Theoretical Computer Science 27 (1983) 167–196
North-Holland

ALGEBRAIC DOMAIN EQUATIONS

Hans-Dieter EHRICH and Udo LIPECK

Institut für Theoretische und Praktische Informatik, Technische Universität Braunschweig, D-3300 Braunschweig, Fed. Rep. Germany.

Communicated by R. Milner
Received December 1981
Revised December 1982

Abstract. This paper discusses a new specification method for algebraic data types consisting of an algebraic analogon to domain equations as known from Scott's theory (1971) of order-theoretic data types. The main result is that strongly persistent algebraic domain equations always have an initial solution, and there is a simple syntactic method to construct a specification of this initial solution. Examples illustrate the usefulness of implicit specifications in certain cases. Then, a parametric version of algebraic domain equations is introduced having parameterized data types as solutions. It is shown that there is always a solution that can be obtained syntactically as in the nonparameterized case. This solution behaves consistently with respect to parameter passing.

1. Introduction

The word 'data type' has been used in many different ways for many different (though somehow related) concepts. Among the approaches to make this notion mathematically precise, there are two main lines that have been discussed widely: the order-theoretic approach originated by Scott [24], and the algebraic approach brought into discussion by Guttag [14] and put on precise mathematical grounds by Goguen, Thatcher and Wagner (ADJ Group) [2]. Although these approaches do not model the same aspects and serve different purposes, it is quite interesting to compare the respective features and try to apply ideas from one approach to the further development of the other one. The theory of continuous algebras [1] and the papers of Lehmann and Smyth [18] and Kanda [15] represent different attempts to do this.

This paper contributes to the algebraic approach by adding an analogon to one of the most distinguished features of the order-theoretic approach, namely implicit (or circular or recursive) specification of data types by means of domain equations like, for example, the following [25, 30]: $N \cong N+1$ (natural numbers), $S \cong S \times D+1$ (stacks over D), $L \cong (L \times L)+D$ (list structures over D), $F \cong [F \to F]$ (a pure λ-calculus model), etc.

In Scott's order-theoretic approach, a data type serves as a semantic domain for the denotational semantics of programming languages. In order to utilize Tarski's lattice-theoretic fixpoint theorem for giving unique meanings to implicit definitions of functionals, data types are defined to be certain continuous partial orderings, e.g., lattices or cpo's. The construction of data types proceeds from given basic types (like BOOL or INT) by means of type constructors, basically sum, product and function space. Additionally, implicit definitions by means of recursive domain equations play a very important role. An elementary introduction into this subject can be found in [25]. There are two different approaches to solving recursive domain equations, using inverse limits of projection sequences [24–26] or using universal domains [27, 22].

In order to cope with nondeterminism, powerdomain constructors have been introduced and investigated [21, 28]. One of the main criteria for the suitability of a powerdomain constructor is that it permits the solution of recursive domain equations, e.g., Plotkin's example for 'resumptions': $R \cong [S \to \mathcal{P}(S+(S\times R))]$ [21]. Smyth [29] and Kanda [16] studied effectivity problems of domains and recursive domain equations.

Wand [31, 32] was the first to apply categorical notions and methods to the solution of recursive domain equations, generalizing Tarski's fixpoint theorem to non-skeletal categories. This theory was further developed by Lehmann [17] and by Smyth and Plotkin [30]. Lehmann and Smyth [18] adopted certain ideas from the algebraic approach. They equipped their data types with operations definable by means of the initial solution of an ω-functor representing a recursive domain equation.

The algebraic approach emphasizes the operational structure of a data type ignored by the order-theoretic approach. According to this philosophy, a data type is adequately modelled by a (many-sorted) algebra. Goguen et al. [1] showed how semantics of programming languages can be understood in these terms. After Guttag's paper [14] this approach has also been quite successful in studying various problems of specification of data abstractions and software modules.

A central issue of the algebraic approach is to utilize initiality to give unique meanings to specifications [1, 2]. This suggests a specification methodology that is quite different from that of the order-theoretic approach. Basically, algebraic data types are specified explicitly by giving the operations as well as conditions (most often equations) they have to obey.

In the framework of their specification language CLEAR, Burstall and Goguen [5] informally introduced an analogon of data type constructors to algebraic specifications, called 'theory procedures'. Their mathematical counterparts, parameterized data types, have been studied extensively: Ehrich [7] and Ehrich and Lohberger [9, 10] concentrated on syntactic aspects of parameter passing utilizing pushouts, and the semantics of parameter passing has been clarified in [11, 12] on the basis of ideas exposed in [3]. Burstall and Goguen [6] have contributed considerably to the theory by providing a formal algebraic semantics for CLEAR. A CLEAR-like specification method has been used by Mosses [20] to describe the semantics of programming languages.

It is not quite obvious how to compare the respective benefits and drawbacks of the two approaches to data types outlined above. There is some discussion on this point in [18] and [3]. The latter point out that the carrier sets of algebraic data types constitute in a certain sense initial solutions of corresponding polynomial domain equations. This aspect is further elaborated by Kanda [16] who uses domain equations to describe denotations of sorts by which algebras are to be extended, describing parameterization this way.

Our approach to implicit specifications follows a different line. Our concern is to describe algebras with all their carriers and all their operations as solutions of 'algebraic domain equations'. This is done by means of fixpoints of endofunctors on categories of algebras. Among the solutions there is an initial one that can serve as a standard semantics. In particular, we offer a surprisingly simple syntactic solution method for algebraic domain equations utilizing coequalizers in the category of specifications.

It turns out that solving a domain equation means constructing an explicit specification whose initial algebra gives the solution. This excludes the possibility of defining genuinely infinite structures, e.g., Scott's reflexive domain [26]. Analogous domain equations, however, have 'finitary' solutions in our framework.

2. The basic idea

Before we go into technical details, we illustrate the basic idea and the main result of our approach informally. Very roughly speaking, a domain equation consists of a *parameterized data type* (or data type constructor), sending actual parameter types to resultant types, and a solution is a sort of fixpoint of this mapping.

The algebraic and order-theoretic approaches to data types agree that parameterized data types are functors. With lattices or cpo's, we have all data types naturally collected in one and the same category, so a (one-argument) parameterization is an endofunctor in a natural way. Thus, we can conveniently interpret it as a recursive domain equation and look for fixpoints. To put it in other words, (one-argument) parameterizations and recursive domain equations happen to coincide in the order-theoretic approach.

Unfortunately, this does not hold for algebraic domains. In all interesting cases, a parameterized data type P is not an endofunctor. Usually, P adds new operations and new sorts of objects when applied to an actual parameter, so that the signature of the resulting data type is different from that one of the parameter. For example, 'stack(X)' sends any algebra A to an algebra of, say, strings of elements of A having one more sort of objects, strings, and several additional operations, e.g., push, pop, etc. As a consequence, stack(X) cannot be expected to have fixpoints.

An obvious approach to giving an equation like '$X \cong \text{stack}(X)$' an algebraic meaning is to look for an actual parameter algebra A such that a suitable *reduct* of stack(A) is isomorphic to A. For instance, if A simply is a set, it makes sense to ask whether the set of stacks underlying stack(A) is isomorphic to A.

More technically speaking, let Act and Res be categories of actual parameter algebras and of resultant algebras, respectively, and let $P: \mathit{Act} \to \mathit{Res}$ be a functor given by a parameterized data type. We assume a 'forgetful functor' $\bar{E}: \mathit{Res} \to \mathit{Act}$ sending each resultant algebra to a suitable reduct that has the same signature as the actual parameter algebras. Now we have an endofunctor $P\bar{E}: \mathit{Act} \to \mathit{Act}$, and we can look for fixpoints A such that $A \cong AP\bar{E}$.

These fixpoints, however, are not very interesting since most of the structure of the resultant algebra is forgotten. Fortunately, P and $\bar{E}$ can also be composed to give another endofunctor, namely $\bar{E}P: \mathit{Res} \to \mathit{Res}$. Fixpoints of $\bar{E}P$ carry all the structure of resultant algebras, so they look more interesting. We will show in Section 5 that fixpoints of $P\bar{E}$ and fixpoints of $\bar{E}P$ are very closely related under the assumptions made there.

But still, fixpoints of $\bar{E}P$ do not give what we want. In order to see this, consider a very simple example in greater detail.

Example 2.1. The domain equation $X \cong X + 1$ is supposed to have the natural numbers as initial solution [25]. In an algebraic interpretation, let X range over pointed sets, and let the parameterized data type that corresponds to '$X+1$' send a pointed set (M, i) to the 2-sorted algebra consisting of two pointed sets

$$(M, i) \quad \text{and} \quad (M + \{0\}, 0)$$

(where a new point is added in the new carrier) and, say, two operations relating the two carriers as follows:

$$\sigma: M \to M + \{0\}, \quad \sigma(m) = m \ \text{ for each } m \in M,$$

$$\pi: M + \{0\} \to M, \quad \pi(0) = i, \ \pi(m) = m \ \text{ for each } m \in M.$$

Let E send any such algebra to $(M + \{0\}, 0)$, i.e., the pointed set constituting the new carrier.

Then, the initial fixpoint of $P\bar{E}$ is the pointed set $(\mathbb{N}, 0)$ of natural numbers.

The fixpoint of $\bar{E}P$ has a richer structure; it is isomorphic to the following 2-sorted algebra:

$$\text{1st sort:} \quad (\mathbb{N}, 0),$$

$$\text{2nd sort:} \quad (\mathbb{N}' + \{0\}, 0),$$

where $\mathbb{N}'$ is the set $\{1, 2, 3, \ldots\}$ of successors of natural numbers, with the following operations:

$$\sigma: \mathbb{N} \to \mathbb{N}' + \{0\}, \quad \text{the successor function,}$$

$$\pi: \mathbb{N}' + \{0\} \to \mathbb{N}, \quad \text{the predecessor function.}$$

This comes close to the desired solution that has just one sort, $\mathbb{N}$, together with the distinguished constant 0 and the successor and predecessor functions on $\mathbb{N}$. We only have to identify the two sorts of our fixpoint of $\bar{E}P$ that are, in a sense, 'the same'.

Generally, solutions of our algebraic domain equations will be neither fixpoints of $P\bar{E}$ nor fixpoints of $\bar{E}P$, but they will result from the latter by identifying certain sorts and operations. Technically, this identification is achieved by a coequalizer construction.

In our view, algebraic domain equations are syntactic entities consisting of a pair (p, e), where p is a syntactic description of a parameterized data type, and e is a syntactic description of a forgetful functor. We will make these notions precise in Section 5. In the following two sections we will develop the necessary mathematical background (Section 3) and give a short account of the theory of parameterization of algebraic data types (Section 4).

3. Specifications and algebras

We adopt the basic assumption that abstract data types are (isomorphism classes of) many-sorted algebras. In this section we briefly review the relevant algebraic notions used in subsequent sections. The algebraic background as applied to abstract data types is treated in greater detail in [1, 2, 7] and several other places. As a general background reference we refer to [4] and [23].

The syntax of a many-sorted algebra is described by a signature giving sorts as names of carrier sets and operators as names of operations on these carrier sets.

Definition 3.1. A *signature* Σ is a quadruple $\langle S, \Omega, \text{arity}, \text{sort}\rangle$ where S is a set of sorts and Ω a set of operators equipped with two mappings, $\text{arity}: \Omega \to S^*$ and $\text{sort}: \Omega \to S$.

The arity of an operator denotes the list of sorts of the arguments, and the sort of an operator denotes its result sort. In order to facilitate notation, we will denote signatures by $\Sigma = \langle S, \Omega\rangle$ omitting the arity and sort mappings. They are tacitly assumed to be given. An operator ω with arity $x = s_1 \cdots s_n$ and sort s_0 will conventionally be denoted by $\omega : s_1 \times \cdots \times s_n \to s_0$.

Signatures may be related by structure preserving maps called signature morphisms.

Definition 3.2. Let $\Sigma = \langle S, \Omega\rangle$ and $\Sigma' = \langle S', \Omega'\rangle$ be signatures. A *signature morphism* $f: \Sigma \to \Sigma'$ is a pair of mappings $\langle f_s: S \to S', f_\omega: \Omega \to \Omega'\rangle$ such that $\text{arity}'(\omega f_\omega) = \text{arity}(\omega) f_s$ and $\text{sort}'(\omega f_\omega) = \text{sort}(\omega) f_s$.

For convenience, we write sf, xf, ωf, etc. respectively for sf_s, xf_s, ωf_ω, etc. Please note that f_s is tacitly extended letterwise to strings $x \in S^*$. The class of all signatures with all signature morphisms forms a category denoted by *sign*.

Algebras are interpretations of signatures. The following definition makes this precise.

Definition 3.3. Let $\Sigma = \langle S, \Omega\rangle$ be a signature. A *Σ-algebra A* is an S-indexed family of sets $\{s_A\}$, the *carrier* of A, together with an Ω-indexed family of mappings $\{\omega_A : \text{arity}(\omega)_A \to \text{sort}(\omega)_A\}$, the *operations* of A.

Here, x_A denotes the cartesian product $s_{1,A} \times \cdots \times s_{n,A}$ if $x = s_1 \cdots s_n \in S^*$. Σ-algebras may be related by structure preserving maps called Σ-algebra morphisms.

Definition 3.4. Let $\Sigma = \langle S, \Omega\rangle$ be a signature, and let A, B be Σ-algebras. A *Σ-algebra morphism $\varphi : A \to B$* is an S-indexed family of mappings $\{\varphi_s : s_A \to s_B\}$, such that, for each operator ω with arity x and sort s, we have $\omega_A \varphi_s = \varphi_x \omega_B$.

Here, $\varphi_x = \varphi_{s_1} \times \cdots \times \varphi_{s_n}$ if $x = s_1 \cdots s_n$. The morphism condition is illustrated by commutativity of the following diagram:

$$\begin{tikzcd}
x_A \arrow[r, "\omega_A"] \arrow[d, "\varphi_x"'] & s_A \arrow[d, "\varphi_s"] \\
x_B \arrow[r, "\omega_B"] & s_B
\end{tikzcd}$$

The class of all Σ-algebras together with all Σ-algebra morphisms forms a category denoted by Σ-*alg*. It is well known that there is an *initial algebra* I_Σ in Σ-*alg* that is unique up to isomorphism. Its defining property is that there is exactly one morphism from I_Σ to any Σ-algebra. A standard construction of I_Σ is as follows: The carrier consists of all terms that can be constructed from the operators Ω without variables, and the operations are given by formal application of the respective operators.

If $f : \Sigma \to \Sigma'$ is a signature morphism, there is a corresponding forgetful functor *f-alg* : Σ'-*alg* $\to \Sigma$-*alg* defined as follows. Let $\bar{F} =$ *f-alg*. If B is a Σ'-algebra, then $A = B\bar{F}$ has carrier sets $s_A = (sf)_B$ and operations $\omega_A = (\omega f)_B$; if φ is a Σ'-algebra morphism, then $\psi = \varphi\bar{F}$ has components $\psi_s = \varphi_{sf}$. It is easy to verify that $\bar{F}$ is indeed a functor, i.e., that $\text{id}_B\bar{F} = \text{id}_{B\bar{F}}$ and $(\varphi\rho)\bar{F} = \varphi\bar{F} \circ \rho\bar{F}$, whenever $\varphi\rho$ is defined. The association *alg*: $\Sigma \mapsto \Sigma$-*alg*$/f \mapsto$ *f-alg* is a functor, too. Its range is the category *cat* of all categories with all functors. Thus we have a (contravariant) functor *alg* : *sign* $\to$ *cat*$^{\text{op}}$.

Abstract data types are specified by giving a signature Σ and a set of conditions E on the operators. It is tradition to use equations for expressing the conditions, because they have nice algebraic properties. It has been argued [11, 12] that a generalization to universal Horn clauses maintains the relevant properties needed in the theory. For the sake of simplicity, we stick to equations. Our results, however, can be generalized to universal Horn clauses.

Definition 3.5. Let $\Sigma = \langle S, \Omega \rangle$ be a signature. A *Σ-equation* is a triple $\langle X, \tau_1, \tau_2 \rangle$ where X is an S-indexed family of sets of *variables* and τ_1, τ_2 are terms over X and Ω of the same sort, called the *sort of the equation.*

Instead of $\langle X, \tau_1, \tau_2 \rangle$ we will write $\forall X\colon \tau_1 = \tau_2$ or simply $\tau_1 = \tau_2$, if X is understood to consist of all variables occurring in τ_1 or τ_2.

Definition 3.6. A Σ-algebra A *satisfies a Σ-equation* $\forall X\colon \tau_1 = \tau_2$ iff this formula is valid when interpreted in A in the obvious way.

Definition 3.7. A *specification* D is a pair $\langle \Sigma, E \rangle$ where Σ is a signature and E is an S-sorted set of Σ-equations. If $\Sigma = \langle S, \Omega \rangle$, we sometimes write $D = \langle S, \Omega, E \rangle$ instead of $D = \langle \Sigma, E \rangle$.

Definition 3.8. Let $D = \langle \Sigma, E \rangle$ be a specification. A *D-algebra* is a Σ-algebra satisfying each equation in E.

A specification $D = \langle \Sigma, E \rangle$ determines the full subcategory of Σ-*alg* consisting of all D-algebras; this category is denoted by D-*alg*. An essential property of D-*alg* is that it has an initial object I_D, determined uniquely up to isomorphism. In the initial algebra approach to abstract data types, I_D is defined to be the abstract data type specified by D. It can be constructed by factorizing I_Σ by the congruence relation generated by E.

Specifications may be related by the structure preserving maps, called specification morphisms. There are several different notions of specification morphism in the literature. The version we use here is stronger than that used in [7], but weaker than that in [11, 12]. Our present notion is equivalent to that used in [6], called theory morphism there.

Definition 3.9. Let $D = \langle \Sigma, E \rangle$, $D' = \langle \Sigma', E \rangle$ be specifications. A *specification morphism* $f\colon D \to D'$ is a signature morphism $f\colon \Sigma \to \Sigma'$ such that the corresponding forgetful functor f-*alg* : Σ'-*alg* $\to \Sigma$-*alg* sends each D'-algebra to a D-algebra.

Thus, a specification morphism $f\colon D \to D'$ defines a corresponding forgetful functor f-*alg* : D'-*alg* $\to D$-*alg* by restricting the forgetful functor corresponding to f, viewed as a signature morphism, to the subcategory D'-*alg*.

The class of all specifications together with all specification morphisms forms a category denoted by *spec*, and there is an obvious forgetful functor from *spec* to *sign* associating with each specification its underlying signature and leaving the morphisms fixed. We also have a functor *alg* : *spec* $\to$ *cat*$^{\mathrm{op}}$ sending D to D-*alg* and f to f-*alg*.

Several constructions to be performed on specifications in the following sections will rely on structural properties of *spec* that guarantee that certain categorical standard constructions work. Especially, we will prove that *spec* has colimits and show how to construct those colimits we need, namely coproducts, coequalizers and pushouts.

In an arbitrary category, a *coproduct* of a family $\{A_k\}$ of objects is an object C together with a family of morphisms $\{i_k : A_k \to C\}$ such that, for any family of morphisms $\{j_k : A_k \to D\}$, there is exactly one morphism $h: C \to D$ such that, for each k, $i_k h = j_k$. In the category *set* of sets and functions, coproducts are disjoint unions with the obvious inclusions.

A *coequalizer* of two morphisms $f, g: A \to B$ with common source and target is an object C together with a morphism $c: B \to C$ such that (i) $fc = gc$, and (ii) for any $d: B \to D$ such that $fd = gd$ there is exactly one morphism $h: C \to D$ such that $ch = d$. In *set*, coequalizers are the canonical mappings $c: B \to B/\!\sim$ mapping each element of B to its equivalence class with respect to the equivalence relation on B generated by the relation $\{(af, ag) \mid a \in A\}$.

A *pushout* of two morphisms $f_k : A \to B_k$, $k = 1, 2$, with common source is an object C together with two morphisms $g_k : B_k \to C$, $k = 1, 2$, such that (i) $f_1 g_1 = f_2 g_2$, and (ii) whenever there are $d_k : B_k \to D$, $k = 1, 2$, such that $f_1 d_1 = f_2 d_2$, then there is a unique morphism $h: C \to D$ such that, for $k = 1, 2$, $g_k h = d_k$.

Such a pushout may be constructed by first constructing the coproduct $\{i_1 : B_1 \to B', i_2 : B_2 \to B'\}$, and then constructing the coequalizer of $f_1 i_1$ and $f_2 i_2$, say $c: B' \to C$. Then, $g_k := i_k c$, $k = 1, 2$, gives a pushout.

For later use, we define one more instance of a colimit, namely that of a chain. A *chain* is a sequence $a_0, a_1, \ldots$ of morphisms such that the target of a_n coincides with the source of a_{n+1}, for $n = 0, 1, \ldots$, i.e., we have the following situation:

$$\begin{tikzcd} A_0 \arrow[r, "a_0"] & A_1 \arrow[r, "a_1"] & A_2 \arrow[r, "a_2"] & \cdots . \end{tikzcd}$$

A colimit of that chain is an object C together with a family of morphisms $\{c_n : A_n \to C \mid n = 0, 1, \ldots\}$ such that (i) $a_n c_{n+1} = c_n$ for each n, and (ii) for any family $\{d_n : A_n \to D\}$ such that $a_n d_{n+1} = d_n$ there is a unique morphism $h: C \to D$ such that, for each n, $c_n h = d_n$.

A category is said to have coproducts (coequalizers, pushouts) iff in this category each family of objects has a coproduct (each appropriate pair of morphisms has a coequalizer or pushout, respectively). Without going into the general definition of a colimit, we quote a well-known category theoretic result that colimits in general exist iff only coproducts and coequalizers exist [23]. A category that has all colimits is said to be *cocomplete*.

Theorem 3.10. *spec is cocomplete.*

Proof. Due to the remark above, we have to show that *spec* has coproducts and coequalizers. We only give a sketch of the constructions. The rest of the proof is straightforward (cf. [7]).

Coproducts of specifications are simply obtained by taking the disjoint unions of their sorts, operators and equations, e.g., $D_1 + D_2 = \langle S_1 + S_2, \Omega_1 + \Omega_2, E_1 + E_2 \rangle$.

The coequalizer of a pair of specification morphisms $f_1, f_2 : D_1 \to D_2$ is constructed in the following steps:

(1) let $f_{3,s} : S_2 \to S_3$ be the coequalizer in *set* of (the sort parts of) f_1 and f_2,

(2) let $f_{3,\omega} : \Omega_2 \to \Omega_3$ be the coequalizer in *set* of (the operator parts of) f_1 and f_2,

(3) the mappings arity: $\Omega_3 \to S_3^*$ and sort: $\Omega_3 \to S_3$ are defined by the requirement that $f_3 = \langle f_{3,s}, f_{3,\omega} \rangle$ be a signature morphism,

(4) let the equations E_3 be obtained from E_2 by renaming all occurring operators and sorts by f_3, i.e., $E_3 = E_2 f_3$. Then, $f_3 : D_2 \to D_3$ where $D_3 = \langle S_3, \Omega_3, E_3 \rangle$ is the coequalizer of f_1 and f_2 in *spec*. □

It immediately follows that *spec* has pushouts. As explained before, they can be constructed by taking first the coproduct and then the coequalizer.

For each specification D in *spec*, the category D-*alg* of D-algebras is cocomplete, too (cf. [23]). This fact will be used in the proof of our main result (Section 6).

One of the most aesthetic features of category theory is that most of the notions occur in dual pairs, related by 'reversing arrows'. The duals of colimits, coproducts, coequalizers and pushouts are limits, products, equalizers and pullbacks, respectively. For instance, the definition of a pullback reads as follows: a *pullback* of two morphisms $f_k : B_k \to A$, $k = 1, 2$, with common target is an object C together with two morphisms $g_k : C \to B_k$, $k = 1, 2$, such that (i) $g_1 f_1 = g_2 f_2$, and (ii) whenever there are $d_k : D \to B_k$, $k = 1, 2$, such that $d_1 f_1 = d_2 f_2$, then there is a unique morphism $h : D \to C$ such that, for $k = 1, 2$, $h g_k = d_k$.

A useful general result due to Lawvere that is implicit in the proof of the main theorem in [11, 12] can now be stated as follows.

Theorem 3.11. *The functor alg: spec → cat*$^{\mathrm{op}}$ *sends colimits in spec to limits in cat.*

We will only use the special instances that coequalizers and pushouts in *spec* are sent to equalizers and pullbacks in *cat*, respectively. The proof requires a careful analysis of the constructions in question and is rather straightforward. We do not present it here.

This theorem makes the proof of a part of the 'extension lemma' of [11, 12] quite short and elegant. First we need the notion of (strong) persistency.

Definition 3.12. Let $f : D \to D'$ be a specification morphism. A functor $F : D\text{-}alg \to D'\text{-}alg$ is called (*strongly*) *persistent with respect to* f iff $F \circ f\text{-}alg \cong \mathrm{id}$ ($F \circ f\text{-}alg = \mathrm{id}$), where id is the identity functor on D-*alg*.

Lemma 3.13. *Let a pushout be given by the following diagram*:

$$\begin{tikzcd}
D_0 \arrow[r, "f_1"] \arrow[d, "f_2"'] \arrow[dr, phantom, "\text{p.o.}"] & D_1 \arrow[d, "f_3"] \\
D_2 \arrow[r, "f_4"'] & D_3
\end{tikzcd}$$

Let $F_1 : D_0\text{-}alg \to D_1\text{-}alg$ be strongly persistent w.r.t. f_1. Then there is exactly one functor $F_4 : D_2\text{-}alg \to D_3\text{-}alg$ that satisfies the following conditions:

(1) *F_4 is strongly persistent w.r.t. f_4.*

(2) *$f_2\text{-}alg \circ F_1 = F_4 \circ f_3\text{-}alg$.*

Proof. Since F_1 is strongly persistent w.r.t. f_1, we have

$$f_2\text{-}alg \circ F_1 \circ f_1\text{-}alg = f_2\text{-}alg = \mathrm{id} \circ f_2\text{-}alg,$$

where id is the identity on $D_2\text{-}alg$. Since, by Theorem 3.11, the four functors $f_i\text{-}alg$, $i = 1, 2, 3, 4$, form a pullback in *cat*, there is exactly one $F_4 : D_2\text{-}alg \to D_3\text{-}alg$ such that $F_4 \circ f_4\text{-}alg = \mathrm{id}$ and $F_4 \circ f_3\text{-}alg = f_2\text{-}alg \circ F_1$. □

This lemma holds for (weak) persistency too, i.e., when we omit the adjective 'strongly' throughout. The proof, however, is somewhat involved. Since the generalization to (weak) persistency is not so important for our purposes, we will restrict ourselves to the simpler case of strong persistency in what follows.

It is known that algebraic functors of the form $f\text{-}alg$, where f is a specification morphism, have left adjoints. If $f : D \to D'$ is given, we have $f\text{-}alg : D'\text{-}alg \to D\text{-}alg$, and the left adjoint is a functor

$$f\text{-}free : D\text{-}alg \to D'\text{-}alg$$

sending each D-algebra A to a free D'-algebra over A (w.r.t. f). $f\text{-}free$ is defined (up to isomorphism) by the following conditions. Let $F = f\text{-}free$ and $\bar{F} = f\text{-}alg$:

$$\begin{tikzcd}
A \arrow[r, "\eta_A"] \arrow[dr, "\varphi"'] & AF\bar{F} \arrow[d, "\varphi^{\#}\bar{F}"] & AF \arrow[d, "\varphi^{\#}"] \\
 & B\bar{F} & B
\end{tikzcd}$$

(1) For each D-algebra A there is a morphism $\eta_A : A \to AF\bar{F}$ (the 'inclusion of generators').

(2) Each D-algebra morphism of the form $\varphi : A \to B\bar{F}$ (with a D'-algebra B) extends uniquely to a D'-algebra morphism $\varphi^{\#} : AF \to B$ (where 'extends' means that $\eta_A \circ \varphi^{\#}\bar{F} = \varphi$).

(3) F sends each D-algebra morphism $\rho: A \to A'$ to the unique extension of $\rho \circ \eta_{A'}$. (So $\eta: \mathrm{id} \Rightarrow F\bar{F}$ becomes a natural transformation.)

If a given functor F is strongly persistent, the identities $A = AF\bar{F}$, for each D-algebra A, can be taken as η_A, and the following condition suffices to show that $F = f$*-free*:

> For each D-algebra morphism of the form $\varphi: A \to B\bar{F}$ there is exactly one D'-algebra morphism $\varphi^{\#}: AF \to B$ such that $\varphi^{\#}\bar{F} = \varphi$.

Now we are in a position to supplement Lemma 3.13 by another statement concerning freeness. This result is also part of the 'extension lemma' of [11, 12].

Lemma 3.14. *Let the situation of Lemma* 3.13 *be given. If* $F_1 \cong f_1$*-free, then* $F_4 \cong f_4$*-free.*

Proof. Because of strong persistency of F_1 and F_4, we make use of the simplified condition given above.

With the notation of Lemma 3.13 and the diagram therein, let A be a D_2-algebra. For $k = 1, 2, 3, 4$, denote f_k*-alg* by F_k. Furthermore, let $\varphi: A \to B\bar{F}_4$ be a D_2-algebra-morphism.

Assume that $\psi: AF_4 \to B$ is a morphism such that

(a) $\quad \psi\bar{F}_4 = \varphi.$

If we map ψ by $\bar{F}_3$ and φ by $\bar{F}_2$, we get the situation of Fig. 1, considering $\bar{F}_4\bar{F}_2 = \bar{F}_3\bar{F}_1$ and $F_4\bar{F}_3 = \bar{F}_2F_1$.

$$\begin{tikzcd}
A \arrow[dd, "\varphi"] & & & A\bar{F}_2 \arrow[dd, "\varphi\bar{F}_2"] & A\bar{F}_2F_1 \arrow[dd, "\psi\bar{F}_3"'] & & & AF_4 \arrow[dd, "\psi"'] \\
& {} \arrow[r, mapsto] & {} & & & {} & {} \arrow[l, mapsto] & \\
B\bar{F}_4 & & & B\bar{F}_3\bar{F}_1 & B\bar{F}_3 & & & B
\end{tikzcd}$$

Fig. 1.

Since $F_1 \cong f_1$*-free*, there is exactly one morphism $(\varphi\bar{F}_2)^{\#}$ such that

(b) $\quad (\varphi\bar{F}_2)^{\#}\bar{F}_1 = \varphi\bar{F}_2.$

But we have from (a)

$$(\psi\bar{F}_3)\bar{F}_1 = \psi\bar{F}_4\bar{F}_2 = \varphi\bar{F}_2,$$

so we conclude

(c) $\quad \psi\bar{F}_3 = (\varphi\bar{F}_2)^{\#}.$

Since $(\bar{F}_4, \bar{F}_3)$ is a pullback of $(\bar{F}_1, \bar{F}_2)$ (cf. Theorem 3.11), (b) ensures that there indeed exists a morphism ψ that is uniquely determined by (a) and (c), i.e., $\varphi^{\#} = \psi$. $\square$

4. Parameterization

Parameterized data types are in a sense data type constructors, sending actual parameter types to resultant types. They are syntactically specified by parameterized specifications.

Definition 4.1. A *parameterized specification* is an injective *spec* morphism $p: X \to D$. X is called the *formal parameter* of p.

Definition 4.2. A *parameterized data type* is a pair (p, P) where $p: X \to D$ is a parameterized specification, and $P: X\text{-}alg \to D\text{-}alg$ is a functor.

An obvious choice for a standard semantics of a parameterized specification p is the parameterized data type $(p, p\text{-}free)$. This is consistent with the initial algebra philosophy: If X and p are empty, i.e., D is a nonparameterized specification, then $p\text{-}free$ practically 'is' the initial algebra I_D (technically, $p\text{-}free$ sends the only object of $\emptyset\text{-}alg$ to I_D).

In the sequel, we are mainly concerned with parameterized data types (p, P) that behave in an especially orderly way: They preserve in a sense their parameters. Technically, they have the following property.

Definition 4.3. A parameterized data type (p, P) is called (*strongly*) *persistent* iff $P \circ p\text{-}alg \cong \mathrm{id}$ ($P \circ p\text{-}alg = \mathrm{id}$). A parameterized specification p is called (strongly) persistent iff its standard semantics $(p, p\text{-}free)$ has this property.

We briefly describe the mechanism of parameter passing. More details and ramifications can be found in [7, 11, 12]. An intricate problem is the adequate choice of assignments of actual parameters to the formal parameter X. It is obvious that it should be some signature morphism $f: X \to A$, where A is some actual specification, assigning actual sorts and operators. But it is not at all obvious how f should respect the formal equations in X. There are practical examples showing that it is too restrictive to require f to be a *spec* morphism. E.g., in case of passing a non-parameterized specification A, the papers cited above agree that $I_A f\text{-}alg \in X\text{-}alg$ is the adequate notion. For our purpose here, it does not matter too much, and so we prefer to be a little bit less general, getting the advantage of a much smoother formalism.

Definition 4.4. Let $p: X \to D$ be a parameterized specification. An *actual* (*syntactic*) *parameter* for p is a pair (f, A) where A is a specification, and $f: X \to A$ is a *spec* morphism.

Definition 4.5. Let (p, P) be a parameterized data type. An actual parameter for (p, P) is a triple $(f, A, \mathcal{A})$, where (f, A) is an actual syntactic parameter for p, and $\mathcal{A}$ is an A-algebra.

Let (p', f') be the pushout of p and f in *spec*, and let B be the pushout object (cf. Fig. 2(a)). Intuitively speaking, B is the result of substituting A for X in D, where f indicates the details of the substitution.

$$\begin{tikzcd}
X \arrow[r, "p"] \arrow[d, "f"'] \arrow[dr, phantom, "\text{p.o.}"] & D \arrow[d, "f'"] \\
A \arrow[r, "p'"'] & B
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
X\text{-}alg \arrow[r, "P"] & D\text{-}alg \\
A\text{-}alg \arrow[u, "f\text{-}alg"'] \arrow[r, "P'"'] & B\text{-}alg \arrow[u, "f'\text{-}alg"']
\end{tikzcd}$$

Fig. 2. (b)

Let (p, P) be a strongly persistent parameterized data type. By Lemma 3.13 there is exactly one functor P' such that (p', P') is a strongly persistent parameterized data type and the diagram in Fig. 2(b) commutes. (p', P') is called the *extension* of (p, P) via (f, A). Now, standard parameter passing works in such a way that (p, P) sends each actual parameter $(f, A, \mathcal{A})$ to the resultant data type $\mathcal{A}P'$.

The parameter passing mechanism is easily extended to the case where the actual parameter is parametric again: an actual parameter

$$(f: X \to A, \hat{p}: Y \to A, \hat{P}: Y\text{-}alg \to A\text{-}alg),$$

such that $(\hat{p}, \hat{P})$ is a parameterized data type, is sent to the parameterized data type $(\hat{p}p', \hat{P}P')$ where (p', P') is the extension of (p, P) via (f, A). Clearly, if (p, P) and $(\hat{p}, \hat{P})$ are strongly persistent, so is the resultant parameterized data type.

Since we will only need the strong persistency case, we will not discuss other cases here. A satisfactory general theory of parameter passing is still missing.

In order to illustrate the mechanism of parameterized specification and parameter passing, we give a series of examples. These examples will be used in Section 7 again. For specification, we use the following ad hoc syntax: The sorts, operators and equations are listed under keywords **sorts**, **ops** and **eqs**, respectively. The entities of the formal parameter are preceded by the keyword **formal**. Operators may be written in arbitrary mixfix notation [13] with underbars indicating the positions of arguments (otherwise prefix notation is used). In order to have small and clean examples we avoid problems of error and exception handling.

Example 4.6. For each natural number n, the n-fold product

$$P = X_1 \times X_2 \times \cdots \times X_n$$

is the following specification:

sorts P **formal** $X_1, X_2, \ldots, X_n$

ops $\langle\_,\dots,\_\rangle : X_1 \times X_2 \times \cdots \times X_n \to P$

$\_[i] : P \to X_i \qquad (i=1,\dots,n)$

$\_\equiv\_ : P \times P \to \mathrm{BOOL}$

formal $\bar{x}_i : \to X_i \qquad (i=1,\dots,n)$

$\_\equiv\_ : X_i \times X_i \to \mathrm{BOOL} \qquad (i=1,\dots,n)$

eqs $\langle x_1, x_2 \dots, x_n\rangle[i] = x_i \qquad (i=1,\dots,n)$

$\langle x_1,\dots,x_n\rangle \equiv \langle x_1',\dots,x_n'\rangle = (x_1 \equiv x_1' \;\&\cdots\&\; x_n \equiv x_n')$

formal $x_i \equiv x_i = \mathit{true} \qquad (i=1,\dots,n)$

$x_i \equiv x_i' = x_i' \equiv x_i \qquad (i=1,\dots,n)$

$(x_i \equiv x_i' \;\&\; x_i' \equiv x_i'' \Rightarrow x_i \equiv x_i'') = \mathit{true} \qquad (i=1,\dots,n)$

Remarks 4.6.1. (1) Actually, we have denoted different equality operators by the same symbol $\equiv$. Formally, there should be different symbols, but it is easily recognized from the context which one is meant. For notational convenience, we take the same freedom in the following examples.

(2) BOOL is assumed to be interpreted by the booleans *true* and *false*, equipped with appropriate boolean operations like &, $\Rightarrow$ etc. We tacitly assume that there is a specification *BOOL* that has to be added to the above specification and each of the following. By analogy with the formal equations given above, equations concerning equality operators have to be supplemented in later examples. Also we tacitly assume the existence of an operator *if_then_else_fi*: $\mathrm{BOOL} \times D \times D \to D$, for each sort D, with the conventional meaning.

(3) For $n=0$, we simply have one constant $\langle\;\rangle$ of sort P and no equations. This specification is denoted by *0*.

Example 4.7. The n-fold product with constant

$$P = X_1 \times X_2 \times \cdots \times X_n + 1$$

is defined to be $X_1 \times \cdots \times X_n$ extended by

ops $\bar{p} : \to P$

eqs $\bar{p}[i] = \bar{x}_i \qquad (i=1,\dots,n)$

$\bar{p} \equiv \bar{p} = \mathit{true}$

$\bar{p} \equiv \langle x_1,\dots,x_n\rangle = \mathit{false}$

$\langle x_1,\dots,x_n\rangle \equiv \bar{p} = \mathit{false}$

Remark 4.7.1. For $n=0$ we get the specification *0+ 1* consisting of two constants $\langle\;\rangle$ and $\bar{p}$ of sort P.

Example 4.8. The n-fold product with basis Y,

$$P = X_1 \times X_2 \times \cdots \times X_n + Y,$$

is defined to be $X_1 \times \cdots \times X_n$ extended by

sorts **formal** Y

ops $\bar{p}: \quad Y \to P$

formal $\_\equiv\_: Y \times Y \to \text{BOOL}$

eqs $\bar{p}(y)[i] = \bar{x}_i \qquad (i = 1, \ldots, n)$

$\bar{p}(y) \equiv \bar{p}(y') = y \equiv y'$

$\bar{p}(y) \equiv \langle x_1, \ldots, x_n \rangle = \mathit{false}$

$\langle x_1, \ldots, x_n \rangle \equiv \bar{p}(y) = \mathit{false}$

Example 4.9. The specification

$$F = [X \to Y]$$

specifies the finitary functions from X to Y (i.e., functions yielding a constant $\bar{y}$ almost everywhere) as follows:

sorts F **formal** X, Y

ops $\bar{f}: \to F$

$\_[\_] \leftarrow \_: F \times X \times Y \to F$

$\_-\_: F \times X \quad \to F$

$\_[\_]: F \times X \quad \to Y$

$\_\equiv\_: F \times F \quad \to \text{BOOL}$

formal $\_\equiv\_: X \times X \to \text{BOOL}$

$\bar{y}: \to Y$

$\_\equiv\_: Y \times Y \to \text{BOOL}$

eqs $\bar{f} - x = \bar{f}$

$(f(x] \leftarrow y) - x' = \mathit{if}\ x \equiv x' \ \mathit{then}\ f - x'$

$\mathit{else}\ (f - x')[x] \leftarrow y\ \mathit{fi}$

$\bar{f}[x] = \bar{y}$

$(f[x] \leftarrow y)[x'] = \mathit{if}\ x \equiv x'\ \mathit{then}\ y\ \mathit{else}\ f[x']\ \mathit{fi}$

$\bar{f} \equiv \bar{f} = \mathit{true}$

$$\bar{f} \equiv (f[x] \leftarrow y) = \mathit{false}$$

$$(f[x] \leftarrow y)\bar{f} = \mathit{false}$$

$$(f[x] \leftarrow y) \equiv (f'[x'] \leftarrow y')$$

$$= \mathit{if}\ x \equiv x'\ \mathit{then}\ y \equiv y'\ \&\ f - x \equiv f - x'\ \mathit{else}$$

$$f[x'] \equiv y'\ \&\ f'[x] \equiv y\ \&\ f - x - x' \equiv f - x' - x$$

$$\mathit{fi}$$

Example 4.10. The specification
$M = \mathcal{P}(X)$
specifies the finite sets of elements of sort X as follows:

sorts M **formal** X

ops $\emptyset : \quad \rightarrow M$

$\_+\_ : M \times X \rightarrow M$

$\_-\_ : M \times X \rightarrow M$

$\_\in\_ : X \times M \rightarrow \text{BOOL}$

$\_\equiv\_ : M \times M \rightarrow \text{BOOL}$

formal $\_\equiv\_ : X \times X \rightarrow \text{BOOL}$

eqs $\emptyset - x = \emptyset$

$(m + x) - x' = \mathit{if}\ x \equiv x'\ \mathit{then}\ m - x'\ \mathit{else}\ (m - x') + x\ \mathit{fi}$

$x \in \emptyset = \mathit{false}$

$x \in m + x' = x \equiv x' \vee x \in m$

$\emptyset \equiv \emptyset = \mathit{true}$

$m + x \equiv \emptyset = \mathit{false}$

$m + x \equiv m' = x \in m'\ \&\ m - x \equiv m' - x$

$m \equiv m' = m' \equiv m$

In order to illustrate parameter passing, we apply the parameterized data types specified above to the following actual parameter 'natural numbers' N.

sorts N

ops $0 : \quad \rightarrow N$

$\_+1 : N \quad \rightarrow N$

$\_\equiv\_ : N \times N \rightarrow \text{BOOL}$

eqs $$\begin{aligned} 0 \equiv 0 &= \mathit{true} \\ 0 \equiv n+1 &= \mathit{false} \\ n+1 \equiv 0 &= \mathit{false} \\ n+1 \equiv n'+1 &= n \equiv n' \end{aligned}$$

To build (for instance) $N \times N$, we take the actual parameter (f, N) as follows (cf. Example 4.6 for $n=2$):

$$\begin{aligned} f: X_1 &\mapsto N \quad \bar{x}_1 \mapsto 0 \quad \equiv (\text{on } X_1) \mapsto \equiv \\ X_2 &\mapsto N \quad \bar{x}_2 \mapsto 0 \quad \equiv (\text{on } X_2) \mapsto \equiv \end{aligned}$$

The pushout object of the parameterized specification p of Example 4.6 (the inclusion of the formal parameter part in P) and the parameter morphism f then consists of N as given above, supplemented by

ops $$\begin{aligned} \langle \_, \_ \rangle &: N \times N \to P \\ \_[1] &: P \quad \to N \\ \_[2] &: P \quad \to N \end{aligned}$$

eqs $$\begin{aligned} \langle n_1, n_2 \rangle[1] &= n_1 \\ \langle n_1, n_2 \rangle[2] &= n_2 \\ \langle n_1, n_2 \rangle \equiv \langle n_1', n_2' \rangle &= n_1 \equiv n_1' \;\; \& \;\; n_2 \equiv n_2' \end{aligned}$$

Thus, $N \times N$ consists of (a copy of) N plus a copy of $X_1 \times X_2$ where X_1 and X_2 have been replaced by N.

For our example, we can avoid to state the parameter morphism explicitly, if we adopt the following naming convention: each specification D has a distinguished sort named D and, as required from context, possibly a distinguished constant named $\bar{d}$, and a distinguished equality operator named $\equiv$. If we now write, e.g., $D_1 \times D_2 + D_3$, we mean the pushout object of the parametric specification of Example 4.8 and the parameter morphism f sending X_i to D_i, $\bar{x}_i$ to $\bar{d}_i$, and $\equiv$ to $\equiv$, for $i = 1, 2, 3$. So we can denote specifications by expressions. For example,

$N \times N + 1$	consists of N (with distinguished constant 0), new sort P, new constant $\bar{p}$, new operators $\langle \_, \_ \rangle : N \times N \to P$, $\_[1]: P \to N$, $\_[2]: P \to N$, and the equality operator $\_\equiv\_ : P \times P \to$ BOOL. The equations are easily obtained from Examples 4.6 and 4.7.

P, $\bar{p}$ and $\equiv$ are understood to be the distinguished entities that are to be used for further parameter substitution. So the expressions

$$(N \times N + 1) \times (N \times N), [N \times N \to N], \mathcal{P}(N \times N + 1),$$

$$\mathcal{P}([(N \times N) \to (N \times \mathcal{P}(N))]), \qquad \text{etc.}$$

are well-defined specifications.

Similarly, by providing parameterized specifications of the form $X \rightarrow D$ as actual parameters, we get resultant parameterized specifications. For example,

$X \times N + 1$ consists of N plus a copy of $X_1 \times X_2 + 1$ where X_1, $\bar{x}_1$, $\equiv$ have been replaced by X, $\bar{x}$, $\equiv$, and X_2, $\bar{x}_2$, $\equiv$ by N, 0, $\equiv$.

Other examples of expressions for parameterized specifications are the following:

$$(N \times X + 1) \times (X \times N) \quad [X \times X \rightarrow X] \quad \mathcal{P}(X \times Y + 1) \qquad \text{etc.}$$

5. Algebraic domain equations

As explained in the informal introduction in Section 2, we consider algebraic domain equations to be syntactic descriptions of endofunctors of the form $\bar{E}P$ consisting of a forgetful functor $\bar{E}$ and a parameterized data type functor P. We restrict ourselves to strongly persistent parameterizations.

Definition 5.1. An *algebraic domain equation* is a pair of specification morphisms (p, e) with common source and target,

$$\begin{tikzcd} X \arrow[r, shift left, "p"] \arrow[r, shift right, "e"'] & D, \end{tikzcd}$$

such that p is a strongly persistent parameterized specification.

Let $P = p\text{-}free$, $\bar{P} = p\text{-}alg$, and $\bar{E} = e\text{-}alg$. As explained in Section 2, there are two endofunctors that could possibly serve as a semantics of an algebraic domain equation, namely

$$P\bar{E} : X\text{-}alg \rightarrow X\text{-}alg \quad \text{and} \quad \bar{E}P : D\text{-}alg \rightarrow D\text{-}alg.$$

The fixpoints of these endofunctors are very closely related. (We consider fixpoints 'up to isomorphism', i.e., a fixpoint of $F : \mathfrak{K} \rightarrow \mathfrak{K}$ is an object k of $\mathfrak{K}$ such that $kF \cong k$.)

Lemma 5.2. *If A is a fixpoint of $P\bar{E}$, then AP is a fixpoint of $\bar{E}P$. If B is a fixpoint of $\bar{E}P$, then $B\bar{E}$ is a fixpoint of $P\bar{E}$.*

The proof is trivial. Another immediate observation is the following.

Lemma 5.3. *B is a fixpoint of $\bar{E}P$ iff we have $B\bar{E} \cong B\bar{P}$ and $B \cong AP$ for some X-algebra A.*

Proof. If $B\bar{E}P \cong B$, then $B\bar{E} = B\bar{E}P\bar{P} \cong B\bar{P}$, by persistency of P. Conversely, let $B\bar{E} \cong B\bar{P}$ and $B \cong AP$. Then we conclude $B\bar{E}P \cong B\bar{P}P \cong AP\bar{P}P = AP \cong B$. $\square$

We now give an important necessary condition for B to be a fixpoint of $\bar{E}P$, showing that all fixpoints are of a certain special form. This result considerably restricts the range of search for fixpoints. Let (q, Q) be the coequalizer of p and e in *spec*,

$$\begin{tikzcd} X \arrow[r, shift left, "p"] \arrow[r, shift right, "e"'] & D \arrow[r, "q"] & Q, \end{tikzcd}$$

and let $\bar{Q} = q\text{-}alg$ be the forgetful functor associated with q.

Theorem 5.4. *If B is a fixpoint of $\bar{E}P$, then there is a unique (up to isomorphism) Q-algebra C such that $B \cong C\bar{Q}$.*

Proof. If B is a fixpoint of $\bar{E}P$, we have $B\bar{E} \cong B\bar{P}$ by Lemma 5.3. By replacing $B\bar{P}$ by $B\bar{E}$ in B, we construct a D-algebra B' isomorphic to B, such that $B'\bar{E} = B'\bar{P}$. Since by Theorem 3.11 colimits in *spec* carry over to limits in *cat*, $\bar{Q}$ is an equalizer of $\bar{P}$ and $\bar{E}$ in *cat*. This means that there is a unique Q-algebra C such that $C\bar{Q} = B' \cong B$. Since the construction of B' is unique up to isomorphism, the claim of the theorem follows. □

The following characterization of fixpoints of $\bar{E}P$ is an easy consequence of this theorem.

Corollary 5.5. *B is a fixpoint of $\bar{E}P$ iff the following conditions hold:*
(1) *$B \cong AP$ for some X-algebra A,*
(2) *$B \cong C\bar{Q}$ for some Q-algebra C.*

Proof. The 'only if' part is trivial. The 'if' part follows from Lemma 5.3 and $C\bar{Q}\bar{P} = C\bar{Q}\bar{E}$. □

The uniqueness result of Theorem 5.4 shows that fixpoints of $\bar{E}P$ are closely related to corresponding Q-algebras. Motivated by our example in Section 2, we feel that these Q-algebras are adequate candidates for solutions of algebraic domain equations. They are exactly those algebras that result from fixpoints of $\bar{E}P$ if we identify the isomorphic parts $B\bar{P}$ and $B\bar{E}$. So our definition is as follows.

Definition 5.6. Let $(p, e): X \to D$ be an algebraic domain equation, and let $(q, Q) = \text{coeq}(p, e)$. A *solution* of (p, e) is a Q-algebra C such that $C\bar{Q}$ is a fixpoint of $\bar{E}P$.

6. The initial solution

In this section we prove our main result on algebraic domain equations (p, e): $X \to D$: The initial Q-algebra I_Q is a solution of (p, e), where (q, Q) is the coequalizer of p and e in *spec*. Clearly, this solution is initial in the sense that there is exactly one Q-algebra morphism from I_Q to any other solution of (p, e).

Our proof follows the 'classical' construction of least fixpoints as colimits of certain chains. Let $I_n := I_D(\bar{E}P)^n$ and $i_n := i(\bar{E}P)^n$, $n = 0, 1, \ldots$, where i is the unique initial morphism from $I_0 = I_D$ to I_1.

Lemma 6.1. *The colimit D-algebra C of the chain*

$$\begin{tikzcd} I_0 \arrow[r, "i_0"] & I_1 \arrow[r, "i_1"] & I_2 \arrow[r, "i_2"] & \cdots \end{tikzcd}$$

is a fixpoint of $\bar{E}P$.

Proof. Since *D-alg* is cocomplete, the colimit $(C, c_n : I_n \to C)_{n\geq 0}$ of the chain above exists. Since forgetful functors and left adjoints respect colimits of chains [23], $(C\bar{E}P, c_n\bar{E}P)_{n\geq 0}$ is a colimit of that chain with the first step omitted. But adding the unique initial morphisms we get a colimit of the original chain with colimit object $C\bar{E}P$. Since colimits are unique up to isomorphism, we have $C\bar{E}P \cong C$. □

According to Lemma 5.3, a fixpoint B of $\bar{E}P$ satisfies $B\bar{E} \cong B\bar{P}$. Let *iso* denote the category of all pairs (B, β) where B is a D-algebra and β is an X-algebra isomorphism from $B\bar{E}$ to $B\bar{P}$, together with morphisms $f: (B_1, \beta_1) \to (B_2, \beta_2)$ that are defined to be D-algebra morphisms $f: B_1 \to B_2$ such that $f\bar{E} \circ \beta_2 = \beta_1 \circ f\bar{P}$.

Let *equ* be the full subcategory of *iso* given by all pairs $(B, \mathrm{id}_{B\bar{E}})$, i.e., those algebras B such that $B\bar{E} = B\bar{P}$. Clearly, *equ* is isomorphic to the subcategory of *D-alg* consisting of all D-algebras B satisfying $B\bar{E} = B\bar{P}$ and of all D-algebra morphisms f satisfying $f\bar{E} = f\bar{P}$. This subcategory will also be denoted by *equ*.

Again, let $\bar{Q} = q\text{-}alg : Q\text{-}alg \to D\text{-}alg$ denote the forgetful functor associated with the coequalizer (q, Q) of (p, e).

Lemma 6.2. *Q-alg and equ are isomorphic as categories, such that* $\bar{Q}$ *(with its range restricted) is an isomorphism.*

Proof. Obviously, *equ* (with the inclusion in *D-alg*) is an equalizer of $\bar{E}$ and $\bar{P}$ in *cat*. According to Lemma 3.11, $(Q\text{-}alg, \bar{Q})$ is an equalizer of $\bar{E}$ and $\bar{P}$, too. Since $\bar{Q}\bar{E} = \bar{Q}\bar{P}$, the image under $\bar{Q}$ is even contained in *equ*. Because of the equalizer property, $\bar{Q}$ is an isomorphism. □

Lemma 6.3. *equ and iso are equivalent as categories; an equivalence is given by the inclusion equ* $\subseteq$ *iso.*

Proof. All we have to show is that each object in *iso* is isomorphic to some object in *equ*. The construction is sketched in the proof of Theorem 5.4. □

Now we are in a position to prove our main result.

Theorem 6.4. *I_Q is a solution.*

Proof. Considering the chain of Lemma 6.1, let $(C, c_n : I_n \to C)_{n\geq 0}$ be a colimit. We have seen in the proof of Lemma 6.1 that $(C\bar{E}P, c_n\bar{E}P)_{n\geq 0}$ is another colimit of this chain. Thus, there is exactly one isomorphism $\gamma : C\bar{E}P \to C$ satisfying

(a) $\quad c_{n+1} = c_n\bar{E}P \circ \gamma \quad$ for all $n \geq 0$.

We prove that $(C, \gamma\bar{P})$ is an initial object in *iso*. This implies that, by Lemma 6.3, there is an initial object C' in *equ* that is isomorphic to C, and by Lemma 6.2 we conclude that $I_Q\bar{Q}$ is isomorphic to C', since isomorphisms respect initiality. Thus $I_Q\bar{Q}$ is also a fixpoint of $\bar{E}P$, i.e., I_Q is a solution.

In order to prove that $(C, \gamma\bar{P})$ is initial in *iso*, let (B, β) be an object in *iso*, with $\beta : B\bar{E} \to B\bar{P}$.

For each n, we construct a morphism $b_n : I_n \to B$ in *D-alg* as follows: b_0 is the unique initial morphism, and b_{n+1} is uniquely determined by

(b) $\quad b_{n+1}\bar{P} = b_n\bar{E} \circ \beta\colon \quad I_{n+1}\bar{P} = I_n\bar{E}P\bar{P} = I_n\bar{E} \to B\bar{P}$,

since P is left adjoint and strongly persistent. By induction, we see that

(c) $\quad b_n = i_n \circ b_{n+1} \quad$ for all $n \geq 0$,

because $b_0 = i_0 \circ b_1$, by initiality of I_0, and

$$(i_{n+1} \circ b_{n+2})\bar{P} = i_{n+1}\bar{P} \circ b_{n+2}\bar{P} = i_n\bar{E} \circ b_{n+1}\bar{E} \circ \beta = b_n\bar{E} \circ \beta = b_{n+1}\bar{P}.$$

Since $(C, c_n : I_n \to C)_{n\geq 0}$ is a colimit of our chain, there is a unique D-algebra morphism $h : C \to B$ satisfying

(d) $\quad c_n \circ h = b_n \quad$ for all $n \geq 0$.

The last step of the proof is to show that (d) is equivalent to saying that h is a morphism in *iso*, i.e., that the following holds:

(e) $\quad \gamma\bar{P} \circ h\bar{P} = h\bar{E} \circ \beta$.

Equivalence of (d) and (e) implies that there is a unique morphism $h : (C, \gamma\bar{P}) \to (B, \beta)$ in *iso*, i.e., $(C, \gamma\bar{P})$ is initial in *iso*.

Now we prove this equivalence.

(d)$\Rightarrow$(e). Since P preserves colimits of chains and $P\bar{P} = id$, $(C\bar{E}, c_n\bar{E})_{n\geq 0}$ is colimit of the chain $i_0\bar{P}, i_1\bar{P}, \ldots$. From (d) and (a) we conclude

$$b_{n+1}\bar{P} = c_{n+1}\bar{P} \circ h\bar{P} = c_n\bar{E} \circ (\gamma\bar{P} \circ h\bar{P}), \quad n \geq 0,$$

and from (b) and (d) we get

$$b_{n+1}\bar{P} = b_n\bar{E} \circ \beta = c_n\bar{E} \circ (h\bar{E} \circ \beta), \quad n \geq 0.$$

Because of the colimit property, (e) follows.

(e)$\Rightarrow$(d). This is proven by induction on n. For $n=0$, we have $c_0 \circ h = b_0$ by initiality of I_0. Let $c_n \circ h = b_n$. Then from (a) and (b) we get

$$\begin{aligned}(c_{n+1} \circ h)\bar{P} &= c_{n+1}\bar{P} \circ h\bar{P} = c_n\bar{E} \circ \gamma\bar{P} \circ h\bar{P} = c_n\bar{E} \circ h\bar{E} \circ \beta \\ &= b_n\bar{E} \circ \beta = b_{n+1}\bar{P}.\end{aligned}$$

Since P is left adjoint to $\bar{P}$, we have $c_{n+1}h = b_{n+1}$. Thus, (d) holds for all $n \geqslant 0$. $\square$

An alternative proof of our main theorem could be based on a result of Merzenich [19, Corollary 3.22] about initial fixpoints of functors.

7. Examples

We give a series of examples to demonstrate how algebraic domain equations can be used as a specification tool. The examples should provide enough evidence for the elegance that can be achieved this way in certain cases, considerably reducing the amount of notational detail.

In the examples to follow, the formal parameter X always has one sort X and an equality operator $\equiv : X \times X \to \text{BOOL}$, and sometimes a constant $\bar{x}$: $\to X$. We also have the specification *BOOL* as part of the formal parameter as well as the usual equations for the equality operator.

We also adopt the naming convention of Section 4: each specification D has a distinguished sort D and, as required from context, a constant $\bar{d}$, and an equality operator $\equiv : D \times D \to \text{BOOL}$. These entities serve as standard values for parameter assignment.

An algebraic domain equation will be written as

$$X \Longrightarrow D.$$

The symbol $\Longrightarrow$ is shorthand for a pair (p, e) of specification morphisms from X to D. p is a parameterized specification embedding X into D. The data specifying p are implicitly given by equal denotation. e maps the formal entities X, $\bar{x}$, $\equiv$ to the corresonding entities of D, namely D, $\bar{d}$, $\equiv$.

According to our main Theorem 6.4, an equation (p, e) 'specifies' a data type (algebra) A iff $A \cong I_Q$ where Q is the coequalizer object of p and e.

For the examples to follow, we refer to the parameterized specifications of Examples 4.6 to 4.10 and to the notation used there. For comparison we refer to [25, 30].

Example 7.1. The algebraic domain equation

$$X \Longrightarrow X + 1$$

(cf. Example 4.7) specifies the natural numbers. The syntactic solution can be derived from the specification of $X + 1$ as follows. We have to identify the sorts X and P, and, for convenience, we rename this sort by N; the operators are handled as indicated below (equality operator omitted):

$$\begin{array}{lll}
\textbf{sorts} & N & (X, P) \\
\textbf{ops} & 0 \colon \to N & (\bar{x}, \bar{p}) \\
 & \text{succ} \colon N \to N & (\langle \_ \rangle) \\
 & \text{pred} \colon N \to N & (\_[1]) \\
\textbf{eqs} & \text{pred}(0) = 0 & \\
 & \text{pred}(\text{succ}(n)) = n &
\end{array}$$

Example 7.2. Given some specification *A* with distinguished sort A, constant $\bar{a}$, and equality operator $\equiv$,

$$X \Longrightarrow X \times A + 1$$

specifies stacks with entries of sort A. The syntactic solution is the following explicit specification where sorts X and P have been renamed by S, and the operators have been renamed as follows: $\bar{x}$, $\bar{p}$ by empty, $\langle \_, \_ \rangle$ by push, $\_[1]$ by pop, and $\_[2]$ by top (cf. Example 4.7):

$$\begin{array}{ll}
\textbf{sorts} & \text{BOOL}, A, S \\
\textbf{ops} & \text{empty}\colon \quad \to S \\
 & \text{push}\colon S \times A \to S \\
 & \text{pop}\colon \quad S \to S \\
 & \text{top}\colon \quad S \to A \\
 & \equiv\colon S \times S \to \text{BOOL} \\
 & \cdots \text{ (additional ops of } \textit{BOOL} \text{ and } A) \\
\textbf{eqs} & \text{pop}(\text{empty}) = \text{empty} \\
 & \text{top}(\text{empty}) = \bar{a} \\
 & \text{pop}(\text{push}(s, a)) = s \\
 & \text{top}(\text{push}(s, a)) = a \\
 & \text{empty} \equiv \text{empty} = \mathit{true} \\
 & \text{empty} \equiv \text{push}(s, a) = \mathit{false} \\
 & \text{push}(s, a) \equiv \text{empty} \qquad = \mathit{false} \\
 & \text{push}(s, a) \equiv \text{push}(s', a') = s \equiv s' \ \& \ a \equiv a' \\
 & \cdots \text{ (additional eqs of } \textit{BOOL} \text{ and } A)
\end{array}$$

Example 7.3. The solution of

$$X \Longrightarrow X \times X \times A + 1$$

is the type of binary trees with data of sort A attached to each node. As operators, we have the empty tree ($\bar{p}$), construct tree ($\langle \_, \_, \_ \rangle$), left subtree ($\_[1]$), right subtree ($\_[2]$) and node contents ($\_[3]$).

Example 7.4. The algebraic domain equation

$$X \Longrightarrow X \times X + A$$

(cf. Example 4.8) specifies binary trees with data of sort A attached to the leaves only. The details are left to the reader.

Example 7.5. We obtain a solution for

$$X \Longrightarrow [X \to X]$$

(cf. Example 4.9). Of course, it does not give an analogon to Scott's [26] famous reflexive domain that could serve as a model for pure λ-calculus. We cannot expect that since in our case only finitary functions are involved. The solution consists of finitary functions taking finitary functions as arguments and having finitary functions as values. The operators retain their meanings as empty function ($\bar{f}$), value assignment ($\_[\_] \leftarrow \_$), function application ($\_[\_]$) and equality of functions ($\equiv$). Each element in the carrier can be applied to any other element in the carrier, including itself.

We, however, only have a trivial form of self-application, i.e., we always have $f[f] = \bar{f}$. Intuitively, this follows from the finite term representation for f: Only arguments 'smaller' than f can yield nontrivial values.

Example 7.6. The finitary 'powerdomain constructor' $\mathcal{P}$ described in Example 4.10 gives rise to an equation

$$X \Longrightarrow \mathcal{P}(X).$$

The solution consists of finite sets whose members are finite sets whose members are finite sets ... etc. (only finite depth). The operations retain their set theoretic meaning of empty set ($\emptyset$), inserting or deleting an element ($+, -$), set membership ($\in$) and equality of sets ($\equiv$).

Example 7.7. Let $Stack(X)$ be the parameterized stack specification with formal parameter X obtained from the explicit specification given in Example 7.2 by substituting X, $\bar{x}$ for A, $\bar{a}$. Then, the solution of

$$X \Longrightarrow Stack(X)$$

describes binary trees. It is isomorphic to the solution of

$$X \Longrightarrow X \times X + 1.$$

It is obvious that each specification D is the syntactic solution of some algebraic domain equation, e.g., $(\emptyset, \emptyset): \emptyset \to D$. Thus, explicit specification can be considered as a special case of implicit specification by algebraic domain equations. It is conceivable to use mixed forms, giving part of a specification implicitly and the rest explicitly. We do not pursue the subject here; to sketch the idea we give a simple example, a specification of queues.

Example 7.8. We informally use the word symbol **base** to denote an implicit specification that is extended by subsequent explicit specifications. The operators in the base specification are assumed to be hidden.

A queue with elements of sort A can then be specified as follows:

base $X \Longrightarrow X \times A + 1$

rename P **by** Q, $\bar{P}$ **by** empty, $\langle\_,\_\rangle$ **by** in

extended by

ops out: $Q \to Q$

front: $Q \to A$

eqs $\text{out}(q) = \mathit{if}\ \ q(1) \equiv \text{empty}\ \ \mathit{then}\ \ \text{empty}$

$\qquad \mathit{else}\ \ \text{in}(\text{out}(q(1), q[2])\ \ \mathit{fi}$

$\text{front}(q) = \mathit{if}\ \ q(1) \equiv \text{empty}\ \ \mathit{then}\ \ q[2]$

$\qquad \mathit{else}\ \ \text{front}\,(q[1])\ \ \mathit{fi}$

This comes close to an operational specification of queues as lists of objects of sort A, with operations empty, in, and the hidden projections as standard list operations, as well as with out and front defined recursively in terms of these.

8. Parameterized algebraic domain equations

In Example 7.2 we had to assume a fixed specification A in order to specify stacks over A implicitly. Looking at this example, the question naturally arises whether it is possible to view A as a formal parameter getting 'stacks of something' as a solution of a parameterized algebraic domain equation. Inspection of the examples in the last section gives the impression that the answer is positive, and that the same method of syntactic solution should apply. In this section we show that this works.

First we have to make precise what parameterization of algebraic domain equations means. A useful notion evolves from the intuition of a 'formal parameter Y of the formal parameter X' which is in a sense not affected by solving the algebraic domain equation.

Definition 8.1. A *parameterized algebraic domain equation* is a diagram in **spec** of the form

$$\begin{tikzcd} Y \arrow[r, "r"] & X \arrow[r, shift left, "p"] \arrow[r, shift right, "e"'] & D \end{tikzcd}$$

such that (i) (p, e) is an algebraic domain equation, (ii) r is a parameterized specification, and (iii) $rp = re$. We use the notation $(r; p, e)$ if the objects are clear from the context. $(r; p, e)$ is called *(strongly) persistent* iff r has this property.

We tacitly assume in the sequel that all parameterized algebraic domain equations are strongly persistent. From a parameterized algebraic domain equation we obtain non-parameterized actual instances by an adaptation of the parameter passing mechanism described in Section 4.

Definition 8.2. Let $(r; p, e)$ be a parameterized algebraic domain equation and let (f, A) be an actual parameter for r. Then, the *(f, A)-instance* of $(r; p, e)$ is the algebraic domain equation (p', e') obtained as follows (cf. Fig. 3):

(1) Let r', B, f_1 be such that (1) is a pushout.

(2) Let p', C, f_2 be such that (2) is a pushout, i.e., (p', f_2) is a pushout of (p, f_1).

(3) Let e' be the unique morphism defined by $r'e' = r'p'$ and $f_1e' = ef_2$. (e' is well defined since (1) is a pushout and we have $fr'p' = rf_1p' = rpf_2 = ref_2$.)

$$\begin{tikzcd} Y \arrow[r, "r"] \arrow[d, "f"'] \arrow[dr, phantom, "(1)"] & X \arrow[r, shift left, "p"] \arrow[r, shift right, "e"'] \arrow[d, "f_1"'] \arrow[dr, phantom, "(2)"] & D \arrow[d, "f_2"] \\ A \arrow[r, "r'"'] & B \arrow[r, shift left, "p'"] \arrow[r, shift right, "e'"'] & C \end{tikzcd}$$

Fig. 3.

Now, given a parameterized algebraic domain equation $(r; p, e)$ and an actual parameter (f, A), we can solve its (f, A)-instance (p', e') by the method explained in Section 6. If (q', Q') is the coequalizer of p' and e', we thus have a mapping sending I_A (the actual parameter semantics) to $I_{Q'}$ (the solution of (p', e')). A solution of the equation $(r; p, e)$ is defined to be a parameterized data type producing this mapping by standard parameter passing.

Definition 8.3. Let $(r;p,e)$ be a parameterized algebraic domain equation. A parameterized data type (s, S) is a *solution* of $(r;p,e)$ iff, for each actual parameter (f, A), the solution of the (f, A)-instance of $(r;p,e)$ is isomorphic to the result of applying (s, S) to (f, A, I_A) using standard parameter passing.

Of course, in order for this definition to make sense, (s, S) must be applicable to (f, A, I_A) iff $(r;p,e)$ is applicable to (f, A), i.e., the source of s must be Y. Our main Theorem 6.4 now extends neatly to the parameterized case.

Theorem 8.4. *Let $(r\colon Y\to X;\ p, e\colon X\to D)$ be a strongly persistent parameterized algebraic domain equation, and let $(q, Q) = \mathrm{coeq}(p,e)$. Then $s = rpq\colon Y\to Q$, together with its standard semantics $S = s$-free, is a solution of $(r;p,e)$.*

Proof. We extend Fig. 3, where the squares (1) and (2) are defined as in Definition 8.2, by a new square (3) as follows (cf. Fig. 4). (q, Q) is the coequalizer of p and e, and (q', Q') is the coequalizer of p' and e'. Since $pf_2q' = f_1p'q' = f_1e'q' = ef_2q'$, there is exactly one $f_3\colon Q\to Q'$ such that $qf_3 = f_2q'$. We choose this f_3 to complete the diagram.

$$\begin{tikzcd}
Y \arrow[r, "r"] \arrow[d, "f"'] \arrow[dr, phantom, "(1)"] & X \arrow[r, shift left, "p"] \arrow[r, shift right, "e"'] \arrow[d, "f_1"] \arrow[dr, phantom, "(2)"] & D \arrow[r, "q"] \arrow[d, "f_2"] \arrow[dr, phantom, "(3)"] & Q \arrow[d, "f_3"] \\
A \arrow[r, "r'"'] & B \arrow[r, shift left, "p'"] \arrow[r, shift right, "e'"'] & C \arrow[r, "q'"'] & Q'
\end{tikzcd}$$

Fig. 4.

All we have to show is that (3) is a pushout. By well-known pushout theorems, we then may conclude that the large square (1, 2, 3) is a pushout, too. Thus, $I_{Q'}$ is the solution of (p', e') as well as the result of applying (s, S) to (f, A, I_A) by standard parameter passing, and this proves the theorem.

In order to prove that (3) is a pushout, let $g_1\colon Q\to R$ and $g_2\colon C\to R$ be such that $qg_1 = f_2g_2$. It follows that $pf_2g_2 = pqg_1 = eqg_1 = ef_2g_2$. Since $(q, Q) = \mathrm{coeq}(p, e)$,

(i) there is exactly one h such that $qh = f_2 g_2$, namely $h = g_1$.
Now we have $f_1 p' g_2 = p f_2 g_2 = e f_2 g_2 = f_1 e' g_2$ and $r' p' g_2 = r' e' g_2$. From the pushout property of (1) we conclude that $p' g_2 = e' g_2$. Since (q', Q') is coequalizer of p' and e',

(ii) there is exactly one k such that $q' k = g_2$.
For this k, because of commutativity of (3), we have $q f_3 k = f_2 q' k = f_2 g_2$. From (i) we conclude that

(iii) $f_3 k = g_1$.
Expressions (ii) and (iii) prove the square (3) to be a pushout. □

Examples of implicit parameterized specifications can easily be obtained from the examples in the previous sections by substituting Y for A and declaring Y as the formal parameter. So we get

$X \Longrightarrow X \times Y + 1$	stacks over Y,
$X \Longrightarrow X \times X \times Y + 1$	trees over Y (entries at each node),
$X \Longrightarrow X \times X + Y$	trees over Y (entries at the leaves only),

etc.

9. Conclusions

The examples given in Section 7 should provide enough evidence that algebraic domain equations are useful to consider as an additional specification technique, especially in connection with a selected set of parameterized data types as type constructors, like those given in Section 4. Our theory shows that there is a sound and consistent semantics behind this method, if only (strongly) persistent parameterizations are used. On the syntactic level, there is a simple method of solving algebraic domain equations.

This paper concentrates on theoretical aspects. The feasibility and usefulness of the results for the development of specification methods and languages should be subject to further study.

Another possible area of application is the algebraic semantics of programming languages. In denotational semantics, domain equations are extensively used to specify syntactic and semantic domains. Our theory can provide algebraic interpretations for them. There is, however, one difficulty: we get only 'finitary' solutions, for instance, the algebras of finite functions or finite sets. The central semantic domains of environments and states typically consist of finite functions, so there seems to be no problem. It is, however, not quite clear how to cope with cases like procedure parameters. A solution of this problem may require the extension of our theory to continuous algebras as introduced in [1]. This is subject to further study.

Acknowledgment

We have taken great benefits from various discussions with J. Thatcher, E. Wagner, J. Wright, H. Ehrig, H.-J. Kreowski, M. Arbib and E. Manes. Especially helpful was E. Manes' criticism on an earlier draft of this paper. Of course, we take full responsibility for all remaining deficiencies.

References

[1] J.A. Goguen, J.W. Thatcher, E.G. Wagner and J.B. Wright, Initial algebra semantics and continuous algebras, *J. ACM* **24** (1977) 68–95.

[2] J.A. Goguen, J.W. Thatcher and E.G. Wagner, An initial algebraic approach to the specification, correctness, and implementation of abstract data types, in: R.T. Yeh, ed., *Current Trends in Programming Methodology, Vol. IV* (Prentice-Hall, Englewood Cliffs, NJ, 1978) pp. 80–149.

[3] J.W. Thatcher, E.G. Wagner and J.B. Wright, Data type specification: Parameterization and the power of specification techniques, *Proc. 10th SIGACT STOC*, San Diego (1978) 119–132; revised version: IBM Res. Rept. RC 7757, 1979.

[4] M.A. Arbib and E.G. Manes, *Arrows, Structures, and Functors* (Academic Press, New York, 1975).

[5] R.M. Burstall and J.A. Goguen, Putting theories together to make specifications, *Proc. 5th Internat. Joint Conf. on Artificial Intelligence* (MIT, Cambridge, MA, 1977) pp. 1045–1058.

[6] R.M. Burstall and J.A. Goguen, The semantics of CLEAR, a specification language, in: D. Bjørner, ed., Lecture Notes in Computer Science **86** (Springer, Berlin, 1980) pp. 292–331.

[7] H.-D. Ehrich, On the theory of specification, implementation, and parameterization of abstract data types, *J. ACM* **29** (1982) 206–227; also: Bericht No. 82, Abteilung Informatik, Univ. Dortmund, 1979.

[8] H.-D. Ehrich, Specifying algebraic data types by domain equations, in: F. Gecseq, ed., *Proc. 1981 FCT-Conf.*, Lecture Notes in Computer Science **117** (Springer, Berlin, 1981) pp. 120–129.

[9] H.-D. Ehrich and V.G. Lohberger, Parametric specification of abstract data types, parameter substitution, and graph replacements, in: M. Nagl and H.-J. Schneider, eds., *Graphs, Data Structures, Algorithms*, Applied Computer Science **13** (Hanser, München, 1979) pp. 169–182.

[10] H.-D. Ehrich and V.G. Lohberger, Constructing specifications of abstract data types by replacements, in: V. Claus, H. Ehrig and G. Rozenberg, eds., *Proc. Internat. Workshop on Graph Grammars and Their Application to Computer Science and Biology*, Lecture Notes in Computer Science **73** (Springer, Berlin, 1979) pp. 180–191.

[11] H. Ehrig, H.-J. Kreowski, J.W. Thatcher, E.G. Wagner and J.B. Wright, Parameterized data types in algebraic specification languages, in: J.W. de Bakker and J. van Leeuwen, eds., *Proc. 7th ICALP*, Lecture Notes in Computer Science **85** (Springer, Berlin, 1980) pp. 157–168.

[12] H. Ehrig, H.-J. Kreowski, J.W. Thatcher, E.G. Wagner and J.-B. Wright, Parameter passing in algebraic specification languages, in: J. Staunstrup, ed., *Proc. Workshop on Program Specification* 1981, Lecture Notes in Computer Science **134** (Springer, Berlin, 1982) pp. 322–369.

[13] J.A. Goguen, Some design principles and theory for OBJ-O, in: E.K. Blum et al., eds., *Proc. Math. Studies of Information Processing*, Lecture Notes in Computer Science **75** (Springer, Berlin, 1979) pp. 425–475.

[14] J.V. Guttag, The specification and application to programming of abstract data types, Tech. Rept. CSRG-59, Univ. of Toronto, 1975.

[15] A. Kanda, Data types as initial algebras: A unification of Scottery and ADJery, *Proc. 19th FOCS* (1978) pp. 221–230.

[16] A. Kanda, Fully effective solutions of recursive domain equations, in: J. Becvar, ed., *Proc. 8th MFCS*, Lecture Notes in Computer Science **74** (Springer, Berlin, 1979) pp. 326–336.

[17] D.J. Lehmann, Categories for fixpoint semantics, *Proc. 17th IEEE FOCS*, Houston (1976) pp. 122–126.

[18] D.J. Lehmann and M.B. Smyth, Data types, *Proc. 18th IEEE FOCS*, Providence (1977) pp. 7–12.
[19] W. Merzenich, Allgemeine Operatornetze als Fixpunktgleichungen, Habilitationsschrift, Bericht Nr. 97, Abteilung Informatik, Univ. Dortmund, 1980.
[20] P. Mosses, Modular denotational semantics, Internal Rept., Univ. of Aarhus, 1979.
[21] G.D. Plotkin, A powerdomain construction, *SIAM J. Comput.* **5** (1976) pp. 452–487.
[22] G.D. Plotkin, Π^{ω} as a universal domain, *J. Comput. System Sci.* **17** (1978) 209–236.
[23] H. Schubert, *Kategorien I, II* (Springer, Berlin, 1970).
[24] D.S. Scott, The lattice of flow diagrams, in: E. Engeler, ed., *Proc. Symp. on Semantics of Algorithmic Languages*, Lecture Notes in Mathematics **188** (Springer, Berlin, 1971) pp. 311–372.
[25] D.S. Scott, Lattice theory, data types and semantics, in: R. Rustin, ed., *Formal Semantics of Algorithmic Languages* (Prentice-Hall, Englewood Cliffs, NJ, 1972) pp. 65–106.
[26] D.S. Scott, Continuous lattices, in: F.W. Lawvere, ed., *Toposes, Algebraic Geometry and Logic*, Lecture Notes in Mathematics **274** (Springer, Berlin, 1972) pp. 97–136.
[27] D.S. Scott, Data types as lattices, *SIAM J. Comput.* **5** (1976) 522–587.
[28] M.B. Smyth, Powerdomains, in: A. Mazurkiewicz, ed., *Proc. 5th MFCS*, Lecture Notes in Computer Science **45** (Springer, Berlin, 1976) pp. 537–543.
[29] M.B. Smyth, Effectively given domains, *Theoret. Comput. Sci.* **5** (1977) 257–274.
[30] M.B. Smyth and G.D. Plotkin, The category-theoretic solution of recursive domain equations, *Proc. 18th IEEE FOCS* (1977) pp. 13–17.
[31] M. Wand: On the recursive specification of data types, in: E.G. Manes, ed., *Proc. 1st Internat. Coll. on Category Theory Applied to Computation and Control*, Lecture Notes in Computer Science **25** (Springer, Berlin, 1975) pp. 214–217.
[32] M. Wand, Fixed point constructions in order-enriched categories, Tech. Rept. No. 23, Computer Science Dept., Indiana Univ., Bloomington, 1975.