COMPATIBILITY OF PARAMETER PASSING AND IMPLEMENTATION OF PARAMETERIZED DATA TYPES*

H. EHRIG
Technische Universität Berlin, Fachbereich Informatik (20), Franklinstrasse 28/29,

H.-J. KREOWSKI
Universität Bremen, Fachbereich Mathematik & Informatik, Bibliothekstrasse,

Abstract. The basis for this paper are the concepts of parameterization and implementation of abstract data types which have been developed in the theory of algebraic specifications with initial algebra semantics. In this paper we combine both concepts defining implementations of parameterized data types and studying the compatibility of parameter passing and implementation of parameterized data types. In our main result we show that parameter passing commutes with implementation. This is an important step in order to apply the theory of algebraic specifications to development and stepwise refinement of software systems. We illustrate our notion and results by a small example implementing binary trees over arbitrary data by corresponding strings with brackets. Finally we consider the problem of 2-dimensional compatibility of parameter passing and implementation and discuss the kind of compatibility results which have been shown by other authors in the case of loose and final algebra semantics.

1. Introduction

The theory of algebraic specifications initiated by Zilles [30] is essentially based on the fundamental work of the ADJ-group in [1] and [2]. In [1] the initial algebra approach to the specification and correctness of abstract data types and also a first implementation concept were introduced. The specification concept was extended to parameterized data types in [2]. At the same time an alternative much more syntactic implementation concept was proposed by Guttag [22]. Unfortunately it was not clear at that time whether Guttag's implementation concept should be based on the initial algebra semantics of [1] or the final algebra approach developed by Gianotana et al. [20] and Wand [29].

Our algebraic implementation concept in [15] was on one hand motivated by the examples in [22] and—on the other hand—by the concept of tuple and table constructors used in our algebraic specification schemes for data base systems [17]. Although this concept in [15] already allows parameter parts for actual parameters

* A short version of this paper was presented at ICALP '82 under the title "Parameter Passing Commutes with Implementation of Parameterized Data Types".
in the corresponding specifications, the initial algebra semantics of this concept are not yet suitable for parameterized types with formal parameters. Actually parameterized specifications in the framework of initial algebra semantics need free construction semantics (see [2]), i.e., the parameterized type is freely generated over its parameter part. Taking the initial algebra of a parameterized specification would yield a trivial semantics in most cases. In contrast to actual parameters there are no generating operations for the formal parameter part in general.

The implementation concept for parameterized types in this paper (parameterized implementation, in short) uses essentially the same syntax as in [16] but semantics and the correctness conditions, having a consistency and a completeness part, have to be adapted to the parameterized case. This means that the semantics is given in terms of functors between categories of algebras, the consistency condition becomes a property of functors and operation completeness a proof-theoretical condition based on terms with variables in parameter sorts.

The main idea of studying parameterized implementations—similar to that of parameterized specifications—is to define a family of implementations for all suitable actual parameters in terms of a single ‘higher level implementation’.

Consider the following example: Having shown that sets of natural numbers can be implemented by strings of natural numbers using hash-tables (see [16]), we would also like to be sure to get a correct implementation when natural numbers are replaced by integers. Using our previous concepts we cannot be sure. The obvious idea is to consider implementations of parameterized types like the implementation of sets of data by strings of data using hash-tables. Then the main result of this paper shows that the correctness of such a parameterized implementation implies the correctness of all induced actual implementations, especially that of sets of integers by strings of integers.

The main concept to obtain such results is that of parameter passing for parameterized specifications. This was only touched in [2], but studied in detail on the syntactical level by Ehrig in [9] and concerning semantic and correctness in our joint papers [3, 4] with the ADJ-group. In this paper we are going to apply the concept of parameter passing to parameterized implementations. The main result shows that each correct parameterized implementation and each correct parameter passing procedure replacing formal by actual parameters induces a correct implementation on the level of actual parameters in the sense of [16].

A typical example is the implementation of binary trees \(\text{bintree(data)}\) by strings with brackets \(\text{bracketstring(data)}\), where \(\text{data}\) is the common formal parameter part of both parameterized specifications. Parameter passing means to replace the formal parameter \(\text{data}\) by an actual parameter like integers \(\text{int}\) leading to \(\text{bintree(int)}\) and \(\text{bracketstring(int)}\) respectively. The main result of this paper shows for our example that starting with a correct implementation of \(\text{bintree(data)}\) by \(\text{bracketstring(data)}\) correct parameter passing from \(\text{data}\) to \(\text{int}\) leads to a correct induced implementation of \(\text{bintree(int)}\) by \(\text{bracketstring(int)}\).

The situation is depicted by the following diagram:
In short and more intuitive terms the main result of this paper can be phrased as "Parameter Passing Commutes with Implementation of Parameterized Data Types" which is also the title of our short version [13]. The general idea of such a compatibility is already discussed in [7] within the framework of 2-categories. A system like CAT proposed in [7] seems to be most important for efficient development of software systems. In [11, 16, 18] it is already shown that parameterization and implementation are key concepts in this field.

In Section 2 of this paper we review the concept of parameterized specifications and parameter passing as studied in [4]. For the corresponding algebraic theory of parameterized specifications with requirements (like data constraints or initial restrictions in the sense of [18] and [27]) we refer to [10]. In Section 3 we introduce the main concept of this paper, the implementation of parameterized data types including syntax, semantics and correctness. The main result in Section 4 is the following: Given an implementation \( \text{impl} : \text{spec}(\text{spec}) \rightarrow \text{spec}(\text{spec}') \) of parameterized types and a parameter passing morphism \( \text{h} : \text{spec} \rightarrow \text{spec}' \), then we have an induced implementation \( \text{impl}' : \text{spec}(\text{spec}') \rightarrow \text{spec}(\text{spec}') \). In Theorems 4.3 and 4.4 we show how syntax and semantics of \( \text{impl} \) and \( \text{impl}' \) are related while Theorem 4.5 shows that correctness of \( \text{impl} \) implies correctness of \( \text{impl}' \). These results can be summarized in the conclusion that correct parameter passing commutes with correct implementation.

In Section 5 we sketch how our results can be extended to the general case of implementations with hidden components (see [16]). Moreover, we discuss parameterized parameter passing (see [3, 4]) and formal parameters with constraints like initial restrictions in the sense of Burstall and Goguen [8] or general algebraic constraints in the sense of our new paper [5]. Specifications with constraints are especially important for all the approaches based on loose semantics, like CLEAR in [8], that of Hupbach in [24], and that of Sanella and Wirsing [28], were not only initial resp. freely generated algebras, but the class of all algebras satisfying the constraints is considered as semantics. For the case of final algebra semantics compatibility results for parameter passing and implementation are given in [19]. For ‘module specifications’ corresponding results are sketched by Goguen and Meseguer in [21]. It is discussed how these approaches contribute to the problem of full or general compatibility of parameter passing and implementation, which is referred to as ‘2-dimensional compatibility’ because the situation is similar to the distributive law of a 2-dimensional category (cf. [26]).
At the present time the problem of general compatibility of parameter passing and implementation has been clarified in the literature with respect to a number of different points. But at least in our opinion a full satisfactory solution is yet to be found.

Finally let us make some remarks concerning the exposition of this paper. Assuming that the reader is familiar with basic notions in universal algebra and category theory the paper is self-contained with respect to all definitions of notions and constructions in Computer Science. The corresponding motivation, however, is only briefly sketched here. Most of the motivation is discussed already in detail in our longer versions [16] and [4] concerning implementation and parameter passing respectively.

2. Parameterized types and specifications

We shall assume the algebraic background of [1] or [14, 2] which is based on universal algebra and category theory (see [23, 26]). But we will review the most important notions in connection with this paper. Moreover, we recall the basic algebraic case of parameterized data types and specifications as given in [4].

A abstract data type is regarded as (the isomorphism class of) a many-sorted (heterogeneous) algebra which is minimal, meaning that all data elements are "accessible" using constants and operations of the algebra. A many-sorted algebra consists of an indexed family of sets (called carriers) with an indexed family of operations between those carriers. The indexing system is called a signature and consists of a set $S$ of sorts which indexes the carriers and a family $(\Xi_{w,S})_{w \in S^*}$ of operation names ($\Xi$ is called the operator domain); a symbol $\sigma \in \Xi_{w,S}$ with $w = s_1 \cdots s_m$ names an operation $\sigma_{A,1} : A_{s_1} \times \cdots \times A_{s_m} \to A$ in an algebra $A$ with signature $\Sigma$. The pair $(S, \Sigma)$ determines the category $\text{Alg}_{S, \Sigma}$ of all $S$-sorted $\Sigma$-algebras with $\Sigma$-homomorphisms between them.

A specification, $\text{spec} = (S, \Sigma, E)$, is a triple where $(S, \Sigma)$ is a signature and $E$ a set of equations. $\text{Alg}_{\text{spec}}$ is the category of all $\text{spec}$-algebras, i.e., all $S$-sorted $\Sigma$-algebras satisfying the equations $E$ and all $\Sigma$-homomorphisms between them. When we write the combination $\text{spec}' = \text{spec} + (S', \Sigma', E')$, we mean that $S$ and $S'$ are disjoint, that $\Sigma'$ is an operator domain over $S + S'$ which is disjoint from $\Sigma$, and that $E'$ is a set of equations over the signature $(S + S', \Sigma + \Sigma')$. Constructing such a combination in the sequel means that $+$ is a disjoint union which makes sure (using suitable renamings which are not explicitly mentioned) that sorts and operations in the second component become disjoint from those in the first one.

We follow [1] in saying that the semantics of a specification $\text{spec}$ is the (isomorphism class of the) algebra $T_{\text{spec}}$ which is initial in $\text{Alg}_{\text{spec}}$. $T_{\text{spec}}$ can be constructed as a quotient $T_{\text{spec}} = T_{S, \Sigma}/\equiv_{T}$ of the term algebra $T_{(S, \Sigma)}$ (corresponding to the signature $(S, \Sigma)$) by the congruence generated from the equations $E$.

As an example which will also be used in later sections we specify the maximum of each of two integers using a test whether an integer is negative or not. We use the well-known specifications int of integers and bool of boolean values.
\[ \text{intmax} = \text{int} + \text{bool} \]

\textbf{opns:} \text{IS-NEG} : \text{int} \to \text{bool}
\text{MAX} : \text{int} \to \text{int}
\text{if-then-else} : \text{bool} \text{ int} \to \text{int}

\textbf{eqns:} \text{IS-NEG}(0) = \text{FALSE}
\text{IS-NEG}(\text{PRED}(0)) = \text{TRUE}
\text{IS-NEG}(X + X) = \text{IS-NEG}(X)
\text{IS-NEG}(\text{SUCC}(X + X)) = \text{IS-NEG}(X)

\text{MAX}(0, X) = \text{if IS-NEG}(X) \text{ then } 0 \text{ else } X
\text{MAX}(X, 0) = \text{if IS-NEG}(X) \text{ then } 0 \text{ else } X
\text{MAX}(X, Y) = \text{if IS-NEG}(X)
\quad \text{then (if IS-NEG}(Y) \text{ then } \text{MAX}(X, Y) \text{ else } Y)
\quad \text{else (if IS-NEG}(Y) \text{ then } \text{MAX}(X, Y))
\text{MAX}((\text{SUCC})(X), (\text{SUCC})(Y)) = (\text{SUCC})(\text{MAX}(X, Y))
\text{MAX}((\text{PRED})(X), (\text{PRED})(Y)) = (\text{PRED})(\text{MAX}(X, Y))

\text{if TRUE then } X \text{ else } Y = X
\text{if FALSE then } X \text{ else } Y = Y

intmax extends the specification of integers int (with 0, SUCC, PRED and addition +) and boolean values bool (with constants TRUE and FALSE).

Now let us consider parameterized data types and specifications:

\textbf{2.1. Definition.} A parameterized data type \text{PDAT} = \langle \text{SPEC}, \text{SPEC}_1, T \rangle consists of the following data:

\begin{align*}
\text{PARAMETER DECLARATION} & \quad \text{SPEC} = \langle S, \Sigma, E \rangle \\
\text{TARGET SPECIFICATION} & \quad \text{SPEC}_1 = \text{SPEC} + \langle S^1, \Sigma^1, E^1 \rangle
\end{align*}

and a functor \( T : \text{Alg}_{\text{SPEC}} \to \text{Alg}_{\text{SPEC}_1} \).

PDAT is called \text{persistent} (\text{strongly persistent}) if \( T \) is, i.e. for every \text{SPEC}-algebra \( A \) we have a natural isomorphism \( V(T(A)) \cong A \) (resp. \( V(T(A)) = A \)) where \( V \) is the forgetful functor from \text{SPEC}_1- to \text{SPEC}-algebras (cf. Definition 2.4(3)).

\textbf{2.2. Definition.} A parameterized specification \text{PSPEC} = \langle \text{SPEC}, \text{SPEC}_1 \rangle consists of the following data:

\begin{align*}
\text{PARAMETER DECLARATION} & \quad \text{SPEC} = \langle S, \Sigma, E \rangle \\
\text{TARGET SPECIFICATION} & \quad \text{SPEC}_1 = \text{SPEC} + \langle S^1, \Sigma^1, E^1 \rangle.
\end{align*}

The semantics of the specification is the free construction \( \langle \text{SPEC}, \text{SPEC}_1, F \rangle \), \( F : \text{Alg}_{\text{SPEC}} \to \text{Alg}_{\text{SPEC}_1} \), i.e., the parameterized type \text{PDAT} = \langle \text{SPEC}, \text{SPEC}_1, F \rangle.

PSPEC is called \text{(strongly) persistent} if the free construction \( F \) is \text{(strongly) persistent}, i.e., the unit \( \eta(A) : A \to VF(A) \) for \( A \in \text{Alg}_{\text{SPEC}} \) is a natural isomorphism (identity).
Remark. Note that parameterized specifications are purely syntactical objects which uniquely determine (up to isomorphism) free constructions and hence their associated parameterized data types as semantics. General parameterized data types, on the other hand, are not fully determined by their specifications but by the functor $T$. This may be chosen arbitrarily and is not forced to be free.

For simplicity of presentation we only allow equations in the parameter declaration $\text{SPEC}$. For the case of parameterized specifications with requirements in the sense of [8, 27] we refer to [10, 5].

2.3. Examples. (1) Binary trees $\text{bintree}$ provide a typical example of a parameterized data type (see [4]). A binary tree is generated as a labelled $\text{LEAF}$ (root) with a $\text{LEFT}$ son or a $\text{RIGHT}$ son or $\text{BOTH}$ respectively. The actual labels are not specified, but the formal parameter $\text{data}$ (consisting of one sort $\text{data}$ only) requires some label alphabet. As a sample of retrieval operations we want to measure the heights of the trees. For this computation we use the specification $\text{intmax}$.

\[
\begin{align*}
\text{bintree}(\text{data} + \text{intmax}) &= \\
\text{data} + \text{intmax} &+ \\
\text{sorts:} &\quad \text{bintree} \\
\text{opns:} &\quad \text{LEAF} : \text{data} \to \text{bintree} \\
&\quad \text{LEFT} : \text{bintree \ data} \to \text{bintree} \\
&\quad \text{RIGHT} : \text{data \ bintree} \to \text{bintree} \\
&\quad \text{BOTH} : \text{bintree \ data \ bintree} \to \text{bintree} \\
&\quad \text{HEIGHT} : \text{bintree} \to \text{int} \\
\text{eqns:} &\quad \text{HEIGHT}(\text{LEAF}(x)) = 0 \\
&\quad \text{HEIGHT}(\text{LEFT}(B, x)) = \text{SUC}(\text{HEIGHT}(B)) \\
&\quad \text{HEIGHT}(\text{RIGHT}(x, R')) = \text{SUC}(\text{HEIGHT}(R')) \\
&\quad \text{HEIGHT}(\text{BOTH}(B, x, B')) = \text{SUC}(\text{MAX}(\text{HEIGHT}(B), \text{HEIGHT}(B'))) 
\end{align*}
\]

The parameter $\text{data}$ consists of the sort $\text{data}$ only. Also $\text{intmax}$ is considered as formal parameter because the height can be computed by the given equations relative to an arbitrary $\text{intmax}$-algebra. Especially, this means that $\text{bintree}$ is persistent. A more elaborate specification of binary trees, which also allows to count the number of leaves, edges or nodes respectively, to test ballancedness or degeneracy to traverse binary trees in some orders, can be found in [12].

(2) The parameterized specification $\text{bracketstring}$ has the same formal parameter as $\text{bintree}$. Bracketstrings are generated from the initial symbols by left-addition of $\text{data}$-elements, an opening or closing bracket, or a comma. They can also be built up by right-addition or concatenation. Moreover, the depth of the bracket structure is specified. For each bracketstring, depth returns a sequence of integers
(intseq) representing the increasing and decreasing differences between opening and closing brackets while the string is traversed from left to right. The specification intseq provides all necessary operations (i.e., the single sequence, insertion of integers, access to the maximum entry, increasing and decreasing of each entry of a sequence by one.)

\[
\text{bracketstring(data + intmax)} = \\
data + \text{intseq} + \\
\text{sorts: alphabet, bracketstring}
\]

\[
\text{opns: INCL : data } \rightarrow \text{ alphabet} \\
\text{OPEN : alphabet } \rightarrow \text{ alphabet} \\
\text{CLOSE : alphabet } \rightarrow \text{ alphabet} \\
\text{COMMA : alphabet } \rightarrow \text{ alphabet} \\
\text{INIT : alphabet } \rightarrow \text{ bracketstring} \\
\text{LADD : alphabet bracketstring } \rightarrow \text{ bracketstring} \\
\text{RADD : bracketstring alphabet } \rightarrow \text{ bracketstring} \\
\text{CONCAT : bracketstring bracketstring } \rightarrow \text{ bracketstring} \\
\text{DEPTH : bracketstring } \rightarrow \text{ intseq}
\]

\[
\text{eqns: CONCAT(INIT(a), S)} = \text{LADD}(a, S) \\
\text{CONCAT(LADD(a, S), S′)} = \text{LADD}(a, CONCAT(S, S′))
\]

\[
\text{RADD}(S, a) = \text{CONCAT}(S, \text{INIT}(a))
\]

\[
\text{DEPTH(INIT(INCL(X)))} = \text{SINGLE}(0) \\
\text{DEPTH(INIT(OPEN)))} = \text{SINGLE(SUCC}(0)) \\
\text{DEPTH(INIT(CLOSE))} = \text{SINGLE(PRED}(0)) \\
\text{DEPTH(INIT(COMMA))} = \text{SINGLE}(0) \\
\text{DEPTH(LADD(INCL(X), S))} = \text{DEPTH}(S) \\
\text{DEPTH(LADD(OPEN, S))} = \text{INSERT(SUCC}(0), \text{INCREASE}(\text{DEPTH}(S))) \\
\text{DEPTH(LADD(CLOSE, S))} = \text{INSERT(PRED}(0), \text{DECREASE}(\text{DEPTH}(S))) \\
\text{DEPTH(LADD(COMMA, S))} = \text{DEPTH}(S)
\]

For sake of completeness we also give the specification of intseq used above:

\[
\text{intseq} = \text{intmax} + \\
\text{sorts: intseq}
\]

\[
\text{opns: SINGLE : int } \rightarrow \text{ intseq} \\
\text{INSERT : int intseq } \rightarrow \text{ intseq} \\
\text{MAXIMUM : intseq } \rightarrow \text{ int} \\
\text{INCREASE, DECREASE : intseq } \rightarrow \text{ intseq}
\]

\[
\text{eqns: MAXIMUM(SINGLE(X))} = X \\
\text{MAXIMUM(INSERT(X, S))} = \text{MAX}(X, \text{MAXIMUM}(S))
\]
We now come to the problem of parameter passing. In the basic algebraic case parameter passing morphisms are just specification morphisms allowing to rename sorts and operations of the formal parameter by sorts and operations of the actual parameter in such a way that formal equations are translated into actual ones. A more general version of parameter passing is discussed in Section 5.2. It is well known that each vector space can be considered as an Abelian group by forgetting the outer multiplication and the associated field. In a similar way a specification morphism tells us how each actual-parameter algebra becomes a formal-parameter algebra by renaming some domains and operations and by forgetting the remaining ones. This construction also works for algebra morphisms, such that we obtain a forgetful functor induced by a specification morphism.

2.4. Definition. (1) A specification morphism \( h : (S, \Sigma, E) \rightarrow (S', \Sigma', E') \) consists of a mapping \( h_S : S \rightarrow S' \) and a \((S^* \times S)\)-indexed family of mappings \( h_\Sigma : \Sigma \rightarrow \Sigma' \) with \( h_{2_m, n} : \Sigma_{m,n} \rightarrow \Sigma'_{h_\Sigma(m), h_\Sigma(n)} \) (where \( h_S^* \) is the extension of \( h_S \) to strings of sorts) such that every equation of \( E \), when translated by \( h \), belongs to \( E' \), i.e., \( h(E) \subseteq E' \). The morphism \( h \) is called simple if \( (S, \Sigma, E) \subseteq (S', \Sigma', E') \) and \( h_S, h_\Sigma \) are the inclusions.

(2) The category of all specifications and specification morphisms is called \( \text{Catspec} \).

(3) For each specification morphism \( h : \text{Spec} \rightarrow \text{Spec}' \) there is a functor \( V_h : \text{Alg}_{\text{Spec}} \rightarrow \text{Alg}_{\text{Spec}'} \) called forgetful functor with respect to \( h \) (see [4]) which is defined for each \( A \) in \( \text{Alg}_{\text{Spec}'} \) by

\[
V_h(A)_s = A_{h_S(s)} \quad \text{for } s \in S,
\]

\[
\sigma_{A, h_\Sigma(A)} = h_\Sigma(\sigma)_A \quad \text{for } \sigma \in \Sigma.
\]

In the following we define standard parameter passing as in [4].

2.5. Definition (Standard Parameter Passing). Given a parameterized specification \( \text{pspec} = (\text{spec}, \text{spec}') \), a specification \( \text{spec}' \), called actual parameter, and a specification morphism \( h : \text{spec} \rightarrow \text{spec}' \), called parameter passing morphism, then there is the following parameter passing diagram:

\[
\begin{array}{ccc}
\text{spec} & \xrightarrow{h} & \text{spec}' \\
\downarrow & & \downarrow \\
\text{spec} & \xrightarrow{h} & \text{spec}'
\end{array}
\]
$s$ and $s'$ are simple specification morphisms and $\text{SPEC}'$, called value specification, is given by

$$\text{SPEC}' = \text{SPEC}' + (S1', \Sigma 1', E1')$$

with

$$S1' = S1, \quad \Sigma 1' = h'(\Sigma 1) \quad \text{and} \quad E1' = h'(E1)$$

where $h': \text{SPEC} \to \text{SPEC}'$ is a specification morphism with

$$h'_S(x) = \text{if } x \in S1 \text{ then } x \text{ else } h_S(x), \quad \text{and}$$

$$h'_\Sigma(y) = \text{if } y \in \Sigma 1 \text{ then } y \text{ else } h_\Sigma(y).$$

This mechanism of standard parameter passing is called correct if the following two conditions are satisfied:

1. actual parameter protection, i.e., $V_S(T_{\text{SPEC}'}) = T_{\text{SPEC}'}$.
2. passing compatibility, i.e., $F \circ V_h(T_{\text{SPEC}'}) = V_h(T_{\text{SPEC}'})$.

where $T_{\text{SPEC}'}$ and $T_{\text{SPEC}'}$ are initial algebras and $F$ the semantics of $\text{PSPEC}$ (see Definition 2.2).

**Interpretation.** The value specification $\text{SPEC}'$, also written as $\text{SPEC}'(\text{SPEC}')$, is the result of replacing the formal parameter $\text{SPEC}$ in $\text{SPEC}$, also written as $\text{SPEC}'(\text{SPEC})$, by the actual parameter $\text{SPEC}'$. In terms of category theory the value specification $\text{SPEC}'$ is nothing else but the pushout in the diagram above.

Actual parameter protection means that the actual parameter $\text{SPEC}'$ remains unchanged in the value specification $\text{SPEC}'$. Passing compatibility means that the semantics of parameter passing, especially the transformation from $T_{\text{SPEC}'}$ to $T_{\text{SPEC}'}$, meets the semantics $F$ of $\text{PSPEC}$ (up to renaming with respect to $h$ and $h'$).

The main result for standard parameter passing is the following (see [4, Theorem 5.2]).

2.6. **Theorem** (Correctness of Standard Parameter Passing). Standard parameter passing is correct (with respect to all actual parameters $\text{SPEC}'$ and all parameter passing morphisms $h: \text{SPEC} \to \text{SPEC}'$) if and only if the given parameterized specification $\text{PSPEC} = (\text{SPEC}, \text{SPEC})$ is (strongly) persistent.

This theorem is technically based on the extension Lemma (see [3, 5.1]) which will also be used in our proofs for Section 4.

2.7. **Extension Lemma.** (1) Given a parameterized specification $\text{PSPEC} = (\text{SPEC}, \text{SPEC})$ as in Definition 2.2 and a specification morphism $h: \text{SPEC} \to \text{SPEC}'$, then there is a well-defined parameter passing diagram
as given by Definition 2.5 which is a pushout in the category of specifications and specification morphisms, i.e., we have

(i) \( s' \circ h = h' \circ s \), and

(ii) for all specifications \( \text{SPEC}'' \) and all specification morphisms \( s'' : \text{SPEC}' \rightarrow \text{SPEC}'' \) and \( h'' : \text{SPEC} \rightarrow \text{SPEC}'' \) satisfying \( s'' \circ h = h'' \circ s \) there is a unique specification morphism \( f : \text{SPEC}' \rightarrow \text{SPEC}'' \) such that

\[
\begin{align*}
\text{f} \circ s' &= s'' \\
\text{f} \circ h' &= h''.
\end{align*}
\]

(2) Given a (strongly) persistent parameterized data type \( \text{PDAT} = (\text{PSPEC}, F) \) with \( \text{PSPEC} \) and a specification morphism \( h \) as above, then there is a (strongly) persistent functor \( F' : \text{Alg}_{\text{PSPEC}} \rightarrow \text{Alg}_{\text{SPEC}}, \) called extension of \( F \) via \( (h, s) \), satisfying for all \( A' \in \text{Alg}_{\text{SPEC}} \)

\[ V_h(A') = F(V_h(A')). \]

Moreover \( F'(A') \) is uniquely determined by \( A' \) and \( B = F(V_h(A')) \) in the following sense: For all \( B' \in \text{Alg}_{\text{SPEC}} \) satisfying \( V_h(B') = A' \) and \( V_h(B') = B \) we already have \( B' = F'(A') \).

(3) If in addition \( F \) is free (left adjoint to \( V_h \)), then \( F' \) is also free (left adjoint to \( V_h \)).

2.8. Example. To demonstrate the parameter passing mechanism, we intend to construct binary trees of integers (cf. Example 2.3(1)). The \text{intmax}-part of the formal parameter can be used as actual parameter, and the corresponding parameter passing morphism \( h : \text{data} + \text{intmax} \rightarrow \text{intmax} \) maps the sort \text{data} to \text{int} while the \text{intmax}-part is mapped identically. According to Definition 2.5 the value specification \( \text{bintree}(\text{intmax}) \) is obtained from the parameterized specification \( \text{bintree}(\text{data} + \text{intmax}) \) by erasing the two fragments of text \( \cdot \text{data} + \cdot \) and by replacing all occurrences of the sort \text{data} by the sort \text{int}.

Note that the same parameter passing morphsim also defines an actualization of our second example. This leads to \text{bracketstring}(\text{intmax}) \) where the elements of the strings are integers, opening and closing brackets, and commas.

3. Implementation of parameterized types

Now we are going to define implementations of parameterized types given by parameterized specifications. This extends our implemetation concept for data
type specifications without formal parameters in [15] and [16] to the parameterized case.

The idea of (equational) specification is to design a first solution of a given data processing problem. Hence a specification is said to be correct if it meets the problem exactly.

The aim of programming is somewhat different. It has to produce a solution running on a computer. So it cannot be oriented on the problem only, but also has to make use of the special features of an executing system or of a programming language. For example, many data types, which describe application problems (e.g., indices, catalogues, etc.), behave like subsets or mappings on subsets. Especially the order of inserting pieces of information does not matter— from the point of view of the problem. In a program you will represent such situations by some 'sequential' structures (e.g., strings, arrays, trees, hash-tables etc.) where the order of insertions may be significant. Hence, different data in a program may represent the same object in the problem. But it may also happen that you have data in the program representing nothing with respect to the problem. An obvious case is the use of the standard data type integer (including negative numbers) to solve natural number problems. If you want to compare the meaning of a program with the given problem or its specification, you have to identify multiple representations as well as to remove the junk.

In the semantics of our implementation concept, which is to provide an algebraic version of programming in the sense discussed above, these both steps are called IDENTIFICATION and RESTRICTION respectively. Starting point of an implementation is a specification SPECO, which should solve the given software problem on an abstract level already, but may ignore the concrete possibilities of the computer level or of the intended programming language. Moreover we assume that there is a second specification SPEC1. SPEC1 is assumed to be 'closer' to the computer than SPECO containing those data types which are directly available. A subpart SPEC of SPEC0 may already be included in SPEC1. The remaining sorts S0 and operations S0 of SPEC0 are built up—maybe recursively—from the sorts and operations of SPEC1. For this reason the syntax of our implementation concept consists of sort-implementing operations and operations-implementing equations. Their constructive effect is reflected in the semantics by the SYNTHESIS step.

Writing an implementation means that the sorts and operations of the abstract level SPEC0 are refined in terms of sorts and operations of SPEC1 which is again an algebraic specification. Therefore, the process of implementation can be iterated. In other words, our implementation concept allows and supports stepwise refinement of software development.

A program can be called correct if it fulfills its specification. Because we are dealing with parameterized specifications, the semantics of an implementation is given as a functor. This allows to require in the notion of correctness simply that the semantics of an implementation equals the parameterized data type specified by SPEC0, which is subject to the refinement. Moreover the notion of correctness
includes a more subtle part. The sort-implementing operations provide an explicit data representation for the abstract type \( \text{SPEC}_0 \). But you have to make sure that the abstract operations are completely re-defined on this representation using the operations-implementing equations (otherwise they may generate new data). This requirement will be called OP-completeness.

3.1. General Assumption. We assume that we have the following persistent parameterized specifications \( \text{PSPEC}_0 = (\text{SPEC}, \text{SPEC}_0) \) and \( \text{PSPEC}_1 = (\text{SPEC}_1, \text{SPEC}_1) \) with

\[
\begin{align*}
\text{SPEC} &= (S, \Sigma, E) \quad \text{(parameter declaration)}
\text{SPEC}_0 &= \text{SPEC} + (S_0, \Sigma_0, E_0) \quad \text{(target specification 0)}
\text{SPEC}_1 &= \text{SPEC} + (S_1, \Sigma_1, E_1) \quad \text{(target specification 1)}
\end{align*}
\]

Remark. We assume persistency of \( \text{PSPEC}_0 \) and \( \text{PSPEC}_1 \) because this is necessary and sufficient for correctness of standard parameter passing (see Theorem 2.6).

3.2. Definition (Implementation). An implementation of \( \text{PSPEC}_0 \) by \( \text{PSPEC}_1 \), written \( \text{IMPL} : \text{SPEC}_1 \Rightarrow \text{SPEC}_0 \), is a pair

\[ \text{IMPL} = (\Sigma \text{SORT}, \text{EOP}) \]

of operations \( \Sigma \text{SORT} \), called sort-implementing operations, and equations \( \text{EOP} \), called operations-implementing equations, such that

\[
\begin{align*}
\text{SORTIMPL} &= \text{SPEC} + (S_0, \Sigma \text{SORT}, \emptyset) \quad \text{(sort implementation level)}
\text{OPIMPL} &= \text{SORTIMPL} + (\emptyset, S_0, \text{EOP}) \quad \text{(operation implementation level)}
\text{IDIMPL} &= \text{OPIMPL} + (\emptyset, \emptyset, \text{EOP}) \quad \text{(identification level)}
\end{align*}
\]

are combinations (see the introductory paragraph of Section 2) and for all operations in \( \Sigma \text{SORT} \) the range belongs to \( S_0 \).

The semantics of \( \text{IMPL} \) is the following functor \( \text{SEM}_{\text{IMPL}} : \text{Alg}_{\text{SPEC}} \to \text{Alg}_{\text{SPEC}_0} \) defined as the composition

\[ \text{SEM}_{\text{IMPL}} = \text{Alg}_{\text{SPEC}} \xrightarrow{\text{FREE}} \text{Alg}_{\text{SPEC}_1} \xrightarrow{\text{FREE}_{\text{IMPL}}} \text{Alg}_{\text{IDIMPL}} \xrightarrow{\text{RESTR}} \text{Alg}_{\text{SPEC}_0} \]

where \( \text{FREE} \) and \( \text{FREE}_{\text{IMPL}} \) are the free constructions corresponding to the forgetful functors

\[ V_1 : \text{Alg}_{\text{SPEC}_1} \to \text{Alg}_{\text{SPEC}} \quad \text{and} \quad V_{\text{IMPL}} : \text{Alg}_{\text{IDIMPL}} \to \text{Alg}_{\text{SPEC}_1} \]

respectively. The restriction functor \( \text{RESTR} \) is the composition

\[ \text{RESTR} = \text{Alg}_{\text{IDIMPL}} \xrightarrow{\chi} \text{Alg}_{\text{SPEC}_0} \xrightarrow{\text{REACH}} \text{Alg}_{\text{SPEC}_0} \]

of the forgetful functor \( V : \text{Alg}_{\text{IDIMPL}} \to \text{Alg}_{\text{SPEC}_0} \) and the reachability functor \( \text{REACH} \) which is defined as follows:
Let $\text{FREE}_0: \text{Alg}_{\text{SPEC}} \to \text{Alg}_{\text{SPEC}}$ be the free construction with respect to $V_0: \text{Alg}_{\text{SPEC}} \to \text{Alg}_{\text{SPEC}}$ with counit $\varepsilon$. Then there is a functor $\text{REACH} : \text{Alg}_{\text{SPEC}} \to \text{Alg}_{\text{SPEC}}$ called $\text{REACHABILITY}$, such that $\text{REACH}(A)$ is the image of $\varepsilon(A): \text{FREE}_0 \cdot V_0(A) \to A$.

Remarks. (1) Note that the counit $\varepsilon(A): \text{FREE}_0 \cdot V_0(A) \to A$ evaluates the expressions freely generated over the initial values of $V_0(A)$ (i.e., the $\text{SPEC}$-part of $A$). Hence $\text{REACH}(A)$ returns that part of $A$ which is accessible by $\text{SPEC}$-operations applied to initial values from the actual parameter $V_0(A)$. In other words, we get rid of all junk in $A$ not related to $\text{SPEC}$.

(2) The construction $\text{SEM}_{\text{IMPL}}$ is composed of the semantics $\text{FREE}_1$ of the parameterized specification $\text{PSPEC}_1$, followed by a $\text{SYNTHESIS}$ step (free construction from $\text{Alg}_{\text{SPEC}_1}$ to $\text{Alg}_{\text{OPIMPL}}$), an $\text{IDENTIFICATION}$ step (free construction from $\text{Alg}_{\text{OPIMPL}}$ to $\text{Alg}_{\text{IDIMPL}}$) and the $\text{RESTRICTION}$ step $\text{RESTR}$. Unfortunately, performing $\text{IDENTIFICATION}$ before or after $\text{RESTRICTION}$ leads to different results in general so that one has to handle two kinds of semantics, called IR- and RI-semantics respectively. A detailed discussion of both and of their relationship can be found in [16]. Note that the free construction $\text{FREE}_{\text{IMPL}}$ is the composition of the free constructions $\text{SYNTHESIS}$ and $\text{IDENTIFICATION}$. The semantics corresponds to IR-semantics because we first have the $\text{IDENTIFICATION}$ and then the $\text{RESTRICTION}$ step. This case is easier to handle in the parameterized case than the RI-semantics where $\text{RESTRICTION}$ is followed by $\text{IDENTIFICATION}$ (see Section 5.1 for more details).

(3) In Definition 3.5 we will show how implementations in the parameterized case are related to those in the standard case defined in [16].

(4) For simplicity of presentation we have not introduced the general case including hidden components $\text{HID} = (\text{SHID}, \Sigma_{\text{HID}}, \Sigma_{\text{EID}})$ where hidden sorts, operations and equations can be used in the implementation (see [16, 6.1]). This case will be dealt with in Section 5 (see Section 5.1).

Now we are going to define correctness of implementations where we use the notation $\Sigma(\text{SPEC})$ to denote the operations of $\text{SPEC}$.

Obviously we have to require that the semantics $\text{SEM}_{\text{IMPL}}$ of the implementation yields the semantics $\text{FREE}_0$ of the parameterized specification $\text{PSPEC}_0$ to be implemented.

This property is called IR-correctness. Moreover, the intention of correctness is that each operation call in $\text{SPEC}_0$, i.e., a $\text{SPEC}_0$-term $t$ with variables, can be represented by a 'synthesized' operation call in $\text{SPEC}_1$, i.e., a term $t^*$ with variables in $\text{SORTIMPL}$. Representation means that the term $t$ in $\text{SPEC}_0$ regarded as a term in $\text{OPIMPL}$ is equivalent to the term $t^*$ in $\text{SORTIMPL}$. This property will be referred as OP-completeness.

3.3. Definition (Correctness). An implementation $\text{IMPL} : \text{PSPEC}_1 \Rightarrow \text{PSPEC}_0$ as given
in Definition 3.2 is called
(1) *IR-correct* if we have $\text{SF}_\text{IMPL} \equiv \text{FREE}_0$ where $\text{FREE}_0$ is the semantics of $\text{PSPEC}_0$.

(2) *OP-complete* if for each family of variables $X = (X_s)_{s \in S_0}$ with $X_s = \emptyset$ for $s \in S_0$ and for each term $t \in T_{\text{SPEC}_0}(X)$ there is a term $t^* \in T_{\text{SORTIMPL}}(X')$ such that $t$ is $\text{OPIMPL}$-equivalent to $t^*$, i.e., $t \equiv_{\text{OPIMPL}} t^*$, where $X'_s = X_s$ for $s \in S$ and $X'_s = \emptyset$ otherwise.

3.4. Example. If one pushes trees over to the left, one obtains a well-known linear representation of a tree where the representations of the sons are separated by commas and enclosed by brackets and the root label is added to the left. This is the idea of the following implementation of the binary tree specification using bracketstrings (cf. Examples 2.3):

\[ \text{bracketstring(data + intmax)} \implies \text{bintree(data + intmax)} \]

by

\[ \text{sorts impl opns: c : bracketstring } \rightarrow \text{bintree} \]
\[ \text{impl opns eqns: LEAF}(x) = c(\text{INIT}(\text{INCL}(x))) \]
\[ \text{LEFT}(c(S), x) = c(\text{LADD}(\text{INCL}(x), 1 \text{ADD}(\text{OPEN}, \text{RADD}(S, \text{COMMA}), \text{CLOSE}))) \]
\[ \text{RIGHT}(x, c(S)) = c(1 \text{ADD}(\text{INCL}(x), 1 \text{ADD}(\text{OPEN}, 1 \text{ADD}(\text{COMMA}, \text{RADD}(S, \text{CLOSE})))) \]
\[ \text{BOTH}(c(S), x, c(S')) = c(\text{CONCAT}(1 \text{ADD}(\text{INCL}(x), 1 \text{ADD}(\text{OPEN}, S)), 1 \text{ADD}(\text{COMMA}, \text{RADD}(S', \text{CLOSE})))) \]
\[ \text{HEIGHT}(c(S)) = \text{MAXIMUM}(\text{DEPTH}(S)) \]

**Remark.** In the framework of the general implementation concept with hidden components (see Section 5.1) we would introduce a hidden operation

\[ \text{EXPRESSION : bracketstring data bracketstring } \rightarrow \text{bracketstring} \]

with hidden equation

\[ \text{EXPRESSION}(S, x, S') = \text{CONCAT}(1 \text{ADD}(\text{INCL}(x), 1 \text{ADD}(\text{OPEN}, S)), 1 \text{ADD}(\text{COMMA}, \text{RADD}(S', \text{CLOSE}))) \]

such that the equations for LEFT, RIGHT and BOTH could be replaced by the following ones.

\[ \text{LEFT}(c(S), x) = c(\text{EXPRESSION}(S, x, \text{INIT})) \]
\[ \text{RIGHT}(x, c(S)) = c(\text{EXPRESSION}(\text{INIT}, x, S)) \]
\[ \text{BOTH}(c(S), x, c(S')) = c(\text{EXPRESSION}(S, x, S')) \]

In the implementation above the sort \text{bintree} is a copy of the sort \text{bracketstring}, and the \text{bintree}-operations are derived from \text{bracketstring}-operations so that the implementation is OP-complete. Moreover, one can show that the representations
Compatibility of parameter passing

which are accessible by bintree-operations are those bracketstrings with only positive entries in their depth sequence and with equally many opening and closing brackets. So the RESTRICTION step in the semantics is nontrivial in this case. In contrast to that the IDENTIFICATION step has no semantical effect because the bintree-equations are already satisfied on the operation implementation level in an appropriate way. Finally, IR-correctness can be shown using the correspondence between binary trees and their bracketstring representation as observed above.

In addition to parameterized implementations as given in Definition 3.2 we also have to review the standard case of implementations (see [16]) where formal parameters are replaced by actual ones.

3.5. Definition (Standard Implementation). Given algebraic specifications SPEC0 and SPEC1 as in Assumption 3.1 we consider the specification SPEC as a common actual parameter of SPEC0 and SPEC1. Instead of (strong) persistency of SPEC \( \subseteq \) SPEC0 and SPEC \( \subseteq \) SPEC1 we only require (strong) persistency on the initial algebra \( T_{SPEC} \), i.e.,

\[
V_i \circ FREE_i(T_{SPEC}) = V_i(T_{SPEC_1}) = T_{SPEC} \quad \text{for } i = 0, 1.
\]

(1) A standard implementation of SPEC0 by SPEC1, written IMPL : SPEC1 \( \Rightarrow \) SPEC0, is a pair

\[
IMPL = (\Sigma_{SORT}, EOP)
\]

of operations \( \Sigma_{SORT} \) and equations EOP satisfying syntactical conditions as in Definition 3.2.

(2) The semantical algebra \( S_{IMPL} \), short semantics, of IMPL is defined by

\[
S_{IMPL} = SEM_{IMPL}(T_{SPEC})
\]

where \( SEM_{IMPL} : Alg_{SPEC} \rightarrow Alg_{SPEC_0} \) is the functor given in Definition 3.2.

(3) The standard implementation IMPL : SPEC1 \( \Rightarrow \) SPEC0 is called

(a) IR-correct if \( S_{IMPL} \equiv T_{SPEC_0} \),

(b) OP-complete if for each term \( t \) in \( T_{\Sigma(SPEC_0)} \), there is a term \( t^* \in T_{\Sigma(SORTIMPL)} \) such that \( t \) is OPTIMPL-equivalent to \( t^* \), i.e., \( t \equiv_{OPTIMPL} t^* \).

Remark. The formulation of the semantics given above is slightly different but equivalent to the IR-semantical construction in [16, 5.7 and 4.2] where

\[
S_{IMPL} = RESTRICTION \circ IDENTIFICATION \circ SYNTHESIS(T_{SPEC_1}).
\]

In Definition 3.2 we have given

\[
SEM_{IMPL} = REST \circ FREEIMPL \circ FREE1
\]

where

\[
FREEIMPL = IDENTIFICATION \circ SYNTHESIS, \quad FREE1(T_{SPEC}) = T_{SPEC1}
\]
and

\[ \text{RESTRICTION}(T_{\text{IDIMPL}}) = \text{RESTRICTION}(T_{\text{IDIMPL}}) = \text{image}(\text{eval}: T_\mathcal{F} \rightarrow V(T_{\text{IDIMPL}})). \]

This implies \( S_{\text{IMPL}} = \text{SEM}_{\text{IMPL}}(T_{\text{SPEC}}) \) which is used as definition for the standard semantics above. IR-correctness in the parameterized case implies 
\( \text{SEM}_{\text{IMPL}}(T_{\text{SPEC}}) \equiv \text{FREE}(T_{\text{SPEC}}) = T_{\text{SPEC}}, \)
which means IR-correctness in the standard case. Finally OP-completeness in the parameterized case with \( X = 0 \) corresponds exactly to OP-completeness in the standard case above.

### 4. The main results

In this section we give the main results concerning parameter passing for implementations of parameterized specifications. First we give an explicit construction for induced implementations which are standard implementations in the sense of Definition 3.5.

#### 4.1. Definition (Induced Implementation)

Given an implementation \( \text{IMPL} = (\Sigma \text{SORT}, \text{EOP}) \) of \( \text{SPEC} \) by \( \text{SPEC'} \) as given in Definition 3.2, a parameter passing morphism \( h: \text{SPEC} \rightarrow \text{SPEC'} \) and the corresponding value specifications \( \text{SPEC}_v' \) and \( \text{SPEC}_{v'} \) with \( h_0: \text{SPEC}_v \rightarrow \text{SPEC}_v' \) and \( h_1: \text{SPEC} \rightarrow \text{SPEC}' \) as given in Definitions 2.4 and 2.5. Now let

\[ \Sigma \text{SORT}' = \{ h_2(\sigma)/\sigma \in \Sigma \text{SORT} \} \quad \text{and} \quad \text{EOP}' = \{ h_2(v)/v \in \text{EOP} \} \]

where \( h_2(\sigma : x_1 \cdots x_n \rightarrow x) = \sigma : \tilde{x}_1 \cdots \tilde{x}_n \rightarrow \tilde{x} \) with

\[ \tilde{x}_i = \begin{cases} x_i & \text{if } x_i \in \text{SORT} \\ \text{else } h_1(x_i) & \text{for } i = 1, \ldots, n. \end{cases} \]

and \( h_2(e) \) is obtained from \( e \) by replacing each \( \sigma \) by \( h(\sigma) \) and each variable for a sort \( s \in \text{SORT} \) by a corresponding variable for the sort \( h(s) \in \text{SORT}' \).

Then

\[ \text{IMPL}' = (\Sigma \text{SORT}', \text{EOP}') \]

is called ***induced implementation*** \( \text{IMPL}' \) of \( \text{SPEC}' \) by \( \text{SPEC}', \) written

\[ \text{IMPL}' : \text{SPEC} \rightarrow \text{SPEC}' \]

It will be shown in Theorem 4.3 that the induced implementation \( \text{IMPL}' \) is in fact an implementation of \( \text{SPEC}' \) by \( \text{SPEC}' \) in the standard sense given in Definition 3.5. The semantics of \( \text{IMPL}' \) is essentially determined by that of \( \text{IMPL} \) (Theorem 4.4). In Theorem 4.5 we will show that correctness of \( \text{IMPL}' \) implies that of \( \text{IMPL} \).

#### 4.2. Example

The implementation of \( \text{bintree(data+intmax)} \) by \( \text{bracketstring(data+intmax)} \) in Example 3.4 induces an implementation of \( \text{bintree(intmax)} \) by
bracketstring(intmax) (see Example 2.8). This induced implementation is obtained from the given one by replacing the parameterized specifications bintree and bracketstring by the value specifications and by interpreting the variable x as integer variable.

4.3. Theorem (Syntax of Induced Implementations). Given an implementation \( IMPL : PSPEC \Rightarrow PSPEC_0 \) of parameterized specifications and a parameter passing morphism \( h : SPEC \rightarrow SPEC' \), then the induced implementation \( IMPL' : SPEC' \Rightarrow SPEC_0' \) is an implementation in the standard sense where \( SORTIMPL', OPIMPL' \) and \( IDIMPL' \) can be characterized to be pushouts in \( CATSPEC \) (see Definition 2.5) in the following diagrams, where the horizontal morphisms are inclusions and the vertical ones are induced by \( h \):

\[
\begin{array}{cccccc}
SPEC & \longrightarrow & SPEC' & \longrightarrow & SORTIMPL & \longrightarrow & OPIMPL & \longrightarrow & IDIMPL \\
\downarrow h & \quad & \downarrow h_1 & \quad & \downarrow h_2 & \quad & \downarrow h_3 & \quad & \downarrow h_4 \\
SPEC & \longrightarrow & SPEC' & \longrightarrow & SORTIMPL' & \longrightarrow & OPIMPL' & \longrightarrow & IDIMPL'
\end{array}
\]

Remark. Note that \( SPEC_0' \) and \( SPEC' \) are the value specifications of \( PSPEC_0 \) and \( PSPEC \) respectively (see Definition 2.5), and we have by definition of standard implementation

\[
\begin{align*}
SORTIMPL' &= SPEC' + (S0', \Sigma_{SORT'}, \emptyset) \\
OPIMPL' &= SORTIMPL' + (0, \Sigma'0', EOP') \\
IDIMPL' &= OPIMPL' + (0, 0, E0')
\end{align*}
\]

Proofidea. Let \( SPEC', SORTIMPL', OPIMPL' \) and \( IDIMPL' \) be pushouts in diagrams (1)–(4) respectively. Then the explicit constructions due to Definition 2.5 coincide with that in Definition 4.1. The remaining syntactical properties are easy to check.

4.4. Theorem (Semantics of Induced Implementations). Given an implementation \( IMPL : PSPEC \Rightarrow PSPEC_0 \) with semantics \( SEMIMPL : AlgSPEC \rightarrow AlgSPEC_0 \), a parameter passing morphism \( h : SPEC \rightarrow SPEC' \) and let \( IMPL' : SPEC' \Rightarrow SPEC_0' \) be the induced implementation with semantics \( S_{IMPL} \). Furthermore assume that the semantics \( SEMIMPL \) is persistent with respect to the forgetful functor \( V0 : AlgSPEC_0 \rightarrow AlgSPEC \). Then the semantics \( S_{IMPL} \) is uniquely defined by the following properties:

1. \( V0'(S_{IMPL}) = T_{SPEC}' \)
2. \( Vh(S_{IMPL}) = SEMIMPL \circ Vh(T_{SPEC}) \)

where \( V0', Vh \) and \( Vh \) are forgetful functors. \( T_{SPEC} \) the initial \( SPEC' \)-algebra and \( S_{IMPL} = SEMIMPL(T_{SPEC}) \).
Remark. Similar to Theorem 2.6 the persistency of \( \text{SEM}_{\text{IMPL}} \) is necessary and sufficient if properties (1) and (2) are required for all actual parameters \( \text{SPEC}' \) and all parameter passing morphisms \( h : \text{SPEC} \rightarrow \text{SPEC}' \).

Proof. The proof is based on the Extension Lemma (see 2.7) and three additional lemmas. Since \( \text{SEM}_{\text{IMPL}} \) is persistent, the Extension Lemma implies that there is a unique persistent extension \( F' : \text{Alg}_{\text{SPEC}} \rightarrow \text{Alg}_{\text{SPEC}'} \) of \( \text{SEM}_{\text{IMPL}} \) such that properties (1) and (2) are satisfied with \( \text{SEM}_{\text{IMPL}} \) replaced by \( F'(T_{\text{SPEC}}) \). It remains to show that

\[
F' = \text{SEM}_{\text{IMPL}}
\]

where, similar to Definition 3.2, \( \text{SEM}_{\text{IMPL}} \) is given by \( \text{SEM}_{\text{IMPL}} = \text{REST}^{\circ} \circ \text{FREE} \circ \text{FREE}' \). This follows from the Extension Lemma if we have the following properties:

1. \( V^0 \circ \text{SEM}_{\text{IMPL}} = \text{id} \circ \text{Alg}_{\text{SPEC}} \).
2. \( V_{h0} \circ \text{SEM}_{\text{IMPL}} = \text{SEM}_{\text{IMPL}} \circ V_h \).

To show this, we need the following three lemmas which will be proved below.

**Lemma 1.** \( V^0 \circ \text{REST} = V^4 \) where \( V^0 \) and \( V^4 \) are forgetful functors corresponding to \( \text{SPEC} \in \text{SPEC} \) and \( \text{SPEC} \in \text{IDM}_{\text{IMPL}} \). Analogously, \( V^0 \circ \text{REST}' = V^4' \).

**Lemma 2.** \( \text{SEM}_{\text{IMPL}} \) is persistent iff \( \text{FREE} \circ \text{FREE}' \) is persistent.

**Lemma 3.** \( \text{REST} \circ V_{h4} = V_{h0} \circ \text{REST}' \).
Lemma 2 implies by assumption that $\text{FREEIMPL} \circ \text{FREEI}$ is persistent. Now we are able to prove properties (3) and (4):

$$V0' \circ \text{SEMIMPL} = V0' \circ \text{RESTR}' \circ \text{FREEIMPL}' \circ \text{FREEI}'$$

$$= V4' \circ \text{FREEIMPL}' \circ \text{FREEI}' \quad \text{(Lemma 1)}$$

$$= id_{\text{AlgSPEC}} \quad \text{(by the Extension Lemma and the persistence of \text{FREEIMPL} \circ \text{FREEI})}.$$ 

$$V_{h0} \circ \text{SEMIMPL} = V_{h0} \circ \text{RESTR}' \circ \text{FREEIMPL}' \circ \text{FREEI}'$$

$$= \text{RESTR} \circ V_{h4} \circ \text{FREEIMPL}' \circ \text{FREEI}' \quad \text{(Lemma 3)}$$

$$= \text{RESTR} \circ \text{FREEIMPL} \circ \text{FREEI} \circ V_h \quad \text{(Extension Lemma)}$$

$$= \text{SEMIMPL} \circ V_h.$$ 

As mentioned above, properties (3) and (4) imply $F' = \text{SEMIMPL}$ which was left to prove the theorem.

It remains to prove the lemmas above.

**Proof of Lemma 1.** Consider the following diagram:

![Diagram](attachment:image.png)

Subdiagrams (1) and (2) commute by definition of $\text{RESTR}$ and forgetful functors. In order to prove the lemma it remains to show commutativity of (3).

By definition of $\text{REACH}(A)$ for $A \in \text{AlgSPEC}$ we have

$$\text{FREE} \circ V(0)(A) \xrightarrow{f(A)} A$$

where $\rightarrow$ stands for a surjective homomorphism and $\hookrightarrow$ for an inclusion. Hence we also have
By definition of $\varepsilon(A)$ we have $V_0(\varepsilon(A)) \circ \eta(\varepsilon(A)) = \text{id}_{V_0(A)}$ and $\eta$ is identity such that also $V_0(\varepsilon(A)) = \text{id}_{V_0(A)}$. But this implies $V_0 \circ \text{REACH}(A) = V_0(A)$ and hence subdiagram (3).

**Proof of Lemma 2.** Let $F = \text{FREEIMPL} \circ \text{FREEI};$ then we have $\text{SEMIMPL} = \text{RESTR} \circ F$. Persistency of $F$ means $V_4 \circ F = \text{id}_{\text{AlgSpec}}$, and that of $\text{SEMIMPL}$ means $V_4 \circ \text{RESTR} \circ F = \text{id}_{\text{AlgSpec}}$. But we have $V_4 \circ \text{RESTR} = V_4$ by Lemma 1.

**Proof of Lemma 3.** Using $V_{h0} \circ V' = V \circ V_{h4},$ $\text{RESTR} = \text{REACH} \circ V$ and $\text{RESTR}' = \text{REACH}' \circ V'$ it suffices to show that $\text{REACH} \circ V_{h0} = V_{h0} \circ \text{REACH}'$. We will verify the equality $\text{REACH} \circ V_{h0}(A') = V_{h0} \circ \text{REACH}'(A')$ for all $A' \in \text{AlgSpec}_v$. The corresponding property for morphisms follows in a similar way.

From the Extension Lemma we know the commutativity of

By definition of $\text{REACH}'$ we have for each $A' \in \text{AlgSpec}_v$

Applying $V_{h0}$ and using $V_{h0} \circ \text{FREEo} = \text{FREEo} \circ V_h$ we obtain
Let $A = V_{h0}(A')$; then it suffices to show that $V_{h0}(\varepsilon'(A')) = \varepsilon(A)$: Since $\text{REACH}(A)$ is by definition the image of $\varepsilon(A)$ uniqueness of image factorization of $V_{h0}(\varepsilon'(A')) = \varepsilon(A)$ would imply the desired property $\text{REACH} \circ V_{h0}(\varepsilon'(A')) = \text{REACH}(A)$ by uniqueness of $\varepsilon(A)$ satisfying $V0(\varepsilon(A)) \circ \eta(V0(A)) = \text{id}_{V0(A)}$ (where $\eta(V0(A)) = \text{id}_{V0(A)}$, by strong persistency of $\text{FREE0}$). Because strong persistency of $\text{FREE0}$ implies

$$V0 \circ V_{h0}(\varepsilon'(A')) = V_{h0} \circ V0(\varepsilon'(A')) = V_{h0}(\text{id}_{V0(A)}) = \text{id}_{V0(A)}.$$

This completes the proof of Lemma 3 and Theorem 4.4. □

In our last result we show that correctness of the parametrized implementation implies correctness of the induced standard implementation. The part of the proof concerning OP-completeness is closely related to the proof in [19] showing that sufficient completeness in the parameterized case is preserved by pushouts. It is remarkable that the corresponding property does not hold in the unparameterized case.

4.5. Theorem (Correctness of Semantics). Given an implementation $\text{IMPL} : \text{PSPE} \Rightarrow \text{PSPE}$ with induced implementation $\text{IMPL}' : \text{SPEC'} \Rightarrow \text{SPEC'0}$ as in Theorems 4.3 and 4.4, then we have

1. $\text{IMPL}$ IR-correct implies $\text{IMPL}'$ IR-correct,
2. $\text{IMPL}$ OP-complete implies $\text{IMPL}'$ OP-complete.

Proof. (1) $\text{IMPL}$ IR-correct means $\text{SEM}_{\text{IMPL}} = \text{FREE0}$. Since $\text{FREE0}$ is persistent by General Assumption 3.1, we also have persistency of $\text{SEM}_{\text{IMPL}}$. Hence we are able to use the proof of Theorem 4.4 with $\text{SEM}_{\text{IMPL}} = \text{FREE0}$ showing that $\text{SEM}_{\text{IMPL}}$ is the unique extension of $\text{FREE0}$. On the other hand the Extension Lemma implies that the unique extension of $\text{FREE0}$ is the free construction $\text{FREE0}' : \text{Alg}_{\text{SPEC'}} \rightarrow \text{Alg}_{\text{SPEC'0}}$ such that we have $\text{SEM}_{\text{IMPL}} = \text{FREE0}'$. Moreover, we have $\text{FREE0}'(T_{\text{SPEC'}}) \cong T_{\text{SPEC'0}}$. This implies $S_{\text{IMPL}} : T_{\text{SPEC'}}$ which means IR-correctness of $\text{IMPL}'$.

(2) We have to show that for each $\tilde{i} \in T_{\Sigma(\text{SORTIMPL})}$ there is a $\tilde{i}^* \in T_{\Sigma(\text{SORTIMPL})}$ which is $\text{OPIMPL}$-equivalent to $\tilde{i}$. This can be shown by induction on the size of $\tilde{i}$. For size $(\tilde{i}) = 1$ we have $\tilde{i} \in \Sigma'(\text{SORTIMPL}) = \Sigma + \Sigma 0'$. In the case $\tilde{i} \in \Sigma' \subseteq \Sigma(\text{SORTIMPL})$ we can take $\tilde{i}^* = \tilde{i}$. Otherwise, we have $\tilde{i} \in \Sigma 0'$ and hence also $t \in \Sigma 0$ with $h0(t) = \tilde{i}$.

OP-completeness of $\text{IMPL}$ implies that there is a $t^* \in T_{\Sigma(\text{SORTIMPL})}$ which is $\text{OPIMPL}$-equivalent to $t$. Taking $\tilde{i}^* = h2(t^*)$ we have

$$\tilde{i} = h0(t) = h3(t) \equiv \text{OPIMPL}. h3(t^*) = h2(t^*) = \tilde{i}^* \in T_{\Sigma(\text{SORTIMPL})}.$$

This completes the proof for size $(\tilde{i}) = 1$.

For size $(\tilde{i}) = N > 1$ we again consider two cases. If the root of $\tilde{i}$ belongs to $\Sigma'$, the corresponding term $i^*$ can be obtained by applying the induction hypothesis to the arguments of the root in $\tilde{i}$. If the root $\tilde{o}$ of $\tilde{i}$ belongs to $\Sigma 0'$, we have $\sigma \in \Sigma 0$
with \( h_0(\sigma) = \sigma \). Now we consider all proper maximal subterms of \( \bar{t} \) with sorts in \( S' \) (not in \( S_0' \)), say \( \bar{t}_1, \ldots, \bar{t}_m \in T_{X(SPEC')} \) with sorts \( s_1, \ldots, s_m \in S' \).

Let \( x_1, \ldots, x_m \) be pairwise different variables of sort \( s_1, \ldots, s_m \) and \( \bar{X} = \{ x_1, \ldots, x_m \} \). Then there is a \( \bar{t}_0 \in T_{X(SPEC')} \) and an assignment \( \text{ass}: \bar{X} \rightarrow T_{X(SPEC')} \) defined by \( \text{ass}(x_i) = \bar{t}_i \) for \( i = 1, \ldots, m \) with \( \text{ass}(\bar{t}_0) = \bar{t} \), where \( \text{ass} \) is the extension of \( \text{ass} \). By choice of \( \bar{t}_1, \ldots, \bar{t}_m \) we also have \( \bar{t}_0 \in T_{X(SPEC')} \) with \( h_0(\bar{t}_0) = \bar{t}_0 \) and \( X_s = \bar{X}_{(h s)} \) for \( s \in S + S_0 \). Now we use \( \Omega \)-completeness of \( \text{impl} \) to obtain \( \bar{t}_0^* \in T_{X(SORT\text{impl})} \) which is \( \text{opimpl} \)-equivalent to \( \bar{t}_0 \). Hence also \( \bar{t}_0 - h_0(\bar{t}_0) \) is \( \text{opimpl} \)-equivalent to \( \bar{t}_0 - h_2(\bar{t}_0^*) \).

Now we use induction hypothesis to find \( \bar{t}_1^*, \ldots, \bar{t}_m^* \in T_{X(SORT\text{impl})} \) which are \( \text{opimpl} \)-equivalent to \( \bar{t}_1, \ldots, \bar{t}_m \in T_{X(SPEC')} \) respectively. Define a new assignment \( \text{ass}^* \) by \( \text{ass}^*(x_i) = \bar{t}_i^* \) for \( i = 1, \ldots, m \) and let \( \bar{t}^* = \bar{\text{ass}^*}(\bar{t}_0^*) \in T_{X(SORT\text{impl})} \). Then we have

\[
\bar{t} = \text{ass}(\bar{t}_0) = \text{opimpl} \bar{\text{ass}}(\bar{t}_0) = \text{opimpl} \bar{\text{ass}}(\bar{t}_0^*) = \bar{t}^*
\]

because of \( \text{ass}(x_i) = \bar{t}_i \equiv \text{opimpl} \bar{\text{ass}}(\bar{t}_0) = \bar{\text{ass}}(x_i) \) for \( i = 1, \ldots, m \) and \( \bar{t}_0 = \text{opimpl} \bar{t}_0^* \). This completes the proof. \( \square \)

From Theorems 4.3, 4.4 and 4.5 we conclude the following.

### 4.6. Conclusion
(Commutativity of Parameter Passing with Implementation),
Correct parameter passing commutes with correct implementation, i.e., if \( \text{impl} \circ \text{pspec} \Rightarrow \text{pspec} \) is correct and \( h : \text{spec} \rightarrow \text{spec}' \) a parameter passing morphism, then we have the following commutative diagram of correct implementation and parameter passing steps:

\[
\begin{array}{ccc}
\text{SPEC} \circ \text{SPEC} & \xrightarrow{\text{PARAMETERIZED IMPLEMENTATION impl}} & \text{SPEC} \circ \text{SPEC} \\
\text{PARAMETER PASSING} & & \text{PARAMETER PASSING} \\
\text{SPEC} \circ \text{SPEC} & \xrightarrow{\text{INDUCED IMPLEMENTATION impl}} & \text{SPEC} \circ \text{SPEC} \\
\end{array}
\]

Although this diagram is not a diagram in any category introduced so far, speaking of commutativity of this diagram we mean that any parametrized implementation followed by a parameter passing step has the same effect (value specification) as first parameter passing followed by the induced standard implementation.

### 5. Further development and conclusion

In this final section we discuss a number of possible modifications with respect to the concepts of implementation, parameterizations and parameter passing studied...
in Sections 3 and 4. Moreover, we briefly sketch other approaches based on loose resp. final algebra semantics. Finally we discuss the problem of 2-dimensional compatibility of implementation and parameter passing showing that the results in this and other papers provide important steps but not yet the full solution of the problem.

5.1. Modifications of the implementation concept

There are several ways in which the parameterized implementation concept given in Section 3 could be modified. Most of the modifications we are going to consider are already studied for the case of implementations without formal parameters in [16]. In Definition 3.5 such implementations are called standard implementations. This coincides with the corresponding terminology in [16]. Actually there is also an extension of the standard case concerning implementations with hidden components which is discussed in [16, Section 6]. Hidden components, including hidden sorts, hidden operations and hidden equations, not only allow more flexibility in the design of implementation but are also necessary in order to have composition of implementations. Say, if \texttt{set(int)} is implemented by \texttt{bintree(int)} and \texttt{bintree(int)} by \texttt{bracketstring(int)} the composition would become an implementation of \texttt{set(int)} by \texttt{bracketstring(int)} including parts of the specification \texttt{bintree(int)} as hidden parts. In order to allow the corresponding composition for parameterized implementations, i.e., for formal parameter data instead of actual parameter \texttt{int}, we consider the following generalization of Definition 3.2.

An implementation with hidden components of \texttt{pspec0} by \texttt{pspec1}, also written \texttt{impl : pspec1 \Rightarrow pspec0}, is a triple

\[
\text{impl} = (\Sigma \text{sort}, \Sigma \text{op}, \Sigma \text{hid})
\]

of operations \(\Sigma \text{sort}\), equations \(\Sigma \text{op}\) and hidden part

\[
\Sigma \text{hid} = (\Sigma \text{hid}, \Sigma \text{hid}, \Sigma \text{hid}).
\]

The hidden part consists of hidden sorts \(\Sigma \text{hid}\), hidden operations \(\Sigma \text{hid}\) and hidden equations \(\Sigma \text{hid}\) such that

\[
\begin{align*}
\text{sort}\text{impl} &= \text{spec1} + (\Sigma 0 + \Sigma \text{hid}, \Sigma \text{sort}, 0) \\
\text{op}\text{impl} &= (\text{sort}\text{impl} + (0, \Sigma \text{hid}, 0)) + (0, \Sigma 0, \Sigma \text{op}) \\
\text{hid}\text{impl} &= \text{op}\text{impl} + (0, 0, \Sigma \text{hid})
\end{align*}
\]

are combinations and for all operations in \(\Sigma \text{sort}\) the range belongs to \(\Sigma 0 + \Sigma \text{hid}\).

The semantics and the correctness conditions for implementations with hidden components are exactly those given in Definitions 3.2 and 3.3 which, however, are now based on the new definitions for \texttt{sort}\text{impl}, \texttt{op}\text{impl} and \texttt{hid}\text{impl} given above.
All our results in Section 4 can be extended without additional problems to implementations with hidden components. Another kind of change, however, seems to be more difficult: Our semantics in this paper corresponds to IR-semantics (first IDENTIFICATION and then RESTRICTION) while in [16] we also have studied RI-semantics (first RESTRICTION and then IDENTIFICATION). It is no problem to extend the RI-semantic definition to parameterized implementations but we have not succeeded yet to extend the proofs in Section 4 to RI-semantics. Problems seem to be much easier using a restricted implementation concept as in [28] where the complete SYNTHESIS-part does not belong to the implementation concept. Sanella and Wirsing have shown [28] for the case of loose semantics (see Section 5.3) how to combine such a restricted implementation concept with a separate extension step in order to simulate implementations and the composition of implementations as studied in our framework. The main assumption to preserve correctness, however, is persistency of the extension step. In our case this would imply that SORTIMPL is a persistent subspecification of OPIMPL which is not valid for most of our examples studied in [16].

Actually we also have considered persistent implementations in this sense in [16]. They turn out to lead to much nicer results but seem to be of more limited applicability to practical implementation problems. Persistent implementations in a similar sense are also studied in [1] and [21].

Finally we should mention another special case of our implementation concept where the sort implementing operations \( \sigma \in \Sigma \) are restricted to copy operations \( \sigma : s1 \rightarrow s0 \) resp. \( \sigma : \text{shid} \rightarrow s0 \) with \( s0 \in S0, s1 \in S1 \) and \( \text{shid} \in \text{shid} \). In this case we can replace the sort implementing operations conceptually by a function \( f : S0 \rightarrow S1 + \text{shid} \). This case still seems to be general enough to handle most of the interesting case. In [28], however, this function is extended to a signature morphism from \( \text{SPECO} \) to \( \text{SPEC} \) (not regarding hidden parts) which requires that the signature of \( \text{SPECO} \) is already available in \( \text{SPEC} \).

5.2. Modifications of the parameterization and parameter passing concept

Our notion of parameterized specifications and parameter passing used in Section 2 is based on [2, 4]. Up to now we have only used standard parameter passing because this allows in a natural way to show the compatibility of parameterized implementations and induced standard implementations. The more general case, however, is to study parameterized parameter passing in the sense of [4]. That means that the actual parameter and hence also the value specification are parameterized specifications. Actually there are only slight changes in Theorems 4.3, 4.4 and 4.5 and the corresponding proofs for the case of parameterized parameter passing. Essentially we only need an additional lemma showing compatibility of different restriction constructions. Here we take a ‘passing consistent’ parameter passing morphism \( h : \text{data} \rightarrow \text{stack(par)} \) to the parameterized specification \( \text{stack(par)} \) of
Compatibility of parameter passing

stacks of parameters, the correct implementation

\[
\text{IMPL} : \text{string(data)} \Rightarrow \text{set(data)}
\]

induces a correct implementation

\[
\text{IMPL'} : \text{string} \ast \text{stack(par)} \Rightarrow \text{set} \ast \text{stack(par)}
\]

of parameterized types, where \( \ast \) corresponds to the composition of parameterized types in the sense of [4].

Another useful extension of our approach studied so far is to allow parameterized specifications with requirements (see [10]) or algebraic constraints (see [5]) in the formal parameter part of the specifications. This allows to add requirements like initial restrictions, e.g., initial (bool), and general logical formulas instead of equations. In this framework we can formulate parameterized specifications like \( \text{set(data)} \) and \( \text{string(data)} \), where initiality of the bool-part in data is essential, and also an implementation of \( \text{set(data)} \) by \( \text{string(data)} \). Parameter passing from data to int leads to the induced implementation of \( \text{set(int)} \) by \( \text{string(int)} \) which was studied independently in [16] before.

The notion of requirements studied in [10] is an axiomatic or semantical framework where the structure on the syntactical level is still open. Actually there are essentially no restrictions for the syntactical level in order to extend all the results concerning parameter passing from the basic algebraic case given in [4] and reviewed in Section 2 of this paper to the case with requirements. But a syntactical structure of requirements is necessary in order to give general results concerning verification techniques for passing consistent parameter passing. This means techniques to show that the actual parameter satisfies the requirements given in the formal parameter part. This should be studied separately for logical formulas and algebraic constraints which seen to be the two main classes of requirements. Algebraic constraints were motivated by the notion of initial restrictions introduced by Hurbach, Kaphengst and Reichel (see [27]) and essentially the same notion, called data constraints, which is used in CLEAR (see [8]). The general theory of algebraic constraints given in [5] provides a number of techniques to transform and show equivalence of constraints which can be applied to verify consistency of parameter passing.

It remains open to discuss how our approach in Sections 3 and 4 can be extended to parameterized implementations where the formal parameter part \( \text{SPEC} \) includes requirements. In this case we would expect that the semantical construction preserves the requirements, i.e., \( \text{SEM}_{\text{IMPL}}(A) \in \text{Alg}_{\text{SPEC}} \) satisfies the \( \text{SPEC} \)-requirements provided that they are satisfied by \( A \in \text{Alg}_{\text{SPEC}} \). But this is an easy consequence of IR-correctness which implies \( \text{SEM}_{\text{IMPL}}(A) \equiv \text{FREE_0}(A) \) where \( \text{FREE_0}(A) \) preserves the \( \text{SPEC} \)-requirements because of persistency. The results in Section 4, i.e., Theorems 4.3, 4.4 and 4.5, can be extended to the case with requirements if we assume that the parameter passing morphism \( h : \text{SPEC} \to \text{SPEC}' \) is passing consistent, i.e., \( V_h(T_{\text{SPEC}}) \) satisfies the \( \text{SPEC} \)-requirements.
5.3 Discussion of approaches based on final algebra and loose semantics

As mentioned already in Section 1 there are a number of other approaches within the theory of algebraic specifications which are based on final algebra and loose semantics respectively.

The final algebra approach was introduced by Gianotana et al. [20] and Wand [29]. Parameterized specifications and implementations within this approach are studied by Ganzinger [19]. In [19] it is shown how the concepts of parameterization and parameter passing studied in [4] for the initial free case can be generalized to the final-cofree case. In addition, Ganzinger was able to give a proof-theoretical characterization of persistency. The implementation concept in [19] is similar with respect to the syntax to that given by Ehrich [9] and also to our concepts in [16]. But the semantics lacks a counterpart of OP-completeness and does not allow restriction. An additional technical completeness condition allows to show that implementations are closed under composition and that implementation commutes with parameter passing.

Algebraic specifications with loose semantics are studied in [8, 27]. Corresponding parameterized implementation concepts are studied in [28] and [24] respectively. Both concepts are using parameterized specifications with requirements which are called hierarchy resp. data constraints in [28] and initial restrictions in [24]. In principle this would allow to specialize their constructions and results to our case of initial-free semantics. A closer look at the problem of correct parameter passing, however, shows that the corresponding results after specialization are at first sight much weaker than our results in the initial-free case. Actually one needs the techniques of the initial-free case to show that the corresponding translated constraints are equivalent to those needed for the initial-free case. A number of such techniques is provided in [5] but has not been applied to parameterized implementations. The main result in [24] concerning parameterized implementations with loose semantics is similar to our Theorems 4.3 and 4.5 where, however, the correctness conditions are different. Similar to [19] there is no restriction counterpart in [24]. The implementation concept in [20] is more or less our concept in [16] without the synthesis step but including restriction and identification. As mentioned in Section 5.1 this can be extended to the case including synthesis only in the case of persistent enrichments. Implementations of parameterized specifications are defined to be correct in [28] if for all actual parameters the induced implementations are correct. We have considered this to be the main problem to be solved (see Theorem 4.5). There should be a notion of correctness for parameterized implementations which can be checked independently of actualizations such that all induced implementations become correct. On the other hand, the problem of ‘outer parameterization’ (see Section 5.4) has been solved in [28] which is not yet solved in our framework of initial-free semantics.

Results similar to those in [28] are also stated by Goguen and Meseguer [21] where specifications of ‘modules’ (i.e., data types with interface) without constraints
are considered. A purely semantical approach to the problem of compatibility of parameter passing and implementation is proposed by Lippeck [25]. This semantical framework is intended to serve as a semantical basis for different kinds of specification languages. The results in [25] concerning compatibility will be related to our approach and to those mentioned above in the final version of [25].

5.4. The problem of 2-dimensional compatibility

The idea of modularization means that complex, large systems are built up from small, handable pieces. Say, we have specifications stack, queue, set, int (maybe from a library) and we write parameter passing morphisms to combine them to a complex modul stack (queue(set(int))). On the other hand we may refine each modul by implementation steps:

- array with pointer \( \Rightarrow \) stack,
- array with two pointers \( \Rightarrow \) cyclic list \( \Rightarrow \) queue,
- array \( \Rightarrow \) hashtable \( \Rightarrow \) set.

(array may be the only structured data type of our programming language.) It would be quite nice now if the final system, i.e.,

\[
\text{array with pointer (array with two pointers (array(int)))} \Rightarrow \text{stack(queue(set(int)))},
\]

is obtained from the parts above automatically and that the result of their composition does not depend on the order. You would also like that the result is correct if all its building blocks are. (Otherwise you could not say too much about your system until you have finished to develop it.) The key to satisfy such desires is the concept of 2-dimensional compatibility which we are going to discuss in the following.

Our main result in Section 4 shows that actualization of the formal parameters of a correct parameterized implementation leads to a correct induced implementation. Let us call this process 'inner actualization'. In Example 4.2 the inner parameters data and int (now short for intmax) of the parameterized implementation

\[
\text{impl} : \text{bracketstring(data + int)} \Rightarrow \text{bintree(data + int)}
\]

are both actualized by int leading to the induced implementation

\[
\text{impl'} : \text{bracketstring(int)} \Rightarrow \text{bintree(int)}.
\]

If the actual parameter is a parameterized specification, the induced implementation becomes a parameterized implementation.

In addition to 'inner actualization' there is also another compatibility problem of parameter passing and implementation, called 'outer parameterization'.

Given the parameterized implementation

\[
\text{impl} : \text{bracketstring(data + int)} \Rightarrow \text{bintree(data + int)}
\]
as above and an ‘outer’ parameterized specification, say $\text{stack}(\text{param})$, we can actualize the formal parameter \text{param} with the parameterized specifications of the given implementation leading to parameterized specifications

$\text{stack} \ast \text{bracketstring}(\text{data} + \text{int})$ and $\text{stack} \ast \text{bintree}(\text{data} + \text{int})$.

The problem is now whether we obtain a correct induced implementation

$\text{IMPL}^*: \text{stack} \ast \text{bracketstring}(\text{data} + \text{int}) \Rightarrow \text{stack} \ast \text{bintree}(\text{data} + \text{int})$

This problem is called ‘outer parameterization’. A solution for this problem in some restricted cases is given in [28] for loose and in [19] for final algebra semantics. The problem is still open for our case of initial-free semantics and for the general case of parameterized implementations as studied in this paper.

If the processes of ‘inner actualization’ and ‘outer parameterization’ are well-defined and commutable (see below) they can be combined to the process of ‘horizontal composition’:

Given implementations

$\text{IMPL} : \text{SPEC1}(\text{SPEC}) \Rightarrow \text{SPEC0}(\text{SPEC})$  \hspace{1cm}  $\text{IMPL}^* : \text{SPEC1}'(\text{SPEC}') \Rightarrow \text{SPEC0}'(\text{SPEC}')$

and suitable parameter passing morphisms

$h_0 : \text{SPEC} \Rightarrow \text{SPEC0}'(\text{SPEC}')$ \hspace{1cm} $h_1 : \text{SPEC} \Rightarrow \text{SPEC1}'(\text{SPEC}')$

the horizontal composition

$\text{IMPL} \ast \text{IMPL}^* : \text{SPEC1} \ast \text{SPEC1}'(\text{SPEC}') \Rightarrow \text{SPEC0} \ast \text{SPEC0}'(\text{SPEC}')$

is well defined as composition if the following diagram of inner actualizations (INACT) and outer parameterizations (OUTPAR) commutes:

In contrast to ‘horizontal composition’ as introduced above the usual composition of implementations as studied in [16] will now be called ‘vertical composition’:
Given parameterized implementations with hidden components (see Section 5.1), say

\[
\text{SPEC}_2(\text{SPEC}) \xrightarrow{\text{IMPL}_2} \text{SPEC}_1(\text{SPEC}) \xrightarrow{\text{IMPL}_1} \text{SPEC}_0(\text{SPEC}),
\]

the \textit{vertical composition} can be defined as in [16, Section 6] leading to a parameterized implementation

\[
\text{SPEC}_2(\text{SPEC}) \xrightarrow{\text{IMPL}_1 \cdot \text{IMPL}_2} \text{SPEC}_0(\text{SPEC})
\]

where the correctness problem has to be handled similar to [16, Section 7].

Now we are able to formulate the problem of \textquote{2-dimensional compatibility of parameter passing and implementation}:

Assume that we have horizontal and vertical composition of parameterized implementations as introduced above where both compositions preserve correctness of implementations:

Given correct implementations

\[
\text{SPEC}_2(\text{SPEC}) \xrightarrow{\text{IMPL}_2} \text{SPEC}_1(\text{SPEC}) \xrightarrow{\text{IMPL}_1} \text{SPEC}_0(\text{SPEC})
\]

and

\[
\text{SPEC}_2'(\text{SPEC}') \xrightarrow{\text{IMPL}_2'} \text{SPEC}_1'(\text{SPEC}') \xrightarrow{\text{IMPL}_1'} \text{SPEC}_0'(\text{SPEC}')
\]

and suitable parameter passing morphisms such that the following horizontal compositions are well defined:

1. \[
\text{SPEC}_2 \cdot \text{SPEC}_2'(\text{SPEC}) \xrightarrow{\text{IMPL}_2 \cdot \text{IMPL}_2'} \text{SPEC}_1 \cdot \text{SPEC}_1'(\text{SPEC}')
\]
2. \[
\text{SPEC}_1 \cdot \text{SPEC}_1'(\text{SPEC}') \xrightarrow{\text{IMPL}_1 \cdot \text{IMPL}_1'} \text{SPEC}_0 \cdot \text{SPEC}_0'(\text{SPEC}')
\]

On the other hand we have the following vertical compositions:

3. \[
\text{SPEC}_2(\text{SPEC}) \xrightarrow{\text{IMPL}_1 \cdot \text{IMPL}_2} \text{SPEC}_0(\text{SPEC})
\]

4. \[
\text{SPEC}_2'(\text{SPEC}') \xrightarrow{\text{IMPL}_1 \cdot \text{IMPL}_2} \text{SPEC}_0'(\text{SPEC}')
\]

Now we say that we have \textit{2-dimensional compatibility of parameter passing and implementation} if there are uniquely defined induced parameter passing morphisms such that the vertical composition (5) of (1) and (2):

\[
\text{SPEC}_2 \cdot \text{SPEC}_2'(\text{SPEC}') \xrightarrow{\text{IMPL}_1 \cdot \text{IMPL}_1' \cdot \text{IMPL}_2 \cdot \text{IMPL}_2'} \text{SPEC}_0 \cdot \text{SPEC}_0'(\text{SPEC}')
\]
is well defined and equal to the horizontal composition (6) of (3) and (4):

\[
(6) \quad \text{SPEC2} \times \text{SPEC2}'(\text{SPEC}') \quad \xrightarrow{(\text{IMPL}_1 \times \text{IMPL}_2) \circ (\text{IMPL}_1' \times \text{IMPL}_2')} \quad \text{SPEC0} \times \text{SPEC0}'(\text{SPEC}')
\]

In short formulas full compatibility means

\[
(7) \quad ((\text{IMPL}_1 \times \text{IMPL}_1') \circ (\text{IMPL}_2 \times \text{IMPL}_2')) = ((\text{IMPL}_1 \circ \text{IMPL}_2) \times (\text{IMPL}_1' \circ \text{IMPL}_2'))
\]

where all composite implementation are assumed to be well defined and correct. We have called the property defined above '2-dimensional compatibility' because (7) corresponds exactly to the 'double law' of a 2-dimensional category (see [26]). The idea to require compatibility between horizontal and vertical composition of implementations in terms of this 'double law' was first proposed by Burstall and Goguen for the specification language CAT [7]. In [28] and [21] it is stated without proof that the double law holds for parameterized implementations with loose semantics provided that a number of additional assumptions is satisfied. However, as already mentioned in Section 5.3, the implementation concepts in [28] and [21] are both restricted with respect to a number of desirable properties. This means that even for loose semantics the problem of 2-dimensional compatibility of parameter passing and implementation can be considered to be still open.

### 5.5. Conclusion

In this paper we have started to show how the concepts of parameterization and parameter passing developed in [2] and [4] on one hand and the concept of implementation as given in [16] can be combined to a theory of parameterized implementations. Our main result shows that inner actualization of a correct parameterized implementation yields a correct induced implementation. This is an important step in order to achieve 2-dimensional compatibility of parameter passing and implementation. Such a compatibility is necessary in order to guarantee the correctness of a stepwise refinement strategy for structured specifications, independent of the order in which horizontal and vertical composition are carried out. Hence this compatibility is highly desirable for the development of software systems.

In the last section we have discussed a number of other approaches which are based on loose or final algebra semantics. Each of these approaches is based on a different—in most cases simplified—implementation concept. The 'intersection' of all these concepts would lead to the following 'most simplified' implementation concept (ms-implementation, for short) based on loose semantics and persistent parameterized specifications without requirements:

An ms-implementation of $\text{PSPEC0} = (\text{SPEC}, \text{SPEC0})$ by $\text{PSPECT} = (\text{SPEC}, \text{SPEC1})$ is a signature morphism $\sigma$ from the signature of $\text{SPEC0}$ to that of $\text{SPEC1}$ which is the identity on the formal parameter $\text{SPEC}$. The loose semantics of this ms-implementation is the functor $F : \text{AlgSPEC1} \rightarrow \text{AlgSPEC0}$ defined as composition of RESTRICTION
Compatibility of parameter passing

and identification. Restriction is based on the forgetful functor \( V \), associated with the signature morphism \( f \) and identification is done by the \( \text{spec} \)-equations. This makes sure that the semantical functor \( F \) is well defined and each ms-implementation is correct.

Now it would be an easy practice in universal algebra to define horizontal and vertical composition of ms-implementation and to show 2-dimensional compatibility of parameter passing and implementation.

This example shows that the quality of the results concerning compatibility of parameter passing and implementation highly depends on the kind of semantics, on the question of requirements for parameterized specifications, and the notion of implementation including correctness. If—on the other hand—we take the union of all the concepts studied in all the approaches mentioned above very little would be known concerning compatibility of parameter passing and implementation.

But all the results in the different approaches known up to now are promising enough to suggest that the problem of 2-dimensional compatibility can be solved within the next years for a notion of implementations which is general enough for software engineering purposes.

References


