations. Let $D_i = \langle S_i, \Omega_i, E_i \rangle$, $i = 0, 1$, be specifications. The sorts are related by a mapping

$$h: S_0 \rightarrow S_1.$$

Let $h^+: S_0^+ \rightarrow S_1^+$ be the string homomorphism determined by h. Then we can relate operation symbols by a mapping $g: \bar{\Omega}_0 \rightarrow \bar{\Omega}_1$, where $\bar{\Omega}_i$, $i = 0, 1$, are the domains of Ω_i such that $\Omega_0 h^+ = g\Omega_1$ (cf. Figure 3). Thus g is a morphism $g: \Omega_0 h^+ \rightarrow \Omega_1 \in \mathbf{set}_{S_1^+}$.

Given h and g, we can map each term t over signature Ω_0 and S_0-sorted set of variables X to a term $t(X\hat{f})$ over signature Ω_1 by simply replacing each operation symbol $\omega: s_1 \times s_2 \times \cdots \times s_p \rightarrow s_{p+1}$ by its image $\omega g: s_1 h \times s_2 h \times \cdots \times s_p h \rightarrow s_{p+1} h$. Formally, for each X we have a morphism

$$X\hat{f}: XT_{\Omega_0} h \rightarrow XhT_{\Omega_1} \in \mathbf{set}_{S_1}$$

(cf. Figure 4) sending each variable x of sort s to the "same" variable x but viewed as having sort sh, and sending each term $[t_1, \ldots, t_p]\omega$, where each t_i has sort s_i and ω has index $s_1 \times \cdots \times s_p \rightarrow s_{p+1}$, to the term $[t_1(X\hat{f}), \ldots, t_p(X\hat{f})](\omega g)$, where each $t_i(X\hat{f})$ has sort $s_i h$ and ωg has index $s_1 h \times \cdots \times s_p h \rightarrow s_{p+1} h$.

Letting X vary over S_0-sorted sets, we get a natural transformation

$$\hat{f}: T_{\Omega_0} h \rightarrow hT_{\Omega_1},$$

where $T_{\Omega_0} h$ and hT_{Ω_1} are functors from $\mathbf{set}_{S_0}$ to $\mathbf{set}_{S_1}$ sending S_0-sorted sets of variables X to S_1-sorted sets of terms.

Let $E_0 \subset Y_0 T_{\Omega_0} \times Y_0 T_{\Omega_0}$ be an S_0-sorted set of equations. By mapping both sides of each equation by $Y_0\hat{f}$, we get an S_1-sorted set of equations

$$E_0\hat{f} := \{t(Y_0\hat{f}) = t'(Y_0\hat{f}) \mid t = t' \in E_0\}.$$

If $E \subset YT_\Omega \times YT_\Omega$ is a set of equations over some signature Ω and some variable set Y, we can deduce other equations from E by the usual rules of equational calculus,

that is, substitution of terms for variables and the laws of equality. We write $E \vdash t = t'$ if equation $t = t'$ is deducible from E. Let

$$\mathring{E} := \{t = t' \mid E \vdash t = t' \text{ and } t, t' \in \varnothing T_\Omega\}$$

be the set of equations without variables, also called *constant* equations, deducible from E.

Definition 3.6. The category **spec** has specifications $D = \langle S, \Omega, E\rangle$ as objects, and its morphisms $f: D_0 \to D_1$ are pairs of mappings $f = (h, g)$, $h: S_0 \to S_1$, $g: \bar{\Omega}_0 \to \bar{\Omega}_1$, such that (1) $\Omega_0 h^+ = g\Omega_1$ and (2) $\mathring{E}_0 \hat{f} \subset \mathring{E}_1$.

Condition (2) means that all and only the constant equations deducible from E_0 map to valid equations deducible from E_1. The obvious alternative of requiring that all equations in E_0 (and thus all those deducible from E_0) map to valid equations deducible from E_1 is too restrictive for our purpose. This would, for instance, restrict proof methods for the correctness of implementations (cf. Section 5) to deductions in equational calculus, whereas our weaker condition allows for term induction, as suggested by Guttag's approach [14].

Each morphism $f \in$ **spec** determines a natural transformation $\hat{f}$, as shown above. Condition (2) in the above definition means that congruent (w.r.t. E_0) constant terms map to congruent (w.r.t. E_1) constant terms. Thus we can factorize $\varnothing\hat{f}$ by sending each congruence class (w.r.t. E_0) of constant Ω_0-terms to that congruence class (w.r.t. E_1) of constant Ω_1-terms that contains its image. This defines a mapping

$$\varnothing\tilde{f}: \varnothing T^{E_0}_{\Omega_0 h} \to \varnothing T^{E_1}_{\Omega_1}.$$

Definition 3.7. A morphism $f = (h, g) \in$ **spec** is called an *embedding* iff h and g are injective. f is called an Ω-*embedding* iff h is bijective and g is injective.

Up to renaming, an embedding $f: D_0 \to D_1$ describes the situation in which D_0 is a subspecification of D_1, that is, $S_0 \subset S_1$, $\Omega_0 \subset \Omega_1$ and $\mathring{E}_0 \subset \mathring{E}_1$ (note that $E_0 \subset E_1$ is not required). Ω-embeddings denote the special cases where the sort sets are equal, that is, only operation symbols are added.

If $f = (h, g)$ and $X \in |\mathbf{set}_{S_0}|$, let $(\mathbf{init}\ D_1)^0$ be the reduct induced by $S_0 h$ and $(\Omega_0 h^+)g$. Clearly, if f is an embedding, there is an isomorphism from $(\Omega_0 h^+)g$-**alg** to Ω_0-**alg**. We do not make this isomorphism explicit in our notation but consider $(\mathbf{init}\ D_1)^0$ to be an algebra in Ω_0-**alg** (i.e., we take h, g to be inclusions). The next theorem relates embeddings to morphisms between the associated initial algebras.

THEOREM 3.8 *Let $f = (h, g): D_0 \to D_1$ be an embedding. Then $\varnothing\tilde{f}: \mathbf{init}\ D_0 \to (\mathbf{init}\ D_1)^0$ is an Ω_0-algebra morphism.*

The proof is straightforward by the definitions of $\hat{f}$ and $\tilde{f}$.

In what follows, special embeddings to be defined now will be of essential interest. If $f_i = (h_i, g_i): D_i \to D_2$, $i = 0, 1$, are embeddings, let $S_i' = S_i h_i$ and $\Omega_i' = (\Omega_i h_i^+)g_i$. Furthermore, let $T_j = \varnothing T_{\Omega_j}$, $j = 0, 1, 2$, and $T_i' = \varnothing T_{\Omega_i' h_i}$, $i = 0, 1$.

Definition 3.9. f_1 is called

(1) *full w.r.t.* f_0 iff for each term $t_0 \in T_0'$ with a sort in S_1', there is a term $t_1 \in T_1'$ such that $t_0 = t_1 \in \mathring{E}_2$.
(2) *full* iff for each term $t_2 \in T_2$ with a sort in S_1', there is a term $t_1 \in T_1'$ such that $t_2 = t_1 \in \mathring{E}_2$.
(3) *true* iff for all terms $t_1, t_2 \in T_1'$, $t_1 = t_2 \in \mathring{E}_2$ implies $t_1 = t_2 \in \mathring{E}_1 \hat{f}_1$.
(4) *extension* (*w.r.t.* f_0) iff f_1 is a full (w.r.t. f_0) and true embedding.
(5) *enrichment* (*w.r.t.* f_0) iff f_1 is a full (w.r.t. f_0) and true Ω-embedding.

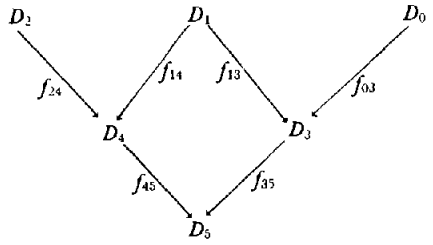


FIGURE 5

These notions of extension and enrichment are compatible with those given in [6, 7, 11]. Clearly, if f_1 is full, it is full w.r.t. any f_0. Full embeddings correspond to sufficiently complete specifications, and true embeddings correspond to consistent specifications, both in the sense of Guttag [14].

Referring to Examples 3.2–3.5, all inclusions $D_b \to D_{nb}$, $D_n \to D_{nb}$, $D_{nb} \to D_a$, $D_a \to D_s$, $D_b \to D_a$, etc., are extensions. If we add, for example, the equation $top(create) = top(push(s, a))$ to D_s, we have a full inclusion $D_a \to D_s$ that is not true. If we drop, for example, the equation $top(create) = new$ in D_s, the inclusion $D_a \to D_s$ is not full but true.

We give some immediate consequences of the above definitions.

PROPOSITION 3.10. *If f and g are both full (true) embeddings (extensions, enrichments), so is fg.*

PROPOSITION 3.11. *Let $f_i: D_i \to D_2$, $i = 0, 1$, be embeddings. With the notation of Definition 3.9 we have*

(1) *If $S_0' \subset S_1'$, then f_1 is full w.r.t. f_0 iff* **(init** D_1**)**$(\varnothing \tilde{f}_1) U_1 \supset$ **(init** D_0**)**$(\varnothing \tilde{f}_0) U_0$. *Here U_i is the respective forgetful functor sending algebras to their carriers.*
(2) *f_1 is full iff $\varnothing \tilde{f}_1$ is surjective.*
(3) *f_1 is true iff $\varnothing \tilde{f}_1$ is injective.*
(4) *f_1 is an extension iff $\varnothing \tilde{f}_1$ is an isomorphism.*
(5) *f_1 is an enrichment iff $\varnothing \tilde{f}_1$ is an isomorphism and h is bijective. In this case,* **(init** D_1**)**^{0} *and* **init** D_1 *have the same carrier set.*

PROPOSITION 3.12. *Let $f_i: D_i \to D_2$ be embeddings, $i = 0, 1$, and let f_1 be full w.r.t. f_0.*

(1) *If $f_0': D_0' \to D_0$ is any embedding, then f_1 is full w.r.t. $f_0' f_0$.*
(2) *If $f_1': D_1' \to D_1$ is a full embedding, then $f_1' f_1$ is full w.r.t. f_0.*

THEOREM 3.13. *Consider the situation depicted in Figure 5, and suppose that all the morphisms are embeddings. If f_{13} is full w.r.t. f_{03} and f_{24} is full w.r.t. f_{14}, then $f_{24} f_{45}$ is full w.r.t. $f_{03} f_{35}$. (Note that the square need not be commutative.)*

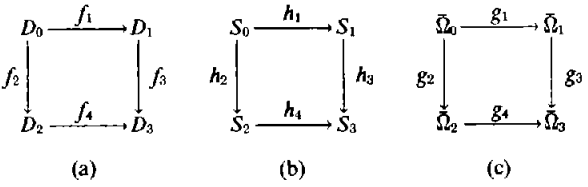
PROOF. In order to facilitate notation we assume that all morphisms are inclusions. Let $T_i = \varnothing T_{\Omega_i}$, $i = 0, \ldots, 5$. Let $t_0 \in T_0$. Since f_{13} is full w.r.t. f_{03}, there is a term $t_1 \in T_1$ such that $t_0 = t_1 \in \mathring{E}_3$. Thus we have $t_0 = t_1 \in \mathring{E}_5$, since f_{35} is a morphism. Moreover, since f_{24} is full w.r.t. f_{14}, there is a term $t_2 \in T_2$ such that $t_1 = t_2 \in \mathring{E}_4 \subset \mathring{E}_5$. It follows that $t_0 = t_2 \in \mathring{E}_5$, proving the theorem. □

This theorem is of a more technical nature and will be utilized in Section 5 when we are discussing constructions for the composition of implementations.

4. *Pushouts*

The categorical pushout construction provides our main technical tool for describing how to put implementation steps together (Section 5) and how to apply parametric

FIGURE 6



specifications to actual parameters (Section 6). In this section we give the mathematical justification for these constructions by showing that they can be done in **spec** and behave as desired.

Formally, a *pushout diagram* in a category is a square like that in Figure 6a (that is commutative and has the additional property that whenever there are two morphisms $f'_3: D_1 \to D'_3$ and $f'_4: D_2 \to D'_3$ such that $f_1 f'_3 = f_2 f'_4$, there is a unique morphism $k: D_3 \to D'_3$ such that $f_3 k = f'_3$ and $f_4 k = f'_4$. In this case the pair of morphisms (f_3, f_4) is called the *pushout* of f_1 and f_2, and D_3 is called the *pushout object*. If a pushout exists, it is determined up to isomorphism. A category *has pushouts* iff a pushout exists for each pair of morphisms (f_1, f_2) with a common source.

The relevance of pushouts for our purposes lies in the fact that they give a neat and concise description of the following situation: Given two objects (D_1 and D_2 in Figure 6a), we want to construct a new object (D_3) by combining them while identifying certain parts of them (as given by D_0, f_1, *and* f_2). The pushout construction gives—in a rough sense—the "minimal" such D_3, and the morphisms f_3 and f_4 tell us what happens to the components D_1 and D_2, respectively, in the "combination" D_3. That is why it is important to know which properties f_3 and f_4 have, dependent on those of f_1 and f_2.

We show that there are pushouts in **spec**, and we investigate how relevant properties of morphisms carry over to opposite sides of pushouts.

THEOREM 4.1. **spec** *has pushouts.*

PROOF. Let $f_1: D_0 \to D_1$ and $f_2: D_0 \to D_2$ be given. We construct $f_3: D_1 \to D_3$ and $f_4: D_2 \to D_3$ such that the diagram in Figure 6a is a pushout. Let $f_i = (h_i, g_i)$ and $D_j = \langle S_j, \Omega_j, E_j \rangle$ for $1 \le i \le 4$ and $0 \le j \le 3$. Let h_3, h_4 and g_3, g_4 be given by the pushouts in **set** depicted in Figures 6b and c, respectively. Then we have the commutative diagram depicted in Figure 7 (without arrow Ω_3). Since the front diagram (6c) is a pushout, there is a unique mapping $\Omega_3: \bar{\Omega}_3 \to S_3^+$ making the diagram in Figure 7 commutative.

Now we have S_3, Ω_3 and $f_3 = (h_3, g_3)$, $f_4 = (h_4, g_4)$, and we know that $\Omega_1 h_3^+ = g_3 \Omega_3$ and $\Omega_2 h_4^+ = g_4 \Omega_3$ hold. We still have to construct equations E_3 such that for $D_3 = \langle S_3, \Omega_3, E_3 \rangle$, the diagram in Figure 6a is a pushout.

Equation set E_3 is defined as follows:

$$E_3 = \mathring{E}_1 \hat{f}_3 \cup \mathring{E}_2 \hat{f}_4.$$

It is evident that f_3 and f_4 are morphisms in **spec**.

That the diagram in Figure 6a is a pushout follows directly from those in Figures 6b and c being pushouts: if $f_5: D_1 \to D_4$ and $f_6: D_2 \to D_4$ are any morphisms such that $f_1 f_5 = f_2 f_6$, there are morphisms h_5, h_6 and g_5, g_6 in **set** such that $h_1 h_5 = h_2 h_6$ and $g_1 g_5 = g_2 g_6$, respectively. Since Figures 6b and c are pushouts, there is exactly one $h: S_3 \to S_4$ and exactly one $g: \bar{\Omega}_3 \to \bar{\Omega}_4$ such that $h_3 h = h_5$, $h_4 h = h_6$ and $g_3 g = g_5$, $g_4 g = g_6$, respectively. Thus there is at most one morphism $f: D_3 \to D_4$ in **spec** such that $f_3 f = f_5$ and $f_4 f = f_6$, namely, $f = (h, g)$. That f is indeed a morphism follows easily from its construction and that of E_3. □

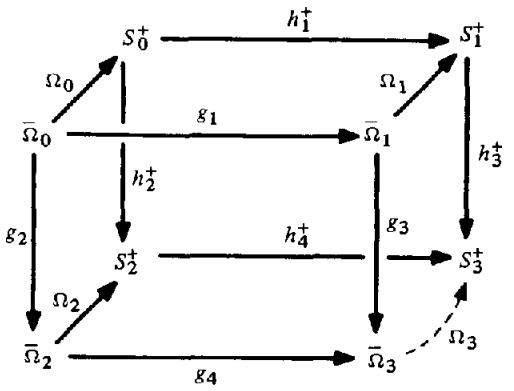


FIGURE 7

We note in passing that $\langle \emptyset, \emptyset, \emptyset \rangle$ is an initial object in **spec**. Thus **spec** has finite colimits [23]. **spec** can even be shown to be cocomplete since there are arbitrary coproducts.

There is some similarity between pushout constructions in **spec** and those in the category of graphs [21, 22] as used in the theory of graph grammars. The connection is established by associating the syntax graph [11] with each specification: the edges are Ω, and the incident nodes are those elements in S^* occurring as domains or codomains of operations in Ω. Then the forgetful functor sending specifications to their syntax graphs preserves pushouts (cf. [5]).

Consider Figure 6a. From the construction of pushouts in **spec** we have immediately that if f_1 is an (Ω-) embedding, the same holds for f_4.

THEOREM 4.2. *If f_1 and f_2 are both full embeddings, so are f_3 and f_4.*

PROOF. Since f_1, f_2 and consequently f_3, f_4 are embeddings, we facilitate our notation by assuming that they are inclusions. Then we have $S_3 = S_1 \cup S_2$, $S_0 = S_1 \cap S_2$ and $\Omega_3 = \Omega_1 \cup \Omega_2$, $\Omega_0 = \Omega_1 \cap \Omega_2$, as well as $E_3 = \mathring{E}_1 \cup \mathring{E}_2$, $\mathring{E}_0 \subset \mathring{E}_1 \cap \mathring{E}_2$, by the pushout construction in **spec**. If both f_1 and f_2 are full, we can reduce each term $t \in T_3$ (again, $T_i := \emptyset T_{\Omega_i}$) with a sort in S_2 to an equivalent term $t' \in T_2$ (actually, T_0) by a bottom-up reduction, removing operation symbols in $\Omega_1 - \Omega_2$ and $\Omega_2 - \Omega_1$, respectively, by applying $\mathring{E}_1$ and $\mathring{E}_2$, respectively. Thus f_4 is full. By symmetry, f_3 is full too. □

THEOREM 4.3. *If f_1 and f_2 are both true embeddings, so are f_3 and f_4.*

PROOF. Again we assume that $f_1, \ldots, f_4$ are inclusions, that is, that we have the same situation concerning the sorts and operations as in the previous proof. Let $\mathring{E}'_i$, $i = 1, 2$, be the subset of $\mathring{E}_i$ consisting of all equations in which only operation symbols from Ω_0 occur. Trueness of f_1 and f_2 means that

$$\mathring{E}_0 = \mathring{E}'_1 = \mathring{E}'_2 = \mathring{E}_1 \cap \mathring{E}_2.$$

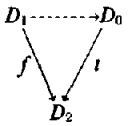
We now want to show that f_4 is true, that is, $\mathring{E}_2 = \mathring{E}'_3$, where $\mathring{E}'_3$ is the subset of $\mathring{E}_3$ consisting of all equations in which only operation symbols from Ω_2 occur. That f_3 is true will then follow from the symmetry of the situation.

Clearly $\mathring{E}_2 \subset \mathring{E}'_3$, by definition of morphism in **spec**. Let $t = t' \in \mathring{E}'_3$, and let

$$t = t_0 \rightarrow t_1 \rightarrow t_2 \rightarrow \cdots \rightarrow t_n = t'$$

be a sequence of terms such that t_{i+1} results from t_i by substituting τ_{i+1} for a subterm τ_i of t_i, according to the equation $\tau_i = \tau_{i+1} \in E_3$. From the pushout construction we have $E_3 = \mathring{E}_1 \cup \mathring{E}_2$.

FIGURE 8



Let $t_p \to t_{p+1}$ be the first step in which an equation in $\hat{E}_1 - \hat{E}_2$ is applied, say $\sigma = \sigma'$. (If there is no such step, then $t = t' \in \hat{E}_2$, and we are done.) Subterm σ of t_p can have only operation symbols in Ω_0. We may assume that σ is a maximal subterm of t_p with this property. In σ', at least one operation symbol $\omega \in \Omega_1 - \Omega_0$ occurs, and all such ω's must be removed in the further reduction process. We assume that this happens in the steps immediately after $t_p \to t_{p+1}$, without being interrupted by reductions affecting only subterms independent from σ' (those using only $\hat{E}_2$ should have been executed before; others can be postponed).

Consequently, reduction goes on with equations in $\hat{E}_1$, say $t_{p+1} \to \cdots \to t_{p+r}$, until all $\omega \in \Omega_1 - \Omega_0$ have been removed from subtree σ' in t_{p+1}. (Since f_2 is true, there can be no intermediate steps using $\hat{E}'_2 - \hat{E}'_1$.) Because of the maximality of σ, the effect of these steps can be achieved by a single step according to, say, $\sigma = \sigma'' \in \hat{E}'_1$. Since f_1 is true, we have $\hat{E}'_1 \subset \hat{E}'_2$. This means that there is a reduction $t \to \cdots \to t_{p+r}$ without using equations in $\hat{E}_1 - \hat{E}_2$. By induction, there is a reduction $t \to \cdots \to t'$ using only equations in $\hat{E}_2$, that is, $t = t' \in \hat{E}_2$. This proves that $\hat{E}'_3 = \hat{E}_2$, that is, that f_4 is true. By symmetry, f_3 is true. □

COROLLARY 4.4. *If f_1 and f_2 are both extensions, so are f_3 and f_4. If additionally f_1 is an enrichment, so is f_4.*

5. *Implementations*

Subsequent to Guttag's paper [14] there have been several approaches to making the notion of implementation mathematically precise [4, 6, 7, 11, 20]. Our approach here is based on that given in [4], and there are some connections to the approaches of Goguen et al. [11] and Ehrig et al. [6, 7]. We will comment on these connections at the end of this section.

Roughly speaking, a specification D_1 implements a specification D_0 if the operations in D_0 can be associated with derived operations in D_1 realizing the behavior expressed in the equations of D_0. If we add new operation symbols for the derived operations and corresponding defining equations to D_1, we get another specification D_2, and there are obvious morphisms from both D_0 and D_1 to D_2.

Definition 5.1. An *implementation of D_0 by D_1* is a triple $I = (D_2, f, t)$, where $f: D_1 \to D_2$ is an Ω-embedding that is full w.r.t. t and $t: D_0 \to D_2$ is a true embedding (cf. Figure 8). I is called a *full* implementation iff f is full. D_1 *implements* D_0 (*fully*) iff there is a (full) implementation I of D_0 by D_1. We use the notation $I: D_1 \dashrightarrow D_0$ if I is an implementation of D_0 by D_1, and we write $D_1 \gg D_0$ if D_1 implements D_0.

Please note that our definition of implementation is general enough to include (1) arbitrary recursion schemes for specifying the derived operations used for implementation, (2) identification of (derived) operations that are different in D_1, since f need not be true, and (3) the existence of "redundant" items in D_2 that have no interpretation in D_0, since t need not be full.

Example 5.2. We give a simplified version of Guttag's symbol table [14] (cf. also [6, 7]) and implement it by the specification D_s of Example 3.5. We assume that identifiers and attributes have already been implemented by **nat**.

D_{sy} is D_{nb} (see Example 3.3) extended by

$$\begin{aligned} \mathit{init}&: \rightarrow \mathbf{Sytb} \\ \mathit{begin}&: \mathbf{Sytb} \rightarrow \mathbf{Sytb} \\ \mathit{end}&: \mathbf{Sytb} \rightarrow \mathbf{Sytb} \\ \mathit{add}&: \mathbf{Sytb} \times \mathbf{nat} \times \mathbf{nat} \rightarrow \mathbf{Sytb} \\ \mathit{retrieve}&: \mathbf{Sytb} \times \mathbf{nat} \rightarrow \mathbf{nat} \end{aligned}$$

$$\begin{aligned} \mathit{end}(\mathit{init}) &= \mathit{init} \\ \mathit{end}(\mathit{begin}(s)) &= s \\ \mathit{end}(\mathit{add}(s, i, a)) &= \mathit{end}(s) \\ \mathit{retrieve}(\mathit{init}, i) &= 0 \\ \mathit{retrieve}(\mathit{begin}(s), i) &= \mathit{retrieve}(s, i) \\ \mathit{retrieve}(\mathit{add}(s, i, a), j) &= \mathit{if}\ \mathit{eq}(i, j)\ \mathit{then}\ a\ \mathit{else}\ \mathit{retrieve}(s, j)\ \mathit{fi} \end{aligned}$$

In order to implement D_{sy} by D_s, we specify D_2 as follows: D_2 consists of D_s plus the following operations and equations:

$$\begin{aligned} \mathit{init}'&: \rightarrow \mathbf{stack} \\ \mathit{begin}'&: \mathbf{stack} \rightarrow \mathbf{stack} \\ \mathit{end}'&: \mathbf{stack} \rightarrow \mathbf{stack} \\ \mathit{add}'&: \mathbf{stack} \times \mathbf{nat} \times \mathbf{nat} \rightarrow \mathbf{stack} \\ \mathit{retrieve}'&: \mathbf{stack} \times \mathbf{nat} \rightarrow \mathbf{nat} \end{aligned}$$

$$\begin{aligned} \mathit{init}' &= \mathit{push}(\mathit{create}, \mathit{new}) \\ \mathit{begin}'(s) &= \mathit{push}(s, \mathit{new}) \\ \mathit{end}'(s) &= \mathit{push}(\mathit{pop}(\mathit{pop}(s)), \mathit{top}(\mathit{pop}(s))) \\ \mathit{add}'(s, i, a) &= \mathit{push}(\mathit{pop}(s), \mathit{store}(\mathit{top}(s), i\ a)) \\ \mathit{retrieve}'(\mathit{create}, i) &= 0 \\ \mathit{retrieve}'(\mathit{push}(s, \mathit{new}), i) &= \mathit{retrieve}'(s, i) \\ \mathit{retrieve}'(\mathit{push}(s, \mathit{store}(a, k, e)), i) & \\ = \mathit{if}\ \mathit{eq}(k, i)\ \mathit{then}\ e\ \mathit{else}\ \mathit{retrieve}'(\mathit{push}(s, a), i)\ \mathit{fi} & \end{aligned}$$

Clearly, the inclusion $f: D_s \rightarrow D_2$ is a full embedding, so we have a full implementation. We define $t = (h, g): D_{sy} \rightarrow D_2$ by

$$\begin{aligned} h&: \mathbf{Sytb} \mapsto \mathbf{stack} \\ \sigma &\mapsto \sigma && \text{for all other sorts } \sigma, \\ g&: \omega \mapsto \omega' && \text{for } \omega \in \{\mathit{init}, \mathit{begin}, \ldots, \mathit{retrieve}\}, \\ \tau &\mapsto \tau && \text{for all other operations } \tau. \end{aligned}$$

The correctness proof for this implementation consists of showing that t is a true embedding. The first part is to show that t is a morphism, that is, that the constant equations of D_{sy} carry over to valid equations in D_2. This is a straightforward exercise, and it has been done for several examples in [14]. The second part is to show that t is true. In our example it is easy to see that init', $\mathit{begin}'(s)$, $\mathit{begin}'(t)$, $\mathit{add}'(s, i, a)$, $\mathit{add}'(t, i, a)$ are pairwise unequal in D_2 if s and t are unequal. A general possibility for giving this part of the correctness proof is to give an "interpretation function" Φ as is done in [6, 7, 14]:

$$\begin{aligned} \Phi[\mathit{create}] &= \mathit{init} \\ \Phi[\mathit{push}(s, \mathit{new})] &= \mathit{begin}(\Phi[s]) \\ \Phi[\mathit{push}(s, \mathit{store}(\mathit{ar}, i, \mathit{at}))] &= \mathit{add}(\Phi[\mathit{push}(s, \mathit{ar})], i, \mathit{at}) \end{aligned}$$

Next we consider the semantic issues of our notion of implementation, that is, its effect on the associated initial algebras.

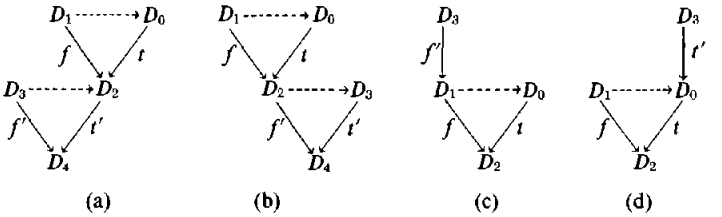


FIGURE 9

PROPOSITION 5.3. *If $I = (D_2, f, t)$ is an implementation of D_0 by D_1, we have*

(1) *an Ω_1-algebra morphism $\varnothing\tilde{f}$: **init** $D_1 \to$ (**init** D_2)^1 and*
(2) *an injective Ω_0-algebra morphism $\varnothing\tilde{t}$: **init** $D_0 \to$ (**init** D_2)^0 such that*
(3) (**init** D_1)$(\varnothing\tilde{f})U_1 \supset$ (**init** D_0)$(\varnothing\tilde{t})U_0$.

If I is full, $\varnothing\tilde{f}$ is a surjective Ω_1-algebra morphism. (**init** D_2)^1 *and* **init** D_2 *have the same carrier.*

The proof is immediate from the definitions and from Proposition 3.11.

The following proposition gives some more simple consequences that are useful for getting new implementations from given ones.

PROPOSITION 5.4. *Let $I = (D_2, f, t) : D_1 \dashrightarrow D_0$ be an implementation.*

(1) *If $(D_4, f', t') : D_3 \dashrightarrow D_2$ is a (full) implementation, then we have a (full) implementation $(D_4, f', tt') : D_3 \dashrightarrow D_0$ (cf. Figure 9a).*
(2) *If $(D_4, f', t') : D_2 \dashrightarrow D_3$ is a (full) implementation and I is full, then we have a (full) implementation $(D_4, ff', t') : D_1 \dashrightarrow D_3$ (cf. Figure 9b).*
(3) *If $f' : D_3 \to D_1$ is a full Ω-embedding, then we have an implementation $(D_2, f'f, t) : D_3 \dashrightarrow D_0$ that is full iff I is full (cf. Figure 9c).*
(4) *If $t' : D_3 \to D_0$ is a true embedding, then we have an implementation $(D_2, f, t't) : D_1 \dashrightarrow D_3$ that is full iff I is full (cf. Figure 9d).*

For the special case where $D_0 = D_1 = D_2$, we see from (3) and (4) that a true embedding $t : D_3 \to D_0$ yields a full implementation in the opposite direction, $D_0 \dashrightarrow D_3$, and a full Ω-embedding $f : D_3 \to D_0$ yields a full implementation in the same direction, $D_3 \dashrightarrow D_0$.

In practice, it is essential that implementations can be done stepwise in multiple levels. For example, if we have an implementation I_1 of a symbol table in terms of stacks (like that in Example 5.2), and if we have another implementation I_2 of stacks in terms of, say, arrays with integer top pointers, then it should be possible to construct an implementation I of a symbol table in terms of arrays with integer top pointers that is in some sense the composition of I_1 and I_2.

Considering Figure 10, we show that there is always an overall implementation $I : D_2 \dashrightarrow D_0$ if we are given I_1 and I_2. This will be an easy consequence of our next theorem, which tells us that there is always an implementation of any D_0 by any D_1, provided that D_1 supports a "sufficient number of items." Of course, this is a necessary condition, too.

THEOREM 5.5. *Let $D_i = \langle S_i, \Omega_i, E_i \rangle$, $i = 0, 1$, be specifications. D_1 implements D_0 ($D_1 \gg D_0$) iff there are two injective mappings*

$$h : S_0 \to S_1 \in \mathbf{set},$$
$$\alpha : \varnothing T^{E_0}_{\Omega_0} h \to \varnothing T^{E_1}_{\Omega_1} \in \mathbf{set}_{S_1}.$$

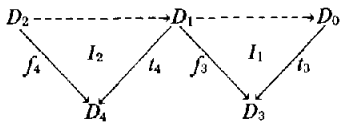


FIGURE 10

PROOF. In order to facilitate notation, we assume that h is an inclusion. Then we construct $I = (D_2, f, t): D_1 \dashrightarrow D_0$ as follows: $D_2 = \langle S_1, \Omega_0 + \Omega_1, E_0 + E_1 + E' \rangle$, where + denotes disjoint union, f is the obvious inclusion, $t = (h, g)$ where g is the obvious inclusion, and $E' = \{t' = t'' \mid ([t'] \mapsto [t'']) \in \alpha\}$. Here $[t']$ denotes the congruence class containing t'. Actually, it would be sufficient if we include in E' just one pair of representatives $t' = t''$ for each element $([t'] \mapsto [t'']) \in \alpha$. Then we have $\alpha = \tilde{\varnothing t}$, that is, t is true, and for each $t' \in \varnothing T_{\Omega_0}$ there is a $t'' \in [t']\alpha$ such that $t' = t'' \in E'$. Hence f is full w.r.t. t. □

This theorem holds for full implementations, too. This can be proved following the lines of the above proof, but we must introduce more equations in order to define the derived operations $\omega \in \Omega_1$ totally: for each $\omega \in \Omega_1$ and each argument p-tuple $([t_1], \ldots, [t_p])$ of constant term classes (in $\varnothing T_{\Omega_1}^{E_1}$), where at least one $[t_i]$ is not in $(\varnothing T_{\Omega_0}^{E_0} h)\alpha$, we must introduce an equation $\langle t_1, \ldots, t_p \rangle \omega = t$, for some representatives $t_1, \ldots, t_p$ and some $t \in \varnothing T_{\Omega_1}$ of the appropriate sort.

COROLLARY 5.6. *$D_2 \gg D_1$ and $D_1 \gg D_0$ implies $D_2 \gg D_0$.*

Again, under the above mentioned condition, this corollary holds for full implementations too.

The proof of Theorem 5.5 is, practically, not very useful. Proposition 5.4, however, gives some suggestions on how to construct compositions of implementations: considering Figure 10, we get an overall implementation $I: D_2 \dashrightarrow D_0$ if we succeed in finding an implementation $I_{23}: D_2 \dashrightarrow D_3$ or $I_{43}: D_4 \dashrightarrow D_3$ or, in case I_2 is full, $I_{40}: D_4 \dashrightarrow D_0$. I_{23} must give derived operations based on D_2 not only for the D_1 operations, but also for the derived operations based on D_1, given in I_1 for the D_0 operations. This means, practically, that I_{23} explicitly "reprograms" implementation I_1 in terms of D_2. Considering I_{40}, we must give derived operations for the D_0 operations in terms of the D_4 operations, and these consist of all D_2 operations plus the derived operations used in I_2 for the D_1 operations. This means, practically, that implementation I_2 is extended explicitly in order to implement D_0. The third possibility, finding implementation I_{43}, combines these features. It means, practically, that I_1 is "reprogrammed" in terms of I_2.

In these constructions we utilize only part of the information available. For I_{23} we do not make use of the derived operations of I_2, and for I_{40} we do not exploit I_1. I_{43} gives only a little improvement: it allows the use of I_2, but still, not only the D_1 operations but also the derived operations of I_1 have to be implemented. The question is whether we can do better.

In the case where f_3 is an enrichment w.r.t. t_3, Theorems 3.13 and 4.3 give a pleasant construction of an overall implementation $D_2 \dashrightarrow D_0$ (cf. Figure 11): the pushout (D_5, f_5, t_5) of f_3 and t_4 gives immediately a true t_5 and a composite implementation,

$$(D_5, f_4 f_5, t_3 t_5): D_2 \dashrightarrow D_0.$$

This means, practically, that the derived operations for D_0 in terms of D_1 (represented by f_3) have been translated automatically to derived operations for D_0 in terms of D_2 (represented by $f_4 f_5$), so that we have fully utilized what we had already.

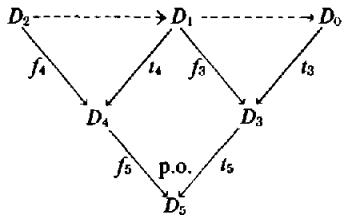
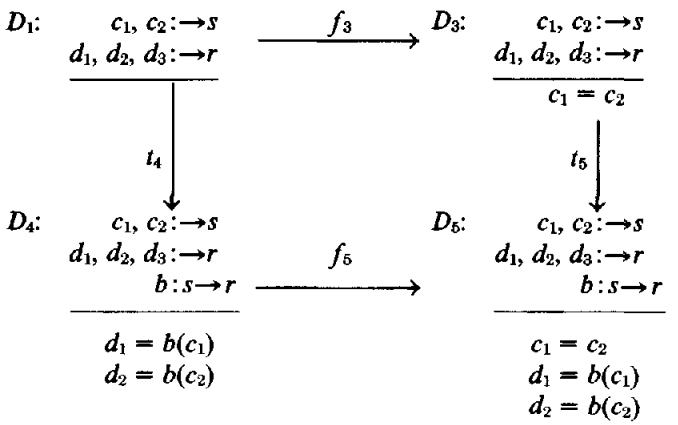


FIGURE 11

Unfortunately, in most practical cases f_3 is not an enrichment w.r.t. t_3. For example, if we consider a stack implemented by an array with an integer top pointer, we would like to express the idea that arrays whose contents differ only beyond the top are equal as stacks. The corresponding f would then not be true.

In such cases pushouts do not give composite implementations in general. This is demonstrated by the following counterexample. We give only D_1, D_3, D_4, D_5 and f_3, t_4, f_5, t_5, assuming that t_3 and f_4 are identities.

Example 5.7 (*due to U. Lipeck*)



The morphisms are the obvious inclusions given by equal denotations. It is easily checked that (D_5, f_5, t_5) is a pushout of f_3 and t_4, f_3 is full, and t_4 is an extension. However t_5 is not true, since in D_5 we can conclude that $d_1 = b(c_1) = b(c_2) = d_2$.

This counterexample shows that implementations cannot be composed in general by just taking the pushout. Moreover, pushouts are not extendible in general to implementations by some simple modifications, for example, adding equations.

In a less direct sense, however, we can utilize pushouts very well in practical cases. Let us consider a full w.r.t. t_3 but nontrue f_3. Such an f_3 is most often given by two nontrivial factors, $f_3 = f_{31}f_{32}$, where f_{31} is true and f_{32} just adds equations (i.e., h_{32} and g_{32} are both bijective, thus f_{32} is full). Intuitively speaking, f_{31} represents the addition and definition of derived operations terminating for the relevant arguments, while f_{32} expresses which items should be considered equal with respect to the implementation.

Let $f_{32} : D_3' \rightarrow D_3$, and suppose we have an implementation (cf. Figure 12) $I_3 = (D_3'', f_3', t_3') : D_3' \dashrightarrow D_3$, where f_3' is an enrichment w.r.t. t_3'. Then we can construct (D_5, f_5, t_5) as the pushout of $f_{31}f_3'$ and t_4. Theorem 4.3 guarantees that t_5 is true, and Theorem 3.13 gives us the desired result that $I = (D_5, f_4f_5, t_3t_3't_5)$ is an implementation of D_0 by D_2.

The only nonconstructive step in getting I is to find an appropriate I_3 such that f_3' is an enrichment w.r.t. t_3'. But here we have a very special situation, and I_3 has a

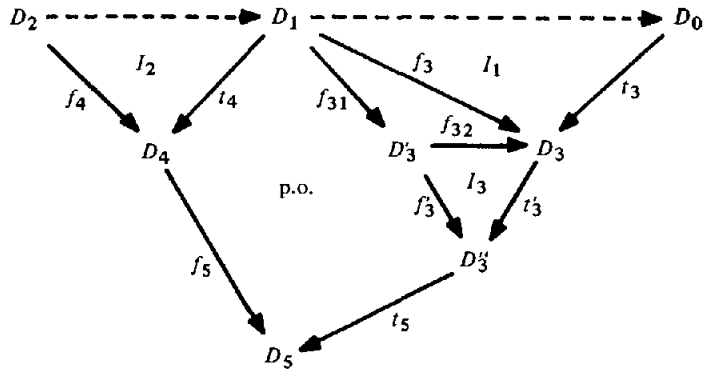


FIGURE 12

natural practical interpretation: finding I_3 means intuitively that we must implement a quotient structure, obtained by adding equations, in terms of the original structure. This can be done by a "canonical term algebra" specification [6, 11] describing how the operations on congruence classes are implemented as operations on a system of distinguished representatives. It is well known that such a canonical term algebra always exists [11], but it cannot be constructed by a general method [6].

If I_1 and I_2 are full implementations and we want to get a full composite implementation, we can do so by extending the above approach, making use of Theorem 4.2, and Theorem 4.3 and Corollary 4.4, respectively.

Practically, a true but nonfull t_4 decomposes most often into two nontrivial factors $t_4 = t_{41}t_{42}$, where t_{41} is an extension and t_{42} describes the true injection of that subspecification of D_4 actually used for implementation into D_4. Let $t_{42}: D'_4 \to D_4$, and suppose we have an implementation $I_4 = (D''_4, f'_4, t'_4): D_4 \dashrightarrow D'_4$, where t'_4 is an extension. Then we have an implementation $I = (D_5, f_4f'_4f_5, t_3t'_3t_5): D_2 \dashrightarrow D_0$ if (D_5, f_5, t_5) is the pushout of $f_{31}f'_3$ and $t_{41}t'_4$. Here we have another nonconstructive step, namely, finding an appropriate I_4. A practical way to find it is to add equations that "make t_{42} full" to D_4 thus getting D''_4. Intuitively, this means that we have to identify items not used for implementation (like an array with negative top pointer in our example of stack implementation) with some items used for implementation. Normally this is done by introducing exceptional "error" constants.

A more detailed discussion of application issues lies outside the scope of this paper. One important aspect is that if we have proven implementation steps I_1 and I_2 correct separately, we cannot conclude that an overall implementation built on these is correct unless we have found I_3 (and, in the case of full implementations, I_4) and corresponding correctness proofs.

The approach to implementation taken in [11] is related to a composition of two special implementations in our sense. Considering Figure 10, implementation $D_2 \dashrightarrow D_1$ corresponds to a Goguen–Thatcher–Wagner derivor from Ω_1 to Ω_2 (with injective sort mapping f) if t_4 is an extension, $\Omega_4 = \Omega_1 + \Omega_2$, and f_4 is the enrichment where each operation $\omega \in \Omega_1$ is defined explicitly by equations of the form $\omega(x_1, \ldots, x_n) = t$ and t is an Ω_2-term over variables $x_1, \ldots, x_n$. If, in the second implementation $D_1 \dashrightarrow D_0$ in Figure 10, f_3 is bijective on sorts and operations, it corresponds to the Goguen–Thatcher–Wagner congruence $\equiv$. Since t_3 is true, we have **init** $D_0 \subsetneq$ **init** D_3 = **init** $D_1 / \equiv$ (cf. Theorem 3.8 and Proposition 3.11). Ehrig et al. [6, 7] generalize the Goguen–Thatcher–Wagner derivor by a functor. An implementation in their sense is obtained by giving a functor from $\mathbf{D_2}$**-alg** to $\mathbf{D_1}$**-alg**, sending **init** D_2 to **init** D_1. Then f_3 defines a surjective homomorphism from **init** D_1 to **init** D_0 if t_3 is an identity and f_3 has the above property.

6. *Parametric Specifications*

Most specifications of parts of programming systems are parametric; that is, they refer to other parts of the system by means of formal entities to be substituted later, possibly in different ways. We give the well known example of a "stack of something."

Example 6.1

$$
\begin{array}{rl}
P_s & \mathit{create}: \rightarrow \textbf{stack} \\
& \mathit{push}: \textbf{stack} \times \textbf{entry} \rightarrow \textbf{stack} \\
& \mathit{pop}: \textbf{stack} \rightarrow \textbf{stack} \\
& \mathit{top}: \textbf{stack} \rightarrow \textbf{entry} \\
& e_0: \rightarrow \textbf{entry} \\
\hline
\multicolumn{2}{c}{\begin{array}{ll} \text{pop(create)} = \text{create} & \text{top(create)} = e_0 \\ \text{pop(push}(s, e)) = s & \text{top(push}(s, e)) = e \end{array}}
\end{array}
$$

Again we avoid introducing errors (cf. Example 3.5). What **entry** and e_0 are is left open. They are formal parameters that can be substituted by actual parameters to get **stack of nat**, **stack of array**, etc. The formal parameter part constitutes a specification itself:

$$F_s = \langle \{\textbf{entry}\}, \{e_0: \rightarrow \textbf{entry}\}, \varnothing \rangle,$$

and we have an embedding $p_s: F_s \rightarrow P_s$ (here, p_s is an inclusion).

Definition 6.2. A *parametric specification* p is an embedding $p: F \rightarrow P$ in **spec**. F is called the *formal parameter part* of p.

We identify nonparametric specifications D with the parametric specifications $\langle \varnothing, \varnothing, \varnothing \rangle \rightarrow D$. The generality of the above definition allows not only formal sorts and operations, but formal equations too. The idea is that only actual parameters satisfying these equations can be substituted. Thus, in order to substitute an actual parameter for a formal one, we have to give an assignment of actual sorts to formal sorts and of actual operations to formal operations such that the equations resulting from the formal equations are valid in the actual parameter.

The actual parameter may itself be a parametric specification. For example, **stack of (stack of entry)** may be a useful concept. So, in general, we have the situation depicted in Figure 13a. We utilize pushouts in order to define what comes out if p_0 is applied to p_1 by means of parameter assignment f (Figure 13b). We must, however, first define what a parameter assignment is, making precise what it means that the formal equations are valid in the actual parameter. We do this in two steps, defining the special case of a nonparametric actual parameter first.

Definition 6.3. Let $p_i: F_i \rightarrow P_i$, $i = 0, 1$, be parametric specifications, and let $f: F_0 \rightarrow P_1$ be a morphism in **spec**. f is called a *parameter assignment from* p_0 *to* p_1 in each of the following cases:

(1) $F_1 = \langle \varnothing, \varnothing, \varnothing \rangle$ and **init** P_1 satisfies $E_0 \hat{f}$, where E_0 are the equations of F_0.
(2) $F_1 \neq \langle \varnothing, \varnothing, \varnothing \rangle$ and, for each parameter assignment $f_1: F_1 \rightarrow D$ in the sense of case (1), **init** P_2 satisfies $E_0 \hat{f}_1''$, where $f_1'' = f f_1'$ and (P_2, p_1', f_1') is the pushout of p_1 and f_1.

The second part of this definition implies that the parameter conditions E_0 hold in each actual parameter that can be obtained as a result of applying p_1 to some nonparametric specification in the sense of the following definition.

Definition 6.4. Let $p_i: F_i \rightarrow P_i$, $i = 0, 1$, be parametric specifications, and let $f: F_0 \rightarrow P_1$ be a parameter assignment from p_0 to p_1. By an *application* of p_0 to p_1 by

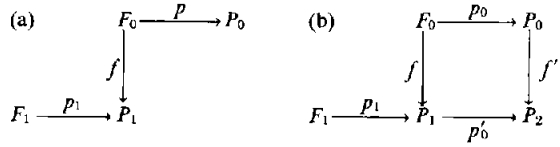


FIGURE 13

f we mean the pushout of Figure 13b. The *result* of this application is the parametric specification $p_2 = p_1 p_0' : F_1 \to P_2$.

That p_2 is indeed a parametric specification is clear from the remark preceding Theorem 4.2. It follows immediately from our pushout theorems in Section 4 that if p_0, p_1, and f are either full or true or both, then the same holds for p_2.

Example 6.5

$$\begin{array}{rl} P_a & \textit{new}: \to \textbf{array} \\ & \textit{store}: \textbf{array} \times \textbf{key} \times \textbf{entry} \to \textbf{array} \\ & \textit{read}: \textbf{array} \times \textbf{key} \to \textbf{entry} \end{array}$$

$$\begin{array}{c} \textit{read}(\textit{new}, k) = e_0 \\ \textit{read}(\textit{store}(a, k, e), h) = \textit{if equ}(k, h) \textit{ then } e \textit{ else read}(a, h) \textit{ fi} \end{array}$$

$$\begin{array}{rl} F_a & e_0 : \to \textbf{entry} \\ & \textit{equ}: \textbf{key} \times \textbf{key} \to \textbf{bool} \\ & \textit{true}: \to \textbf{bool} \\ & \textit{false}: \to \textbf{bool} \\ & \textit{if-then-else-fi}: \textbf{bool} \times \textbf{entry} \times \textbf{entry} \to \textbf{entry} \end{array}$$

$$\textit{equ}(k, k) = \textit{true}$$

P_a is understood to consist of all the sorts, operations, and equations given above. Then $p_a : F_a \subset\!\!\to P_a$ is a parametric specification, and each actual relation substituted for *equ* is required to be reflexive. Of course, additional equations, say, for symmetry and transitivity of *equ*, should be added, together with equations for *if-then-else-fi*.

Let $f : F_a \to D_{nb}$ (cf. Example 3.3) be given by

$$\begin{array}{rlrl} h: & \textbf{entry} \mapsto \textbf{nat} & g: & e_0 \mapsto 0 \\ & \textbf{key} \mapsto \textbf{nat} & & \textit{equ} \mapsto \textit{eq} \\ & \textbf{bool} \mapsto \textbf{bool} & & \textit{true} \mapsto \textit{true} \\ & & & \textit{false} \mapsto \textit{false} \\ & & & \textit{if-then-else-fi} \mapsto \textit{if-then-else-fi} \end{array}$$

In order to prove that f is a parameter assignment, we must prove that $\textit{eq}(n, n) = \textit{true}$ holds in **init** D_{nb}. This can be proved by induction. The result of applying p_a to D_{nb} by means of f is D_a (cf. Example 3.4). Now let $f' : F_s \to P_a$ (cf. Example 6.1) be given by

$$h' : \textbf{entry} \mapsto \textbf{array} \qquad g' : e_0 \mapsto \textit{new}$$

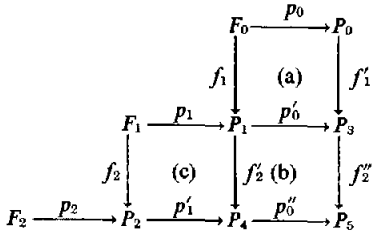
Since F_s has no equations, f' trivially is a morphism. The result of applying p_s to p_a by means of f' is

$$p_{sa} : F_a \to P_{sa},$$

where P_{sa} is P_a extended by the sort **stack** and the stack operations and equations, with **entry** replaced by **array** and e_0 replaced by *new*.

The result of first applying p_a to p_s to get $p_{sa} \triangleq$ **stack of array of key and entry** and then applying p_{sa} to D_{nb} to get $D_s \triangleq$ **stack of array of nat and nat** (cf. Example 3.5)

FIGURE 14



is the same as the result of first applying p_a to D_{nb} to get $D_a \triangleq$ **array of nat and nat** and then applying p_s to D_a. That this is not incidentally so is shown by the next theorem.

THEOREM 6.6. *The application of parametric specifications to parametric specifications by means of parameter assignments is associative.*

PROOF. Let $p_i: F_i \to P_i$, $i = 0, 1, 2$, and $f_j: F_{j-1} \to P_j$, $j = 1, 2$, be given (cf. Figure 14). We have to show that application of p_0 to p_1 by f_1 and then of $p_1 p_0'$ to p_2 by f_2 yields the same result as first applying p_1 to p_2 by f_2 and then applying p_0 to $p_2 p_1'$ by $f_1 f_2'$. That means that we must show that (a) and (c, b) are pushouts iff (c) and (a, b) are pushouts. But both properties hold iff (a), (b), and (c) are pushouts, as we conclude from well known universal pushout properties [23]. □

We now study some consequences of our notion of parameter substitution. For ease of notation we restrict ourselves to nonparametric actual parameters. The ideas can be carried over to the parametric case without any complication.

Let $p: F \to P$ be a parametric specification, and let **spec**$(F, -)$ be the set of morphisms with source F. We take **spec**$(F, -)$ to be the objects of a new category **morph**(F). The morphisms $g: f_0 \to f_1$ in **morph**(F) are those morphisms g in **spec** satisfying $f_0 g = f_1$. There is an obvious forgetful functor $U(F)$: **morph**$(F) \to$ **spec** sending each f to its target.

Let **param**(F) be the full subcategory of **morph**(F) consisting of all parameter assignments. By our pushout approach to parameter substitution we can associate with each parametric specification $p: F \to P$ a functor

$$\Pi: \mathbf{param}(F) \to \mathbf{morph}(P),$$

sending the left side of a pushout diagram like Figure 13b to its right side. To be more precise, let $f_0, f_1 \in |\mathbf{param}(F)|$ and $g: f_0 \to f_1 \in \mathbf{param}(F)$. Furthermore, let $f_0', f_1' \in |\mathbf{morph}(P)|$ result from the pushout of f_0 and f_1, respectively, and p, as shown in Figure 15. Then there is exactly one $g': f_0' \to f_1' \in \mathbf{morph}(P)$ making the diagram commutative, as follows easily from the definition of a pushout.

We now define Π as sending each object $f \in |\mathbf{param}(F)|$ to the corresponding $f' \in |\mathbf{morph}(P)|$ forming the opposite side of the pushout, and sending each morphism g to the unique g' as explained above. The functor

$$\Pi U(P): \mathbf{param}(F) \to \mathbf{spec}$$

reflects the effect of parameter substitution in that each actual parameter (together with a parameter assignment) is sent to the resultant specification.

Considering the initial algebra a parametric specification $p: F \to P$ gives a rule telling how to send an actual parameter type to a resultant type: If $f: F \to D$ is a parameter assignment, then **init** D is sent to **init**$(f \Pi U(P))$. Let this mapping be

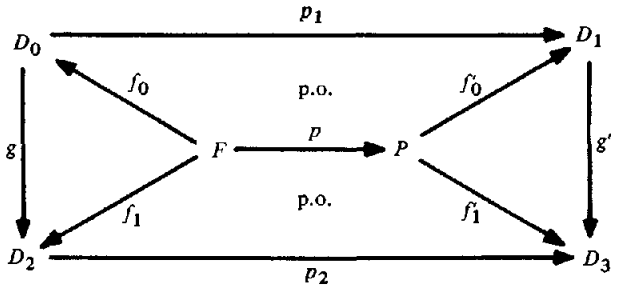
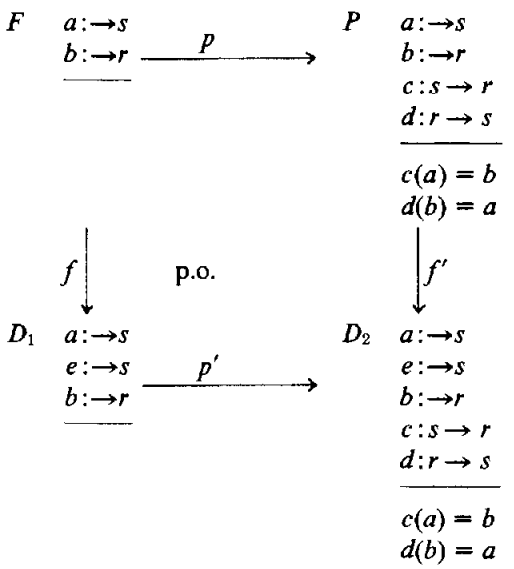


FIGURE 15

denoted by Φ. Lehmann and Smyth [15] as well as Thatcher et al. [25] insist that a parametric data type be a functor from a category of parameter types to a category of resultant types. We could easily extend Φ to a functor by introducing appropriate morphisms between initial algebras, as suggested by our specification morphisms. We feel, however, that this would be somewhat artificial in our approach.

In practice, it is desirable that an actual parameter type A be "preserved" by Φ, that is, that A be isomorphic to a reduct of $A\Phi$. Thatcher et al. [25] confine themselves to such cases by requiring "persistency." Corollary 4.4 and Proposition 3.11 give us sufficient conditions under which we can guarantee this: If a parametric specification $p: F \to P$ and a parameter assignment $f: F \to D$ are both extensions, then **init** D is isomorphic to a reduct of (**init** D)Φ = **init**($f\widetilde{\coprod} U(P)$). It seems to be quite natural to have p be an extension and f be true, but unfortunately this is not the case with f being full, as Example 6.5 shows. If we allow, however, for nonfull f's, the parameter type is not preserved in general. This is illustrated by the following example.

Example 6.7



Obviously p is an extension and f is true but not full. **init** D_1 has carrier $\{a, e, b\}$, disregarding the sorts, but **init** D_2 has carrier $\{a, e, b, c(e), d(c(e)), \ldots\}$. So we cannot have isomorphism.

The situation is more satisfactory if we are content with preserving the actual parameters only as subalgebras of reducts of the resultant types, since p and f need

only be true in order to guarantee this. In such cases the actual parameter type can be "polluted" only by introducing new elements. Actually, this is desired in some cases, for example, if exception or error constants are to be added.

7. *Conclusion*

We have introduced a conceptually simple but powerful notion of implementation of abstract data types as a relation between their equational specifications, and we have made precise the operations of parameter assignment and substitution when dealing with parametric specifications. In both cases we have made profitable use of pushouts in the category **spec** of specifications.

Corollary 5.6 shows that the steps of a multilevel implementation can always be composed to form an overall implementation. While there is no general composition construction, our results suggest a method utilizing pushouts that seems to cover most practical cases. There is one nonsystematic step consisting of, roughly speaking, finding a canonical term algebra. In the case of full implementations, the same method works with one additional nonsystematic step consisting of, roughly speaking, the completion of partially defined operations.

The notion of implementation developed so far does not apply directly to parametric specifications. Of course, we can first substitute actual parameters until we have a nonparametric specification and implement the latter. It would, however, be important to have "parametric implementations" that transform to implementations in our sense if actual parameters are substituted consistently.

Our approach to parametric specifications and parameter substitution defines the rules telling how to construct new specifications by substituting actual parameters for formal ones. The concept of parameter assignment as a morphism in **spec** allows for rather loose and flexible relationships between formal and actual parameters: parametric actual parameters and different sorts and signatures can be handled as well as the assignment of the same actual sort or operation to different formal sorts and operations, respectively. Formal parameter conditions can be formulated as equations, and the existence of a parameter assignment implies that the actual parameter satisfies these equations.

The definition of parameter substitution by pushouts implies a functor from the category of parameters to the resultant specifications, where the category of parameters depends on the parametric specification at hand. We briefly indicate a connection to the semantical approach in [25], where parametrization is understood to define a functor between categories of algebras. Quite a different semantical approach to parametrization is given in [15] in connection with the stepwise solution of systems of domain equations as introduced by Scott [24]. A critical comparison of these two semantical approaches is given in [25].

ACKNOWLEDGMENTS. I am much indebted to H.-J. Kreowski and U. Lipeck for giving valuable comments and criticism on an earlier draft of this paper. I also want to thank H. Ehrig, V. G. Lohberger, J. Thatcher, and E. Wagner for helpful discussions.

REFERENCES

(Note References [2, 17, 18] are not cited in the text)

1. ARBIB, M A, AND MANES, E G *Arrows, Structures, and Functors* Academic Press, New York, 1975.
2. BERGSTRA, J What is an abstract data type? Tech Rep No 77-12, Institute of Applied Mathematics and Computer Science, Univ of Leiden, The Netherlands, 1977
3. BURSTALL, R M, AND GOGUEN, J A Putting theories together to make specifications Proc 5th Int Joint Conf. on Artificial Intelligence, Cambridge, Mass., 1977, pp. 1045–1058.

4 EHRICH, H -D. Extensions and implementations of abstract data type specifications. In Proc. 7th Symp on the Mathematical Foundations of Computer Science, Lecture Notes in Computer Science 64, J. Winkowski, Ed , Springer-Verlag, Berlin, 1978, pp. 155–164

5 EHRICH, H -D , AND LOHBERGER, V G Parametric specification of abstract data types, parameter substitution, and graph replacements. In *Graphs, Data Structures, Algorithms*, Applied Computer Science, Carl Hanser Verlag, Munich, Vienna, 1979, pp 169–182

6 EHRIG, H , KREOWSKI, H -J , AND PADAWITZ, P Some remarks concerning correct specification and implementation of abstract data types Ber Nr 77-13, Institute für Software und Theoretische Informatik, Technische Univ Berlin, FB 20, 1977

7. EHRIG, H , KREOWSKI, H -J , AND PADAWITZ, P Stepwise specification and implementation of abstract data types Internal Rep , Institute für Software und Theoretische Informatik, Technische Univ Berlin, FB 20, 1977

8. GIARRATANA, V , GIMONA, F., AND MONTANARI, U Observability concepts in abstract data type specification In Proc. 5th Symp on the Mathematical Foundations of Computer Science, Lecture Notes in Computer Science 45, A Mazurkiewicz, Ed , Springer-Verlag, Berlin, 1976, pp 576–587

9 GOGUEN, J A Correctness and equivalence of data types In Proc 1975 Conf on Mathematics and Systems Theory, Lecture Notes in Economics and Mathematical Systems 131, G Marchesin, Ed., Springer-Verlag, Berlin, 1976, pp 576–587

10 GOGUEN, J A Abstract errors for abstract data types In Proc Conf. on Formal Description of Programming Concepts, E.J. Neuhold, Ed., North-Holland, Amsterdam, 1978, pp. 491–525.

11 GOGUEN, J A , THATCHER, J W , AND WAGNER, E G 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, N J , 1978, pp 80–149.

12 GOGUEN, J A , THATCHER, J W , WAGNER, E G , AND WRIGHT, J B An introduction to categories, algebraic theories and algebras. Tech Rep. RC 5369, IBM Thomas J Watson Lab , Yorktown Heights, N Y 1975

13 GOGUEN, J A , THATCHER, J W , WAGNER, E G., AND WRIGHT, J B Initial algebra semantics and continuous algebras *J ACM 24*, 1 (Jan 1977), 68–95

14 GUTTAG, J V The specification and application to programming of abstract data types. Tech Rep CSRG-59, Univ of Toronto, Toronto, Ontario, Canada, 1975

15 LEHMANN, D J , AND SMYTH, M B Data types Theory of computation Tech Rep. No. 19, Univ. of Warwick, Coventry, England, 1977

16 LISKOV, B H , AND ZILLES, S.N Specification techniques for data abstractions *IEEE Trans. Softw Eng SE-1* (1975), 7–19

17 MAJSTER, M E Data types, abstract data types and their specification problem Tech. Rep TUM-INFO-7740, Technische Univ Munchen, 1977

18 MAJSTER, M E Limits of the algebraic specification of data types *SIGPLAN Notices 12* (1977), 37–42.

19 MANES, E G *Algebraic Theories* Springer-Verlag, New York, 1976

20 MAYOH, B H Data types as functions Tech Rep DAIMI PB-89, Computer Science Dep , Aarhus Univ., Aarhus, Denmark, 1978

21 ROSEN, B K Deriving graphs from graphs by applying a production *Acta Inf. 4* (1975), 337–357

22 SCHNEIDER, H -J , AND EHRIG, H Grammars on partial graphs *Acta Inf 6* (1976), 297–316

23 SCHUBERT, H *Kategorien* Springer-Verlag, Berlin, 1970

24 SCOTT, D S Data types as lattices *SIAM J Comput 5* (1976), 522–587

25 THATCHER, J W , WAGNER, E G , AND WRIGHT, J B Data type specification. parameterization and the power of specification techniques Proc SIGACT 10th Ann Symp on Theory of Computing, San Diego, Calif , May 1978, pp 119–132

26 WAND, M First-order identities as a defining language Tech Rep No 29, Computer Science Dep., Indiana Univ , Bloomington, Ind , 1976

27 WAND, M Final algebra semantics and data type extensions Tech Rep No 65, Computer Science Dep , Indiana Univ , Bloomington, Ind , 1977

28 ZILLES, S N Algebraic specification of data types M I T Project MAC, Computation Structures Group Memo 119, M I T , Cambridge, Mass 1975, pp 1–12

RECEIVED FEBRUARY 1979, REVISED MARCH 1980, ACCEPTED NOVEMBER 1980