Parametric Specification of Abstract Data Types, Parameter Substitution, and Graph Replacements

H.-D. Ehrich / V.G. Lohberger

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

M. Nagl, H.-J. Schneider (eds)
Proceedings of the Workshop on
"Graphentheoretische Konzepte
in der Informatik", Applied Computer
Science 13, pages 169-182, Carl
Hanser Verlag, München-Wien 1979

PARAMETRIC SPECIFICATION OF ABSTRACT DATA TYPES, PARAMETER SUBSTITUTION, AND GRAPH REPLACEMENTS

H.-D. Ehrich / V.G. Lohberger

Abteilung Informatik, Universität Dortmund,
Postfach 500500, D-4600 Dortmund 50, West Germany

Abstract - Equational specifications of abstract data types can be considered as certain graphs enriched by equations. We introduce a category spec of such specifications and show that spec has pushouts. Moreover, there is a forgetful functor from spec to the category graph of graphs, respecting pushouts. Thus, the patterns of graph replacement by means of pushouts are applicable to specifications. We discuss parametric specifications and parameter substitution on these grounds, and we give an example for a more general replacement operation in analogy to a graph production.

1. Introduction

The promising approaches to a systematic and reliable design of program systems using algebraic methods [8, 9, 13] have raised many questions and problems. Besides the semantics of equational specifications [1, 7], these problems particularly concern their syntactical manipulation, i.e. their stepwise and modular formulation. Especially, parametric specifications and the corresponding mechanisms of parameter substitution play an important role.

While there are two different approaches to semantical aspects of parametrization [2, 9] modelling the idea of a "type constructor", we give a more syntactical approach by considering parametric specifications as "specification constructors", constructing specifications by substituting parameters. This coincides with the intention given in [4]. The relation between the semantical approach of [2] and our approach is discussed in [6].

We utilize pushouts in the category spec of specifications to define the substitution of parameters in parametric specifications. Insofar, there is a formal connection to the use of colimits to cope with the problem of shared subtypes as suggested (but not carried out in detail) by [4].

There is a formal connection, too, to the theory of graph grammars and graph languages where pushouts have been used extensively [11, 12]. As in that case, we have to do with replacements of parts of complex structures. The connection is deepened by the fact that we can associate a syntax graph with each equational specification, reflecting specification replacements by corresponding graph replacements.

In the next section, we introduce the categories spec of specifications [5, 6] and graph of graphs [11, 12]. There is a forgetful functor from spec to graph modelling the concept of the syntax graph. The main results are that spec has pushouts and that the forgetful functor respects pushouts (but does not detect them).

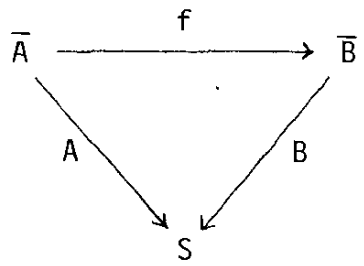
In the third section we make precise the notions of parametric specification, parameter assignment, and parameter substitution. Since actual parameters can be parametric again, parameter substitution can be iterated. We show that this iteration is associative.

In the last section, we give a small example for a more general replacement

operation on specifications in analogy to the concept of a graph production [11, 12]. Then, we give some ideas about possible applications in the field of specification construction and modification. Corresponding applications of graph grammars are discussed in [3].

2. Specifications and their graphs

Let set be the category of sets. The morphisms are the total functions. For a given set S, we have a category $\underline{set}_S$ of "S-sorted sets", whose objects are functions $A:\overline{A}\longrightarrow S \in \underline{set}$, S fixed, and whose morphisms are functions $f:\overline{A}\longrightarrow\overline{B}$ leaving the sorts fixed, i.e. $A=fB$ if $B:\overline{B}\longrightarrow S\in \underline{set}$. (We write composition of morphisms from left to right).



Let S be a given set of sorts. By S^+ we mean the set of nonempty words over S. An object $\Omega\in|\underline{set}_S{}^+|$ is called a signature over S. The elements of Ω are called operations. If $x\in S^+$, $s\in S$, and $(\omega\mapsto xs)\in\Omega$, xs is called the index of Ω, x its domain, and s its codomain.

Associated with each signature Ω, there is an endofunctor $T_\Omega:\underline{set}_S\longrightarrow\underline{set}_S$ sending S-sorted sets X of variables to the s-sorted set of all Ω-terms over X. Morphisms are sent to the corresponding variable substitutions. Given $Y\in|\underline{set}_S|$, the elements of an S-sorted set of pairs of terms $E\subset YT_\Omega\times YT_\Omega$ are called Ω-equations. We use the notation $t_1=t_2\in E$ instead of $(t_1,t_2)\in E$.

2.1 Definition: A specification is a triple $D=\langle S, \Omega, E\rangle$ where S is a set of sorts, Ω is a signature over S, and E is an S-sorted set of Ω-equations, such that for each operation $(\omega\mapsto s_1\ldots s_p s)\in\Omega$ and each $i=1,\ldots,p$, there is an operation $(\pi_i^\omega\mapsto s_1\ldots s_p s_i)\in\Omega$ with an equation $\pi_i^\omega(x_1,\ldots,x_i,\ldots,x_p)=x_i$ in E.

We thus presume that for each domain $s_1\ldots s_p$, there are projections for each component. In our examples we will not explicitly write down these projections.

The models of a specification D form an equational class of algebras containing an "initial algebra" init D that is determined uniquely up to isomorphism. The defining property is that there is a unique homomorphism from init D to each model of D. The idea of initial algebra semantics [1] is to fix init D as the standard semantics of D. For details of the initial algebra construction and the mathematical background we refer to the literature [1, 6].

In order to illustrate the above notions and definitions, we give some examples. We use Guttag's notation [8], denoting an operation $(\omega\mapsto s_1\ldots s_p s_{p+1})$

by $\omega: s_1 \times \ldots \times s_p \longrightarrow s_{p+1}$. Signatures and equations are separated by horizontal lines. The following examples give a four-step specification of a stack whose entries are arrays, whose indices and entries in turn are natural numbers. We start with truth values and natural numbers.

2.2 Example: D_b: true: $\longrightarrow$ bool
false: $\longrightarrow$ bool

D_n: 0 : $\longrightarrow$ nat
succ: nat $\longrightarrow$ nat

There are no equations. init D_b is a two-element set, and init D_n is isomorphic to the set of natural numbers generated by 0 and the successor function.

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

eq: nat × nat $\longrightarrow$ bool
if-then-else-fi: bool × nat × nat $\longrightarrow$ nat

---

eq(0,0) =true
eq(0,succ(n)) =false if true then n else m = n
eq(succ(n),0) =false if false then n else m = m
eq(succ(n),succ(m))=eq(n,m)

init D_{nb} has init D_b and init D_n as reducts, connected by an equality test and a branching operation. Note that we mix infix and prefix notation as we find it convenient.

2.4 Example: D_a is obtained by taking D_{nb} and adding the following items:

new : $\longrightarrow$ array
store: array × nat × nat $\longrightarrow$ array
read : array × nat $\longrightarrow$ nat

---

read(new,n) =0
read(store(a,n,m),p)=if eq(n,p) then m else read(a,p) fi

init D_a behaves like an array whose indices and entries are taken from init D_{nb}.

2.5 Example: D_s is obtained from D_a by adding

create: $\longrightarrow$ stack
push : stack × array $\longrightarrow$ stack
pop : stack $\longrightarrow$ stack
top : stack $\longrightarrow$ array

---

pop(create) =create top(create) =new
pop(push(s,a))=s top(push(s,a))=a

init D_s is a stack whose entries are arrays taken from init D_a.

The specifications in these examples are related in a certain way: each one is contained in those of the subsequent examples. It is very important to have a notion of relationships between specifications. Let $D_i = \langle S_i, \Omega_i, E_i \rangle$, i=0,1, be specifications. Generally, relationships are based on two set mappings:

$$h: S_0 \longrightarrow S_1$$
$$g: \overline{\Omega}_0 \longrightarrow \overline{\Omega}_1$$

where $\bar{\Omega}_i$ are the domains of the signatures, i.e. $\Omega_i: \bar{\Omega}_i \longrightarrow S_i^+$. A natural compatibility condition is that g is a morphism in $\underline{set}_{S_1^+}$ at the same time, namely

$$g: \Omega_0 h^* \longrightarrow \Omega_1 \in \underline{set}_{S_1^+}$$

where $h^*: S_0^* \longrightarrow S_1^*$ is the character by character extension of h to strings. (more precisely: the string homomorphism generated by h). Explicitely stated, the compatibility condition is

$$\Omega_0 h^* = g\,\Omega_1 \ .$$

This means that if we map $\omega \in \bar{\Omega}_0$ by g to $\omega g \in \bar{\Omega}_1$, the index of ω is mapped by h^* to the index of ωg . We can thus map Ω_0-terms to Ω_1-terms simply by mapping each operation symbol by g. Formally, for each variable set $X \in |\underline{set}_{S_0}|$, h and g determine a morphism

$$X\hat{f}: XT_{\Omega_0 h} \longrightarrow XhT_{\Omega_1} \in \underline{set}_{S_1} \ ,$$

by mapping

(1) $(x \mapsto sh) \mapsto (x \mapsto sh)$

(2) $([(t_1 \mapsto s_1), \ldots, (t_p \mapsto s_p)]\omega \mapsto sh) \longmapsto$
$([(t_1 \mapsto s_1 h)(X\hat{f}), \ldots, (t_p \mapsto s_p h)(X\hat{f})](\omega g) \mapsto sh)$.

By the mappings $X\hat{f}$ on terms, we immediately have a mapping

$$Y_0\hat{f}^2: E_0 \longrightarrow E_1^0$$

from Ω_0-equations to Ω_1-equations, where Y_0 is the variable set underlying E_0, by applying $Y_0\hat{f}$ to each side of each equation (formally: $E_1^0 := E_0 h(Y_0\hat{f} \times Y_0\hat{f})$).

2.6 Definition: The category spec has specifications $D = \langle S, \Omega, E \rangle$ as objects. Its morphisms $f: D_0 \longrightarrow D_1$ are pairs of mappings $f = (g,h)$, $h: S_0 \longrightarrow S_1$, $g: \bar{\Omega}_0 \longrightarrow \bar{\Omega}_1$, such that

(1) $\Omega_0 h^* = g\,\Omega_1$ and

(2) the equations in E_1^0 are valid in $\underline{init}\ D_1$.

Specification morphisms correspond to simple cases of theory morphisms in [4].

We are now going to describe the connection to graphs. Following [11, 12], we have the following definition.

2.7 Definition: The category graph has graphs as objects. A graph $G = (F, V, \sigma, \tau)$ consists of two sets, F (edges) and V (vertices), and two mappings $\sigma, \tau : F \longrightarrow V$ (source resp. target). Graph morphisms $f: G_0 \longrightarrow G_1$ are pairs of mappings $f = (h,g)$, $h: V_0 \longrightarrow V_1$, $g: F_0 \longrightarrow F_1$, such that

(1) $\sigma_0 h = g \sigma_1$ and (2) $\tau_0 h = g \tau_1$.

In order to associate graphs with specifications, we define a forgetful functor

$$\Gamma : \underline{\text{spec}} \longrightarrow \underline{\text{graph}}$$

as follows. Specifications $D=\langle S, \Omega, E \rangle$ are sent to

$$D\Gamma = (\bar{\Omega}, V, \sigma, \tau) ,$$

where the edges are just the operation symbols in $\bar{\Omega}$, and the vertices are the domains and codomains occurring in Ω, formally:

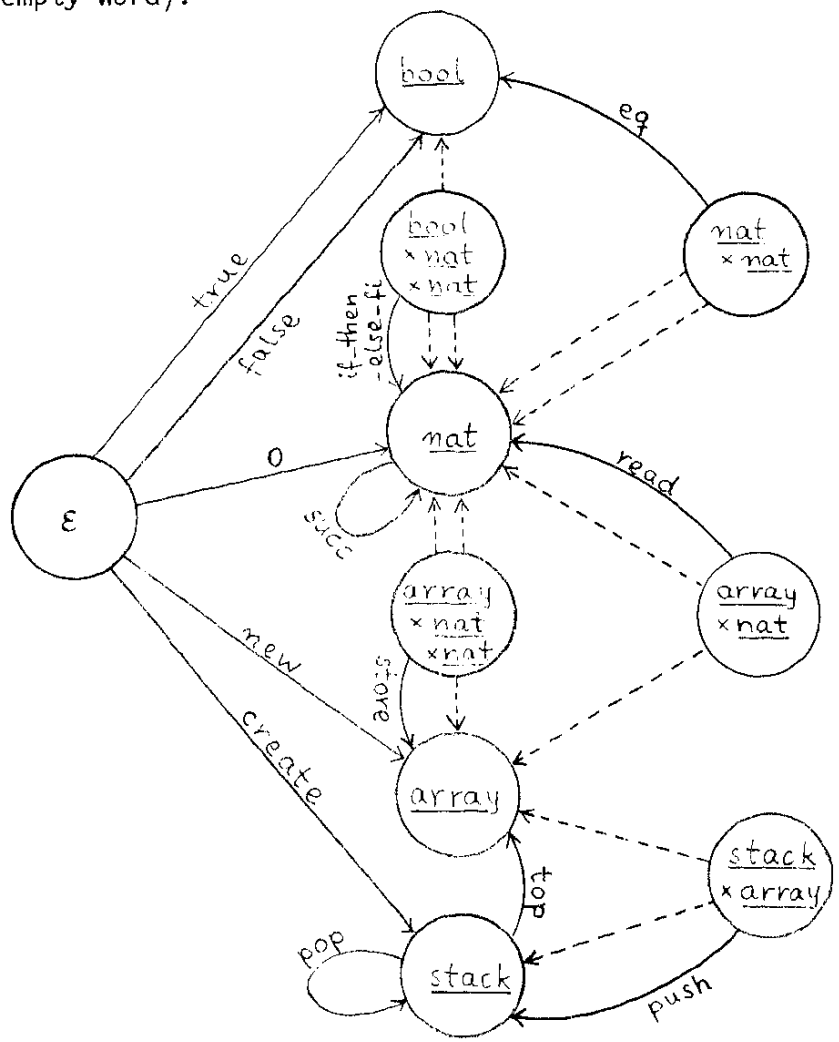
$$V = \{ x,s \mid \exists \omega : (\omega \longmapsto xs) \in \Omega,\ x \in S^*,\ s \in S \} .$$

The source and target functions, σ and τ, are given by

$$\left.\begin{array}{l} \sigma : \omega \longmapsto x \\ \tau : \omega \longmapsto s \end{array}\right\} \text{ if } (\omega \longmapsto xs) \in \Omega .$$

Specification morphisms $f=(h,g): D_0 \longrightarrow D_1$ are sent by Γ to the graph morphisms $f^*=(h^*,g)$. That f^* is graph morphism follows immediately from definition 2.6 (1). It is easily checked that Γ is indeed a functor.

2.8 Example: We show $D_a\Gamma$ for specification D_a of example 2.4 (ε denotes the empty word):

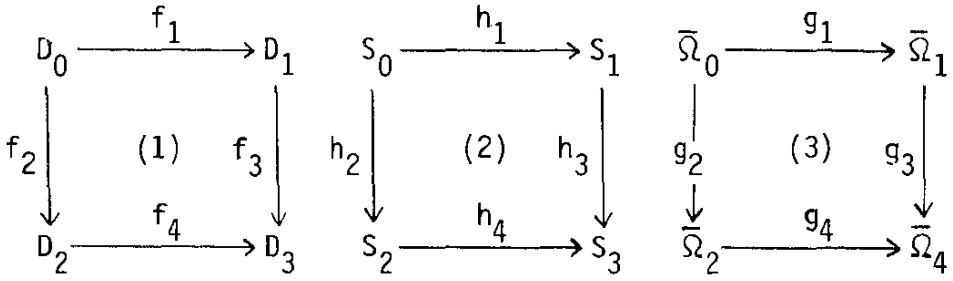


The projections are indicated by broken lines.

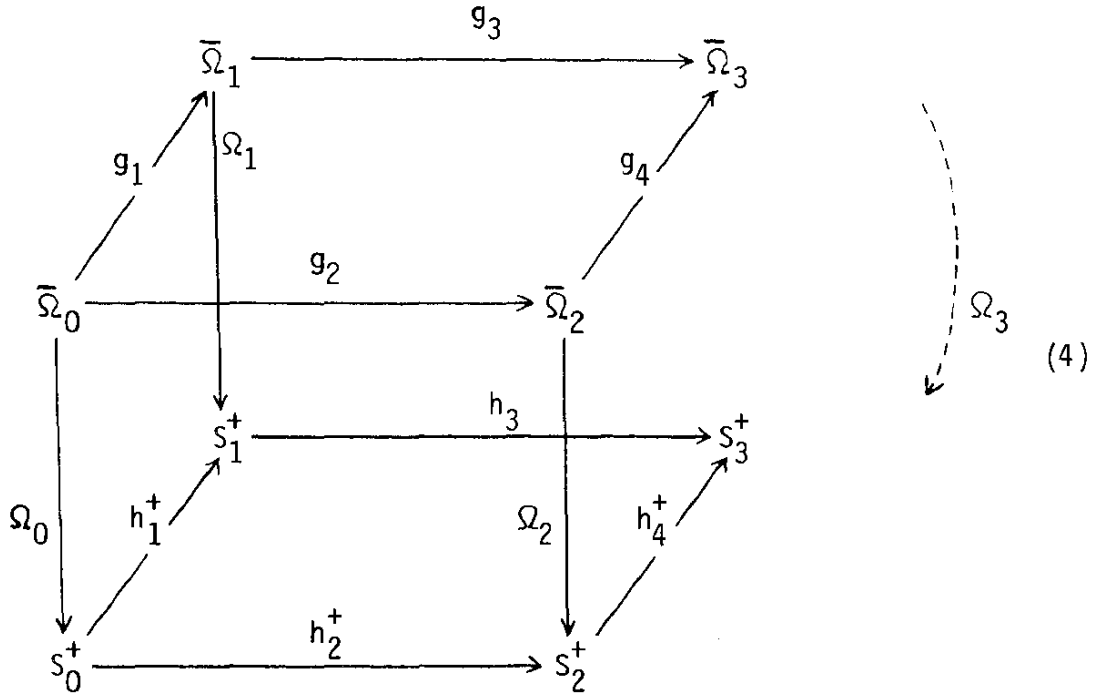
We know that graph has pushouts [11, 12] and how to construct them.

2.9 Theorem: spec has pushouts.

Proof: Let $f_1:D_0 \longrightarrow D_1$ and $f_2:D_0 \longrightarrow D_2$ be given morphisms in spec. We construct D_3, $f_3:D_1 \longrightarrow D_3$ and $f_4:D_2 \longrightarrow D_3$ such that diagram (1) is a pushout. Let $f_i=(h_i,g_i)$ for $1 \leq i \leq 4$, and $D_j=\langle S_j, \Omega_j, E_j\rangle$, $0 \leq j \leq 3$, and let h_3, h_4 and g_3, g_4 be given



such that diagrams (2) and (3) are pushouts in set. Due to the definition of morphisms in spec, we have the following commutative diagram:



Since the upper diagram (3) is a pushout, there is a unique mapping $\Omega_3:\bar{\Omega}_3 \longrightarrow S_3^+$ making diagram (4) a commutative one.

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 must construct equations E_3 such that, for $D_3=\langle S_3, \Omega_3, E_3\rangle$, diagram (1) is a pushout.

Let $E_3^1 = E_1(Y_1 \hat{f}_3^2)$ and $E_3^2 = E_2(Y_2 \hat{f}_4^2)$, where Y_j is the variable set underlying E_j, $j=1,2$. Let

$$E_3 = E_3^1 \cup E_3^2 \quad .$$

Then it is clear that f_3, f_4 are morphisms in spec. To prove that diagram (1)

is a pushout in spec is a straightforward exercise using the fact that diagrams (2) and (3) are pushouts in set. ///

Remark: Evidently $\langle \emptyset, \emptyset, \emptyset \rangle$ is an initial element in spec. Thus, spec has finite colimits. spec can even be shown to be cocomplete.

2.10 Theorem: The forgetful functor Γ respects pushouts.

Proof: The first part of the proof of theorem 2.9 is essentially a pushout construction in graph: the pushout of $f_1^*: D_0\Gamma \to D_1\Gamma$ and $f_2^*: D_0\Gamma \to D_2\Gamma$ is the pair of graph morphisms $f_3^* = (h_3^*, g_3): D_1\Gamma \to D_3\Gamma$, $f_4^* = (h_4^*, g_4): D_2\Gamma \to D_3\Gamma$. ///

Remark: does not detect pushouts: if we add equations to D_3 (cf. diagram (1) above), we may lose the pushout property in spec without affecting the graphs.

3. Parametric specifications

In applications, most specifications refer to other specifications already written or to be inserted later. Often this reference is not completely fixed but lets several possibilities open. So for example, the concept of a stack is rather independent of the type of its elements (cf. example 2.5), as long as there is a constant element serving as the value of top(create). To give another example, the general concept of an array (cf. example 2.4) must allow for arbitrary types for the keys and the entries. The only requirement is that there be an eq relation on keys and a constant entry serving as the value of read(new,n). Thus we would like to give specifications as in the following examples.

3.1 Example: STACK(ELEM) denotes the specification obtained from D_S (cf. example 2.5) by replacing

array	by	elem
new	by	e0

and inserting the operation

e0 : $\longrightarrow$ elem

in the syntax part. We thus have a subspecification

ELEM = $\langle \{ \text{elem} \}, \{ e0 \mapsto \text{elem} \}, \emptyset \rangle$

of STACK(ELEM).

3.2 Example: ARRAY(KEY,ENTRY) denotes the following specification:

new:	$\longrightarrow$ array
store:	array×key×entry $\longrightarrow$ array
read:	array×key $\longrightarrow$ entry
eq:	key × key $\longrightarrow$ bool
e1:	$\longrightarrow$ entry
true,false:	$\longrightarrow$ bool
if-then-else-fi:	bool×entry×entry $\longrightarrow$ entry

---

read(new,n) =e1
read(store(a,k,e),h)=if eq(k,h)' then e else read(a,h) fi

We have a subspecification (KEY,ENTRY) consisting of the sorts key, entry, bool, the operations eq, el, true, false, if-then-else-fi, and no equations. We could add, however, appropriate equations to express that the eq relation on keys should be reflexive, symmetric, and transitive.

The intended meaning of such specifications is that there is a subspecification playing the role of a formal parameter: concrete specifications like those in examples 2.4/5 result if "actual" parameters have been substituted for the formal ones. We are now going to make these concepts precise.

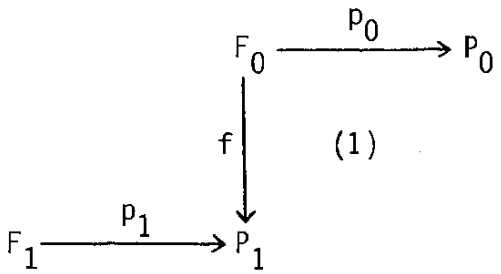
3.3 Definition: A morphism f=(h,g) spec is called embedding iff h and g are injective.

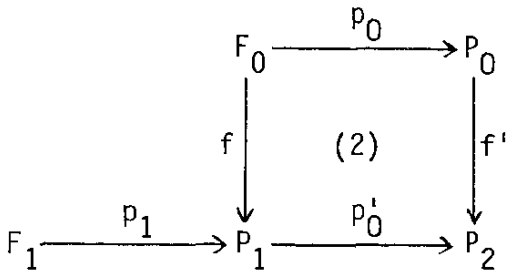
3.4 Definition: A parametric specification p is an embedding $p:F \longrightarrow P \in spec$. F is called the formal parameter part of p.

We identify "concrete" specifications in the sense of definition 2.1 with parametric specifications having an empty formal parameter part, i.e. D with $\langle \emptyset, \emptyset, \emptyset \rangle \rightarrow D$. Definition 3.4 allows for formal sorts, operations, and equations. As indicated in example 3.2, the intended meaning is that only actual parameters satisfying the formal equations can be substituted. Therefore the substitution of an actual parameter shoud be based on a "parameter assignment" that maps formal sorts to actual sorts and formal operations to actual operations such that formal equations carry over to valid equations. This is exactly what morphisms in spec do. Thus, the following definition is immediate.

3.5 Definition: Let $p:F \longrightarrow P$ be a parametric specification. A parameter assignment for p is a morphism $f:F \longrightarrow D$ spec. D is called the actual parameter.

The actual parameter may itself be parametric. For example, it could be useful to have stacks whose elements are arbitrary arrays. Thus, in general we have the situation depicted in diagram (1) below.



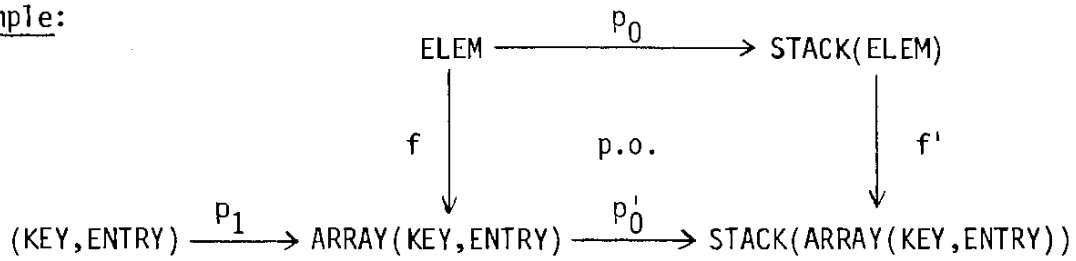


We utilize pushouts in spec in order to define what comes out if p_0 is applied to p_1 by means of f.

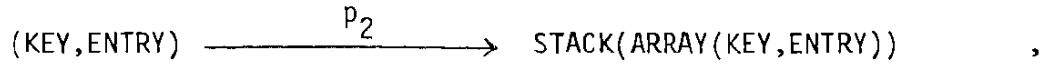
3.6 Definition: By an application of p_0 to p_1 by means of a parameter assignment f we mean the construction of pushout (2) (cf. the diagram above). The result of this application is the parametric specification $p_2=p_1p_0':F_1 \longrightarrow P_2$.

That p_0' and hence p_2 are in fact parametric specifications follows immediately from the construction of pushout in spec and from the fact that injections carry over to opposite sides of pushouts in set.

3.7 Example:



If we have f=(h,g) given by h: elem $\longmapsto$ entry and g: e0 $\longmapsto$ new, we get the result

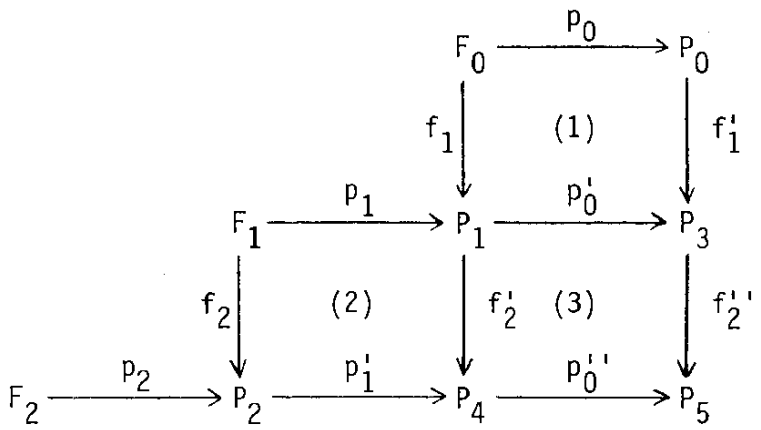


where the latter is obtained from example 3.2 by adding the operations and equations from example 2.5 .

Since actual parameters can be parametric again, parameter substitution can be iterated. We show that the result does not depend on the order of application.

3.8 Theorem: The application of parametric specifications to parametric specifications is associative.

Proof: Let p_i: $F_i \longrightarrow P_i$, i=0, 1, 2 , and f_j:$F_{j-1} \longrightarrow P_j$, j=1, 2 , be given. We then have the following diagram in spec :



We have to show that application of p_0 to p_1 by f_1 and then of p_1p_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_2p_1' by f_1f_2' . This follows from the fact that diagrams (1) and (2,3) are pushouts iff (2) and (1,3) are pushouts. Both properties are equivalent to (1), (2) and (3) being pushouts. The details are left to the reader. ///

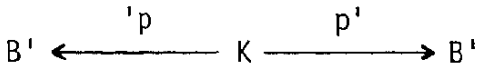
4. Other Applications

We have seen in the last section that pushout constructions in spec can be profitably applied to cope with the problems of parametrization of specifications. Parameter substitution is a special case of a replacement operation, and, in view of theorem 2.10, there is a corresponding graph

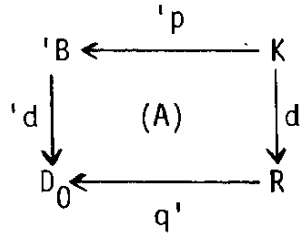
replacement on the syntax graph. Much more general types of graph replacements have been considered in the theory of graph grammars [11, 12]. The question is whether analogous ideas could be applied to define more general "specification replacements", and what the possible applications of such a generalization could be.

To answer the second question first, we can think of situations in the stepwise formulation and edition of specifications, perhaps in a dialog environment, where the need for replacing parts of the current structure arises. Such situations occur naturally when errors must be corrected, and when a better module has to be substituted consistently for an unprofitable one. Another possible application is the updating of specifications as documentations of systems that must be adapted to changed user needs.

The essential tool for developing the machinery of graph replacements by graph productions is the pushout construction in graph. This suggests that analogous replacement operations can be defined on any structures that form a category in which pushouts exist. Since this is the case with spec, we can define a "specification production" as a pair of morphisms

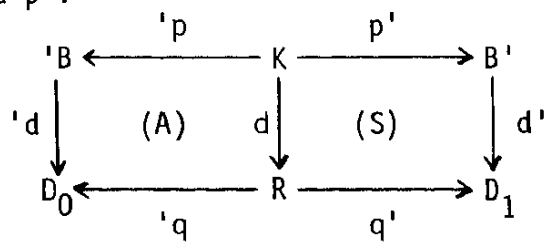


The performance of a replacement by such a production consists of two steps. Given a specification D_0, we first have to look for an "occurrence" of the left hand side 'B of the production in D_0. This means to find an "analysis pushout"



that tells us how D_0 can be built up by means of 'B and the "rest" R, given the "gluing specification" K. There may be a great lot of analysis pushouts for given 'p and D_0.

The second step is deterministic (up to isomorphism): We glue the rest R of D_0 and the right hand side B' of the production together by constructing the pushout of d and p'.



This pushout is called the "synthesis pushout". It gives us the result D_1 of the replacement operation.

This general concept can be restricted in several ways in order to meet the intuition of a given application. In the case of graphs, special conditions are often posed on the morphisms 'd and/or 'q in order to restrict the notion of

"occurrence" of 'B in D_0 to useful (and efficiently computable) cases. Moreover, usually labelled graphs are considered with special rules concerning the labels in order to guarantee that the part of D_1 not affected by the replacement has the same labels as the corresponding part of D_0, i.e. that replacements do not comprise renaming the whole structure.

What restrictions and modifications are useful in the case of specifications is an interesting subject of further study. Here, we restrict ourselves to giving a simple but hopefully illustrating example showing especially some peculiarities about the role equations play for the existence of analysis pushouts.

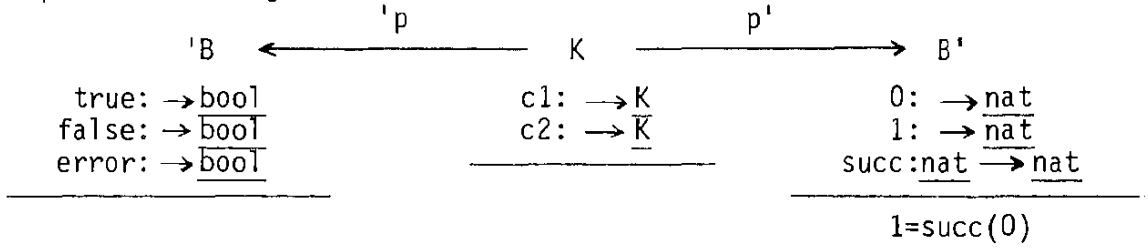
4.1 Example: Let D_b (cf. example 2.2) be enriched by a third constant

$$\text{error:} \longrightarrow \underline{\text{bool}} \quad .$$

We want to solve the following problem: write a specification production that substitutes nat for bool, 1(=succ(0)) for true and 0 for false, and at the same time removes the error constant. The production should be applicable only if the error is "superfluous" in the sense that it does not occur in the equations.

For this example, we want to restrict analysis pushouts (A) to those cases where the left arrow 'd is an inclusion.

The production is given as follows



'p: K ⟼ bool, c1 ⟼ true, c2 ⟼ false

p': K ⟼ nat, c1 ⟼ 1, c2 ⟼ 0

In order to demonstrate an application of this production, let D_{ae} be the specification obtained from D_a (cf. example 2.4) by adding the bool constant error. Then there is an obvious inclusion

$$\text{'d: 'B} \longrightarrow D_{ae}$$

An analysis pushout is now easy to find. The "rest" specification R is identical to D_a from example 2.4, d is given by K ⟼ bool, c1 ⟼ true, c2 ⟼ false, and 'q is the inclusion $D_a \longrightarrow D_{ae}$.

Let D_{an} be the specification resulting from an application of our production with the above analysis. Then, D_{an} is determined uniquely (up to isomorphism). Since D_{ae} did already contain a sort nat and operations 0 and succ, we now have two instances of these items in D_{an}, say nat and nat', 0 and 0', succ and succ' . There is one explicit constant 1 belonging to that nat that has replaced bool.

We can modify our production in that way that there is only one nat in the resulting specification. We simply add D_n (cf. Example 2.2) to 'B in order to find the items of D_n in D_{ae} via 'd . In K we introduce corresponding formal operations, say $\emptyset: \rightarrow \underline{L}$ and $scc: \underline{L} \rightarrow \underline{L}$ that are sent to 'B by 'p as follows: $\underline{L} \mapsto nat$, $\emptyset \mapsto 0$, $scc \mapsto succ$. In the same way, p' must be adapted.

Taking the natural analysis as above, the rest specification R is again D_a. We have, however, sort nat in the range of d, and therefore the synthesis pushout identifies both nat sorts and their operations.

Our modified production is only applicable to specifications containing nat already. This was not the case with our original production that introduced its "own" nat for coding bool.

References

1. ADJ (Goguen,J.A.-Thatcher,J.W.-Wagner,E.G.): An initial algebra approach to the specification, correctness, and implementation of abstract data types. Current trends in programming methodology IV, ed. by R. Yeh, Prentice Hall, New Jersey, 1976

2. ADJ (Thatcher,J.W.-Wagner,E.G.-Wright,J.B.): Data type specification, parametrization, and the power of specification techniques. Proc. SIGACT 10th Annual Symp. Theory Comp., 1978

3. Brendel,W.-Bunke,H.-Nagl,M.: Syntaxgesteuerte Programmierung und incrementelle Compilation. Proc. GI-7.Jahrest., Informatik Fachber. 10, pp 57-73, Springer Berlin 1977

4. Burstall,R.M.-Goguen,J.A.: Putting theories together to make specifications. Proc. 5th IJCAI-77, MIT, Cambridge, Mass. 1977

5. Ehrich,H.-D.: Extensions and implementations of abstract data type specifications. Report No 55/78 Dept. Informatik, Univ. Dortmund 1978

6. Ehrich,H.-D.: On the theory of specification, implementation, and parametrization of abstract data types. In preparation.

7. Goguen,J.A.: Correctness and equivalence of data types. Proc. 1975 Conf. on Algebraic Systems, Lecture Notes in Comp. Sc., Springer-Verlag, Berlin 1975.

8. Guttag,J.V.: The specification and application to programming of abstract data types. Tech. Report CSRG-59, Univ. Toronto 1975

9. Lehmann,D.J.-Smyth,M.B.: Data types. Proc. 10th IEEE Symp. on Foundations of Computing. Providence, R.I., 1977, pp 7-12

10. Liskov,B.H.-Zilles,S.N.: Specification techniques for data abstractions. IEEE Transact. Software Eng., Vol SE-1 (1975) pp 7-19

11. Rosen,B.K.: Deriving graphs from graphs by applying a production. Acta Inf. 4, 337-357 (1975)

12. Schneider,H.J.-Ehrig,H.: Grammars on partial graphs. Acta Inf. 6, 297-316 (1976)

13. Zilles,S.N.: Algebraic specification of data types. MIT Project MAC, Comp. Struct. Group Memo 119, 1975