Toward formal development of programs from algebraic specifications: implementations revisited[1]

Donald Sannella[2] and Andrzej Tarlecki[3]

Abstract

The program development process is viewed as a sequence of implementation steps leading from a specification to a program. Based on an elementary notion of refinement, two notions of implementation are studied: constructor implementations which involve a construction "on top of" the implementing specification, and abstractor implementations which additionally provide for abstraction from some details of the implemented specification. These subsume most formal notions of implementation in the literature. Both kinds of implementations satisfy a vertical composition and a (modified) horizontal composition property. All the definitions and results are shown to generalise to the framework of an arbitrary institution, and a way of changing institutions during the implementation process is introduced. All this is illustrated by means of simple concrete examples.

1 Introduction

Probably the most exciting potential application of formal specifications is to the formal development of programs by gradual refinement from a high-level specification to a low-level "program" or "executable specification" as in HOPE [BMS 80], Standard ML [Mil 85] or OBJ2 [FGJM 85]. Each refinement step embodies some design decisions (such as choice of data representation or algorithm). If each refinement step can be proven correct, then the program which results is guaranteed to satisfy the original specification.

In order to make this dream a reality, we need at least two things. The first is a theory of formal specifications and the second is an adequate notion of refinement or implementation step. A theory of specifications may be built upon the pioneering work of [GTW 76], [Gut 75] and [Zil 74] on algebraic specifications. It seems especially important to pay attention to the problem of building specifications in a structured way (as in CLEAR [BG 77,80], CIP-L [Bau 81a], LOOK [ETLZ 82], Larch [GH 83], etc.) and to the possibility of using different logical systems (or *institutions* [GB 84a,86]) to write specifications (as in CLEAR, ASL [ST 86a] or Extended ML [ST 86b]).

There has been a lot of interesting work done on notions of refinement as well (see e.g. [GTW 76], [GB 80], [Ehr 81], [Ehr 82], [EKMP 82], [EK 82], [SW 82], [GM 82], [Sch 86], [BMPW 82], [Gan 83], [Lip 83], [BBC 86], [Wand 82]). In [SW 83] and then in [ST 85b,87a] we suggested and used a very simple notion of specification refinement which seems appropriate for loose specifications. Namely, we

---

[1] An extended abstract of this paper appeared in [ST 87b].

[2] Department of Artificial Intelligence, University of Edinburgh and Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh

[3] Institute of Computer Science, Polish Academy of Sciences, Warsaw

say that a specification SP *refines to* a specification SP', written $SP \leadsto SP'$, if every model of SP' is a model of SP. This extends to a notion of refinement of parameterised specifications. In order to be useful for stepwise and modular program development, implementations should compose *vertically* (i.e. $SP \leadsto SP' \leadsto SP''$ should imply $SP \leadsto SP''$) and *horizontally* (i.e. $P \leadsto P'$ and $SP \leadsto SP'$ should imply $P(SP) \leadsto P'(SP')$ for parameterised specifications P,P') [GB 80]. Our simple notion of refinement composes both vertically and horizontally.

This looks suspiciously oversimplified, especially in comparison with most previous work in this area. This is very much in line with our approach to specification languages, however. In [SW 83] and [ST 86a] we presented a set of *kernel* specification-building operations as a basis for defining higher-level and more user-friendly specification languages. In the same sense, the above simple notion of refinement is a kernel notion with respect to the concept of implementation. In this paper we elaborate on how it can provide a basis for realistic and non-trivial program development.

Roughly speaking, one would expect an implementation of a specification SP by another specification SP' to consist of a "program" or construction written in terms of SP' to compute the functions specified in SP. Under a suitable formalisation of the notion of a construction, we say that a specification SP *is implemented by* a specification SP' *via* a construction κ , written $SP \underset{\kappa}{\leadsto} SP'$, if $SP \leadsto \kappa(SP')$. That is, SP refines not to SP' directly, but to a specification consisting of the construction κ "on top of" SP'. This is close to the notions of implementation in [Ehr 81] and [Lip 83] and subsumes most previous notions of implementation in the literature, e.g. [GTW 76], [Ehr 82], [EKMP 82] and [SW 82] (since some of these, including that of [Ehr 81], were defined as implementations of one single algebra by another, and some others, like [BBC 86], were formulated in a more syntactic, proof-oriented style, it may be more accurate to say that this subsumes the model-theoretic ideas behind these notions). It is easy to show that such implementations compose vertically; if we extend this notion of implementation to parameterised specifications we obtain a (modified but perfectly satisfactory) horizontal composition theorem as well. Both composability results hold without any further assumptions, which comes from the fact that we do not require (and see no reason to require) that the composed implementation should take exactly the same form as its component implementations in contrast to e.g. [Ehr 82], [EKMP 82], [EK 82] and [SW 82].

It may be argued that an implementation $SP \underset{\kappa}{\leadsto} SP'$ should be correct as long as the construction κ yields functions which "behave like" those specified in SP instead of being exactly the same. This suggests that ideas concerning *behavioural equivalence* of algebras as discussed in [GGM 76], [Rei 81], [GM 82], [ST 87a] (and elsewhere) should be explored in the context of implementations. Under a suitable formalization of the notion of an abstraction based on an equivalence on algebras, we say that a specification SP *is implemented by* a specification SP' *with respect to* an abstraction α *via* a construction κ, written $SP \overset{\alpha}{\underset{\kappa}{\leadsto}} SP'$, if $\alpha(SP) \leadsto \kappa(SP')$. That is, SP' implements SP (via κ) not "exactly" but only up to the abstraction equivalence associated with α. If this equivalence is the behavioural equivalence relation then this subsumes the notions of implementation in [GM 82], [Sch 86] and [BMPW 86]; other equivalence relations may be useful as well.

Now vertical composition is non-trivial, depending essentially on the requirement that the constructions used preserve the relevant abstraction equivalences. We show that this is the case under certain conditions for each of the constructions we consider to demonstrate that it is not an unreasonable requirement. This notion of implementation extends to parameterised specifications as before. However, horizontal composition holds only for parameterised specifications which preserve the abstraction equivalence (extended to specifications). This turns out to be quite satisfactory in practice and an example shows that a stronger result cannot really be expected.

We present the above ideas in the framework of partial algebras (with first-order formulae) [BrW 82]. This is mainly to take advantage of the reader's intuition, since all of the main definitions and results as well as methodological remarks may be directly restated in the framework of an arbitrary institution [GB 84a]. This means that they can be used to develop programs from specifications in a wide variety of logical systems which involve different notions of signature, logical formula, and model (examples include the standard framework of equational logic and total algebras as well as higher-order logics, LCF [Plo 77], error algebras [GDLE 84] and many others). Thus, a user of the presented program development methodology may choose the logical system which is most suited to his particular task. Moreover, different logical systems may be most suitable at different stages of the development of even a single program, for example when developing an efficient imperative program from a high-level algebraic specification. We enable this by allowing specifications to be implemented by specifications in a different institution using what we call a *semi-institution morphism*.

The concepts we introduce are illustrated by a running example of the implementation of sets of natural numbers.

We assume some familiarity with a few notions from basic category theory, although no use is made of any deep results. See [AM 75] or [MacL 71] for the definitions of e.g. category, initial object, pushout, pullback, functor etc. which we omit here.

2 Algebraic preliminaries

Most of the following definitions are more or less standard so we give them without comment or motivation; for a more detailed presentation see [GTW 76], [BG 82], [EM 85] for total algebras and [BrW 82], [Bur 86] for partial algebras.

Notation Throughout this paper we deal with many-sorted sets, functions, relations, etc. (for any set S, an S-sorted set is just a family $X = \{X_s\}_{s\in S}$ of sets indexed by S, and similarly for functions, relations, etc.). We will feel free to use standard set-theoretic notation without explicit use of indices: for example, we write $x \in X$ rather than $x \in X_s$ for some $s \in S$, and $h\colon X \to Y$ rather than $h = \{h_s\}_{s\in S}$ and $h_s\colon X_s \to Y_s$ for $s \in S$, etc.

A *signature* is a pair $\langle S, \Omega\rangle$ where S is a set (of sort names) and Ω is a family of sets $\{\Omega_{w,s}\}_{w\in S^*, s\in S}$ (of operation names). We write $sorts(\langle S, \Omega\rangle)$ to denote S, $opns(\langle S, \Omega\rangle)$ to denote Ω, and $f\colon w \to s$ to denote $w \in S^*$, $s \in S$, $f \in \Omega_{w,s}$. A *signature morphism* $\sigma\colon \langle S, \Omega\rangle \to \langle S', \Omega'\rangle$ is a pair $\langle \sigma_{sorts}, \sigma_{opns}\rangle$ where $\sigma_{sorts}\colon S \to S'$ and σ_{opns} is a family of maps $\{\sigma_{w,s}\colon \Omega_{w,s} \to \Omega'_{\sigma^*(w),\sigma(s)}\}_{w\in S^*, s\in S}$

where $\sigma^*(s_1,\ldots,s_n)$ denotes $\sigma_{sorts}(s_1),\ldots,\sigma_{sorts}(s_n)$ for $s_1,\ldots,s_n \in S$. We will write $\sigma(s)$ for $\sigma_{sorts}(s)$, $\sigma(w)$ for $\sigma^*(w)$ and $\sigma(f)$ for $\sigma_{w,s}(f)$, where $f \in \Omega_{w,s}$.

The category of signatures **Sign** has signatures as objects and signature morphisms as morphisms; the composition of morphisms is the composition of their corresponding components as functions. (This obviously forms a category.) **Sign** is cocomplete (see [GB 84b]); the initial signature is the empty signature $\Sigma_\emptyset$ with no sorts and hence no operations. $|\mathbf{Sign}|$ denotes the collection of objects of **Sign**, and we use **Sign** to denote the collection of its morphisms. We use the same notation for other categories.

Notation The composition of morphisms in any category (in particular, of functions) is denoted by semicolon and written in the diagrammatic order, e.g. $f\colon A \to B$ and $g\colon B \to C$ implies $f;g\colon A \to C$.

Let $\Sigma = \langle S, \Omega\rangle$ be a signature.

A *(partial)* Σ*-algebra* A consists of an S-indexed family of *carrier sets* $|A| = \{|A|_s\}_{s\in S}$ and for each $f\colon s_1,\ldots,s_n \to s$ a partial function $f_A\colon |A|_{s_1} \times \cdots \times |A|_{s_n} \to |A|_s$. A is called a *total algebra* if all of these functions are total. A Σ*-homomorphism* from a Σ-algebra A to a Σ-algebra B, $h\colon A \to B$, is a family of (total) functions $\{h_s\}_{s\in S}$ where $h_s\colon |A|_s \to |B|_s$ such that for any $f\colon s_1,\ldots,s_n \to s$ and $a_1 \in |A|_{s_1},\ldots,a_n \in |A|_{s_n}$

$$\begin{array}{lll} f_A(a_1,\ldots,a_n) \text{ defined} & \Rightarrow & f_B(h_{s_1}(a_1),\ldots,h_{s_n}(a_n)) \text{ defined and} \\ & & h_s(f_A(a_1,\ldots,a_n)) = f_B(h_{s_1}(a_1),\ldots,h_{s_n}(a_n)) \end{array}$$

([BrW 82] would call this a *total* Σ-homomorphism).

The category of partial Σ-algebras $\mathbf{PAlg}(\Sigma)$ has Σ-algebras as objects and Σ-homomorphisms as morphisms; the composition of homomorphisms is the composition of their corresponding components as functions. (This obviously forms a category.) In the sequel we identify classes of Σ-algebras with full subcategories of $\mathbf{PAlg}(\Sigma)$ and vice versa.

For any signature morphism $\sigma\colon \Sigma \to \Sigma'$ and Σ'-algebra A', the σ*-reduct* of A' is the Σ-algebra $A'|_\sigma$ defined as follows:

- For $s \in S$, $|A'|_\sigma|_s =_{def} |A'|_{\sigma(s)}$.
- For $f\colon w \to s$ in Σ, $f_{A'|_\sigma} =_{def} \sigma(f)_{A'}$.

Similarly, for a Σ'-homomorphism $h'\colon A' \to B'$ where A' and B' are Σ'-algebras, the σ-reduct of h' is the Σ-homomorphism $h'|_\sigma\colon A'|_\sigma \to B'|_\sigma$ defined by $(h'|_\sigma)_s =_{def} h'_{\sigma(s)}$ for $s \in S$.

The mappings $A' \mapsto A'|_\sigma$ and $h' \mapsto h'|_\sigma$ form a functor $\_|_\sigma\colon \mathbf{PAlg}(\Sigma') \to \mathbf{PAlg}(\Sigma)$.

Notice that in the above we have defined a functor $\mathbf{PAlg}\colon \mathbf{Sign} \to \mathbf{Cat}^{op}$ (where **Cat** is the category of all categories; $\mathbf{PAlg}(\sigma) = \_|_\sigma$). For the empty signature $\Sigma_\emptyset$ there is exactly one $\Sigma_\emptyset$-algebra, namely the one with no carriers, which has exactly one (empty) homomorphism on it. Thus, **PAlg** maps the initial signature to the terminal category. This is a consequence of a more general property: **PAlg** is cocontinuous (the proof of this fact for total algebras was given in [BW 85]; the proof for partial algebras is essentially the same). In particular, **PAlg** translates pushouts in **Sign** to pullbacks in **Cat**, which by the construction of pullbacks in **Cat** implies the following lemma:

Lemma 2.1 (amalgamation lemma) *Let*

$$\begin{tikzcd} \Sigma \arrow[r, "\sigma 1"] \arrow[d, "\sigma 2"'] & \Sigma 1 \arrow[d, "\sigma 1'"] \\ \Sigma 2 \arrow[r, "\sigma 2'"'] & \Sigma' \end{tikzcd}$$

be a pushout in **Sign**. *Then for any $\Sigma 1$-algebra $A1$ and $\Sigma 2$-algebra $A2$ such that $A1|_{\sigma 1} = A2|_{\sigma 2}$, there exists a unique Σ'-algebra A' such that $A'|_{\sigma 1'} = A1$ and $A'|_{\sigma 2'} = A2$.* □

A similar fact also holds for homomorphisms.

For any signature Σ, the algebra T_Σ of *ground Σ-terms* is defined in the usual way, as the (to within isomorphism) initial total Σ-algebra, i.e. the carriers $|T_\Sigma|$ contain terms of the appropriate sorts constructed using the operations of Σ without variables and the operations in T_Σ are defined in the natural way (see for example [GTW 76]). Moreover, for any set X of variables, the Σ-algebra $T_\Sigma(X)$ of *Σ-terms with variables X* is defined as $T_{\Sigma(X)}|_\iota$ where $\Sigma(X)$ is the extension of Σ by the elements of X as new constants of the appropriate sorts and $\iota\colon \Sigma \to \Sigma(X)$ is the signature inclusion.

A *partial first-order Σ-sentence* is a closed first-order formula built from Σ-terms using the logical connectives $\neg$, $\wedge$, $\vee$ and $\Rightarrow$, the quantifiers $\forall$ and $\exists$, and the atomic formulae $D_s(t)$ and $t = t'$ (strong equality [BrW 82]) for each sort s in Σ and terms $t, t' \in |T_\Sigma(X)|_s$ (i.e. t, t' are Σ-terms of sort s with variables X).

A partial Σ-algebra A *satisfies* an atomic formula $D_s(t)$ under a (total) valuation $v\colon X \to |A|$, written $A \models_v D_s(t)$, iff the value $t_A(v)$ of t in A under v is defined (we omit the definition of the value of a term in a partial algebra under a valuation; see [Bur 86] for details). If X is empty (i.e. if t is a ground Σ-term) we write t_A to denote the value of t in A. A satisfies an atomic formula $t = t'$ (where $t, t' \in |T_\Sigma(X)|_s$ for some sort s in Σ) under a valuation $v\colon X \to |A|$, written $A \models_v t = t'$, iff

- $A \not\models_v D_s(t)$ and $A \not\models_v D_s(t')$, or
- $A \models_v D_s(t)$ and $A \models_v D_s(t')$ and the values of t and t' in A under v are the same.

Satisfaction of (closed) partial first-order Σ-sentences is defined as usual, but note that $\forall$ and $\exists$ quantify only over defined values. We generalise the satisfaction relation to classes of algebras and sets of sentences in the usual way: $C \models \Phi$ means for all $A \in C$ and $\varphi \in \Phi$, $A \models \varphi$. We will omit the subscript on D when there is no danger of confusion.

Let $\sigma\colon \Sigma \to \Sigma'$ be a signature morphism. The unique Σ-homomorphism $h\colon T_\Sigma \to T_{\Sigma'}|_\sigma$ determines a translation of Σ-terms to Σ'-terms. For a ground Σ-term t of sort s we write $\sigma(t)$ rather than $h_s(t)$. This in turn determines a translation (again denoted by σ) of Σ-sentences to Σ'-sentences: e.g. $\sigma(t = t') =_{def} \sigma(t) = \sigma(t')$ and $\sigma(D_s(t)) =_{def} D_{\sigma(s)}(\sigma(t))$, etc. This notation extends to sets of sentences and sets of terms in the obvious way: $\sigma(W) = \{\sigma(t) \mid t \in W\}$ for any set W of ground Σ-terms and similarly for sentences.

The translations of sentences and of algebras as defined above preserve the satisfaction relation in the following sense:

Lemma 2.2 (satisfaction lemma) *For any signature morphism $\sigma\colon \Sigma \to \Sigma'$, Σ-sentence φ and Σ'-algebra A', $A'|_\sigma \models \varphi$ iff $A' \models \sigma(\varphi)$.*

Proof This follows from the fact that definedness of terms is preserved under change of signature, and by the proof of the analogous lemma for total algebras in [GB 84a]. □

For any signature Σ and Σ-algebra A, a (*closed*) Σ-*subalgebra* of A is a Σ-algebra B such that for any sort s in Σ, $|B|_s \subseteq |A|_s$ and for any operation $f\colon s_1,\ldots,s_n \to s$ in Σ, for $b_1 \in |B|_{s_1},\ldots,b_n \in |B|_{s_n}$, $f_B(b_1,\ldots,b_n)$ is defined iff $f_A(b_1,\ldots,b_n)$ is defined and then $f_B(b_1,\ldots,b_n) = f_A(b_1,\ldots,b_n)$. Thus we can identify Σ-subalgebras of A with subsets of the carrier of A closed under operations as defined in A. Notice that the set of Σ-subalgebras of A is closed under (set-theoretic) intersection.

For any signature Σ and $S \subseteq sorts(\Sigma)$, we say that a Σ-algebra A is *reachable on S* if it contains no proper Σ-subalgebra with carriers of sorts not in S the same as in A. In other words, every element of A is reachable from elements of sorts not in S using the operations of Σ (is the value of a Σ-term with variables of sorts not in S, for some valuation). Notice that any Σ-algebra A contains exactly one Σ-subalgebra which is reachable on S and has carriers of sorts not in S the same as in A, denoted $\mathcal{R}_S(A)$. We omit qualification by S in these definitions if $S = sorts(\Sigma)$.

Let $A \in \mathbf{PAlg}(\Sigma)$. A *congruence on A* is an equivalence relation $\equiv\ \subseteq |A| \times |A|$ such that for any $f\colon s_1,\ldots,s_n \to s$ in Σ and $a_1,b_1 \in |A|_{s_1},\ldots,a_n,b_n \in |A|_{s_n}$, if $a_1 \equiv_{s_1} b_1,\ldots,a_n \equiv_{s_n} b_n$ and $f_A(a_1,\ldots,a_n)$ and $f_A(b_1,\ldots,b_n)$ are defined, then $f_A(a_1,\ldots,a_n) \equiv_s f_A(b_1,\ldots,b_n)$.

Fact 2.3 *If $\equiv$ is a congruence on A, then $A/{\equiv}$ is a well-defined Σ-algebra, where $|A/{\equiv}|_s = |A|_s/{\equiv_s}$ and for every $f\colon s_1,\ldots,s_n \to s$ in Σ and $a_1 \in |A|_{s_1},\ldots,a_n \in |A|_{s_n}$, $f_{A/\equiv}([a_1],\ldots,[a_n])$ is defined iff $f_A(b_1,\ldots,b_n)$ is defined for some $b_1 \equiv_{s_1} a_1,\ldots,b_n \equiv_{s_n} a_n$ and then $f_{A/\equiv}([a_1],\ldots,[a_n]) = [f_A(b_1,\ldots,b_n)]$.*
□

For any set E of Σ-equations and $A \in \mathbf{PAlg}(\Sigma)$, let $\sim_A^E$ be the least congruence on A such that $t_A(v) \sim_A^E t'_A(v)$ for all equations $\forall X.t = t'$ in E and valuations $v\colon X \to |A|$ such that $A \models_v D(t)$ and $A \models_v D(t')$.

We write A/E to denote the quotient algebra $A/{\sim_A^E}$.

3 Specifications and refinement

We are not going to formally define precisely what specifications are; they are just finite syntactic objects of some kind, where the exact syntax used does not matter here (although it may be extremely important from the pragmatic point of view). What does matter is that every specification describes a certain signature and a class of algebras over this signature (intuitively, the class of algebras which satisfy the specification, or perhaps more exactly, which are acceptable realisations of the specification). This semantics is made explicit using two mappings which assign to each specification SP a signature $Sig[SP] \in |\mathbf{Sign}|$ and a class $Mod[SP] \subseteq |\mathbf{PAlg}(Sig[SP])|$ of $Sig[SP]$-algebras. Algebras in $Mod[SP]$ are called *models of* SP. We call a specification *consistent* if it has at least one model.

This rather general description covers high-level user-oriented loose specifications admitting many non-isomorphic models as well as low-level detailed specifications which describe classes of isomorphic algebras or even programs which for us are just very tight specifications describing one particular algebra. Note that we adopt a purely model-theoretic view here and we stop the analysis of the notion of a program at this level. Therefore, we do not distinguish between efficient and inefficient algorithms to compute the same functions or even between effective and non-effective definitions. Any application of the methodology we outline here would require some further syntactic constraints on the notion of a program.

Definition 3.1 *For any signature* Σ, $\mathbf{Spec}(\Sigma)$ *denotes the collection of all* Σ-specifications, *i.e. specifications* SP *such that* $Sig[SP] = \Sigma$, *preordered by the inclusion of model classes. (This preorder turns* $\mathbf{Spec}(\Sigma)$ *into a category.) For any two specifications* $SP1$ *and* $SP2$, *a* specification morphism from $SP1$ to $SP2$ *is a signature morphism* $\sigma\colon Sig[SP1] \to Sig[SP2]$ *such that for any model* $A2 \in Mod[SP2]$, $A2|_\sigma \in Mod[SP1]$. *We denote this by* $\sigma\colon SP1 \to SP2$.

$\mathbf{Spec}(\Sigma)$ is never empty. We assume that it contains at least *basic specifications*. That is, given a signature Σ and a (finite, recursive, recursively enumerable) set Φ of Σ-axioms (e.g. partial first-order Σ-sentences), $\langle \Sigma, \Phi \rangle$ is a specification with:

$$Sig[\langle \Sigma, \Phi \rangle] = \Sigma$$
$$Mod[\langle \Sigma, \Phi \rangle] = \{A \in \mathbf{PAlg}(\Sigma) \mid A \models \Phi\}$$

If the axioms are all (universally quantified) equations or definedness formulae we call $\langle \Sigma, \Phi \rangle$ an *equational specification.*

In any non-trivial application, specifications will tend to grow large and unmanageable. To make them useful, we have to build them in a structured manner and then exploit this structure as a guide in their use and understanding. This is accomplished by use of *specification-building operations* to put together little specifications in nice ways to make progressively bigger ones [BG 77]. Any specification-building operation, given a list of argument specifications, yields a result specification. Again, it does not matter for us how this is written; semantically, a specification-building operation is a function on classes of algebras. It maps classes of models (of the argument specifications) to the class of models (over a signature which must be determined as well) of the result specification. The only assumption we make about these functions is that they are *monotonic* with respect to inclusion of classes of algebras; intuitively, less restrictive argument specifications yield a less restrictive result. Specification languages like CLEAR [BG 77,80], LOOK [ETLZ 82], ACT ONE [EFH 83], [EM 85], ASL [SW 83], [Wir 86], [ST 86a] and others may be viewed just as sets of such operations plus some syntactic sugar.

To make this more concrete, let us recall two simple examples of specification-building operations (taken from [ST 86a]):

Union: Given two specifications $SP1$ and $SP2$ over the same signature Σ (i.e. $Sig[SP1] = \Sigma = Sig[SP2]$), $SP1 \cup SP2$ is a specification with semantics defined as follows:

$$\begin{aligned} &Sig[SP1 \cup SP2] = \Sigma \\ &Mod[SP1 \cup SP2] = Mod[SP1] \cap Mod[SP2] \end{aligned}$$

Translate: Given a specification SP and a signature morphism $\sigma\colon Sig[SP] \to \Sigma'$, the semantics of the specification **translate** SP **by** σ is as follows:

$$\begin{aligned} &Sig[\textbf{translate } SP \textbf{ by } \sigma] = \Sigma' \\ &Mod[\textbf{translate } SP \textbf{ by } \sigma] = \{A' \in \mathbf{PAlg}(\Sigma') \mid A'|_\sigma \in Mod[SP]\} \end{aligned}$$

Further examples will be given in the sequel.

Strictly speaking, both union and **translate** are really *families* of specification-building operations:

$$\cup = \{\cup_\Sigma\colon \mathbf{Spec}(\Sigma) \times \mathbf{Spec}(\Sigma) \to \mathbf{Spec}(\Sigma)\}_{\Sigma \in |\mathbf{Sign}|}$$

and

$$\textbf{translate} = \{\textbf{translate}_{\sigma\colon \Sigma \to \Sigma'}\colon \mathbf{Spec}(\Sigma) \to \mathbf{Spec}(\Sigma')\}_{\sigma \in \mathbf{Sign}}$$

The elements of these families are defined in a uniform way which allows us to leave out the subscripts when convenient and justifies us calling them specification-building operations as in the above informal remarks. For any specification-building operation ω we will write $\omega\colon \mathbf{Spec}(\Sigma) \to \mathbf{Spec}(\Sigma')$ (meaning that ω takes Σ-specifications to Σ'-specifications) when we want to retain some of the information lost due to this informality. Since specification-building operations are required to be monotonic, $\omega\colon \mathbf{Spec}(\Sigma) \to \mathbf{Spec}(\Sigma')$ is a functor as the notation suggests. Note that we have tacitly assumed that ω is a unary operation; to simplify the presentation we are going to make the same assumption throughout when convenient.

Besides providing a certain collection of specification-building operations, a specification language usually provides a way for the user to define his own specification-building operations, i.e. a mechanism for constructing *parameterised specifications*. There are different approaches to parameterised specifications (e.g. [Ehr 82], [BG 80], [EKTWW 80], [Gan 83], [SW 83]); in this paper we use the approach of [ST 86a].

Semantically, any parameterised specification can be viewed as a function taking any specification over a given parameter signature Σ_{par} to a specification over a result signature Σ_{res}. Syntactically, we write a parameterised specification as a λ-expression, $\lambda X\colon \Sigma_{par}.SP_{res}[X]$, where X is an identifier and $SP_{res}[X]$ is a Σ_{res}-specification built using specification-building operations which may involve X as a variable denoting a Σ_{par}-specification. For any Σ_{par}-specification SP, $(\lambda X\colon \Sigma_{par}.SP_{res}[X])(SP)$ is a specification with semantics defined (essentially as β-conversion) as follows:

$$\begin{aligned} &Sig[(\lambda X\colon \Sigma_{par}.SP_{res}[X])(SP)] = \Sigma_{res} \\ &Mod[(\lambda X\colon \Sigma_{par}.SP_{res}[X])(SP)] = Mod[SP_{res}[SP/X]] \end{aligned}$$

(we adopt the usual λ-calculus convention that $E[v/x]$ denotes the result of substituting v for x in E). This easily extends to multiple arguments — see [ST 86a]. Consistently with our notation for specification-building operations, we sometimes write $(\lambda X\colon \Sigma_{par}.SP_{res}[X])\colon \mathbf{Spec}(\Sigma_{par}) \to \mathbf{Spec}(\Sigma_{res})$ to indicate the parameter and result signatures explicitly.

The programming discipline of stepwise refinement suggests that a program (which is a specification) be evolved from a high-level specification by working gradually via a series of successively more detailed lower-level intermediate specifications. A formalisation of this approach requires a precise definition of the concept of refinement.

In programming practice, proceeding from a specification to a program (by stepwise refinement or by any other method) means making a series of design decisions. These will include decisions concerning the concrete representation of abstractly defined data types, decisions about how to compute abstractly specified functions (choice of algorithm) and decisions which select between the various possibilities which the specification leaves open. The following very simple formal notion of refinement [SW 83], [ST 85b,87a] captures this idea.

Definition 3.2 *Given two specifications SP and SP' such that $Sig[SP] = Sig[SP']$, we say that SP* refines to *SP', written $SP \rightsquigarrow SP'$, if $Mod[SP'] \subseteq Mod[SP]$.*

Intuitively, $SP \rightsquigarrow SP'$ if SP' incorporates more design decisions than SP. This simply requires that any realisation of SP' is an acceptable realisation of SP.

This notion of refinement can be extended to parameterised specifications:

Definition 3.3 *Given two parameterised specifications P and P' with the same parameter signature Σ_{par}, we say that P* refines to *P', written $P \rightsquigarrow P'$, if for any Σ_{par}-specification SP, $P(SP) \rightsquigarrow P'(SP)$.*

An important issue for any notion of refinement is whether refinements can be composed *vertically* and *horizontally* [GB 80]. Refinements can be vertically composed if the refinement relation is transitive ($SP \rightsquigarrow SP'$ and $SP' \rightsquigarrow SP''$ implies $SP \rightsquigarrow SP''$) and they can be horizontally composed if the specification-building operations preserve refinements (i.e. $P \rightsquigarrow P'$ and $SP \rightsquigarrow SP'$ implies $P(SP) \rightsquigarrow P'(SP')$). The above notion of refinement has both these properties since specification-building operations are monotonic. There is also an obvious operation of composition of parameterised specifications, and it is easy to see that this preserves refinements as well. These properties allow large structured specifications to be refined in a gradual and modular fashion.

The development of a program from a specification consists of a series of refinement steps $SP_0 \rightsquigarrow SP_1 \rightsquigarrow \cdots \rightsquigarrow SP_n$, where SP_0 is the original high-level specification and SP_n is a program. Vertical composability guarantees the correctness of SP_n with respect to its specification SP_0. This views each of the specifications $SP_0, \ldots, SP_n$ as a single indivisible entity. If, however, we decompose any of them using a parameterised specification, say $SP_k = P(SP)$, then the further developments of P and of SP may proceed separately. Horizontal composability guarantees that the results of these developments may always be combined to give a refinement of SP_k and so of SP_0 as well. Of course, these (sub)developments may themselves involve further decomposition.

4 Constructors and implementations

In the last section we presented a simple notion of refinement which is mathematically elegant but perhaps a bit oversimplified from the practical point of view. In this section and those which follow, we will develop notions of implementation built on top of this simple notion of refinement which are more suited to practical use. We start with a notion of implementation which involves a construction from the implementing specification to the implemented specification.

What is a construction? According to our model-theoretic view, the characteristic feature of a construction is that it takes an algebra over one signature and transforms it to yield another algebra over a (possibly different) signature. Thus, we can identify a construction κ with a function[4] κ mapping Σ-algebras to Σ'-algebras, $\kappa\colon \mathbf{PAlg}(\Sigma) \to \mathbf{PAlg}(\Sigma')$. In an obvious way, this determines a specification-building operation denoted (ambiguously) by the same symbol. We call specification-building operations of this kind *constructors*.

Definition 4.1 *A* constructor *determined by a function* $\kappa\colon \mathbf{PAlg}(\Sigma) \to \mathbf{PAlg}(\Sigma')$ *is a specification-building operation* $\kappa\colon \mathbf{Spec}(\Sigma) \to \mathbf{Spec}(\Sigma')$*, where for any* Σ*-specification SP,* $Sig[\kappa(SP)] = \Sigma'$ *and* $Mod[\kappa(SP)] = \{\kappa(A) \mid A \in Mod[SP]\}$*.*

We find it convenient to view constructors as functions transforming algebras and at the same time as specification-building operations. It should be stressed that the latter view is, in a sense, superfluous. All the following concepts may be introduced and used directly, without assuming that constructors are available as specification-building operations, referring to them only as functions transforming algebras. We do not do this here, though, mainly to be able to highlight the direct relationship between the simple notion of refinement presented in the previous section and the somewhat more complex notions of implementation introduced below. We refrain as much as possible from developing any convenient notation for syntactic presentation of constructors. This is a very important but nevertheless separate task, which must eventually lead to the development of an appropriate programming language with powerful modularisation facilities, which is clearly outside the scope of this paper. Similar remarks also apply to the concept of abstractor we introduce in section 6.

A few easy facts follow immediately from the definition.

Fact 4.2 *Constructors are monotonic.* □

Fact 4.3 *Constructors preserve consistency of specifications.* □

Fact 4.4 *Constructors are closed under composition: if both* $\kappa 1\colon \mathbf{Spec}(\Sigma 1) \to \mathbf{Spec}(\Sigma 2)$ *and* $\kappa 2\colon \mathbf{Spec}(\Sigma 2) \to \mathbf{Spec}(\Sigma 3)$ *are constructors determined by functions* $\kappa 1\colon \mathbf{PAlg}(\Sigma 1) \to \mathbf{PAlg}(\Sigma 2)$ *and* $\kappa 2\colon \mathbf{PAlg}(\Sigma 2) \to \mathbf{PAlg}(\Sigma 3)$*, then the constructor* $\kappa 1;\kappa 2\colon \mathbf{Spec}(\Sigma 1) \to \mathbf{Spec}(\Sigma 3)$ *is determined by the function* $\kappa 1;\kappa 2\colon \mathbf{PAlg}(\Sigma 1) \to \mathbf{PAlg}(\Sigma 3)$. □

[4] From the category-theoretic point of view, it is natural to assume that this is a functor (all our examples are) but since we do not use the morphism part in this paper we take this simplified view here.

Example 4.5 (derive) For any Σ'-specification SP' and signature morphism $\sigma: \Sigma \to \Sigma'$, the semantics of the specification **derive from** SP' **by** σ is as follows:

$$\begin{aligned} &Sig[\textbf{derive from } SP' \textbf{ by } \sigma] = \Sigma \\ &Mod[\textbf{derive from } SP' \textbf{ by } \sigma] = \{A|_\sigma \mid A \in Mod[SP']\} \end{aligned}$$

The **derive** specification-building operations (one for each $\sigma: \Sigma \to \Sigma'$) are constructors determined by the corresponding reduct functors $\_|_\sigma$ (cf. section 2). Intuitively, **derive** can be used to hide and/or rename some of the sorts and operations of a specification. □

Example 4.6 (restrict) For any Σ-specification SP and set $S \subseteq sorts[\Sigma]$ of sorts, the semantics of the specification **restrict** SP **on** S is as follows:

$$\begin{aligned} &Sig[\textbf{restrict } SP \textbf{ on } S] = \Sigma \\ &Mod[\textbf{restrict } SP \textbf{ on } S] = \{\mathcal{R}_S(A) \mid A \in Mod[SP]\} \end{aligned}$$

The **restrict** specification-building operations (one for each Σ and $S \subseteq sorts[\Sigma]$) are constructors determined by the corresponding restrict functors $\mathcal{R}_S$ (cf. section 2). **Restrict** is used to remove "junk" (unreachable elements) of selected sorts. □

Example 4.7 (quotient) For any Σ-specification SP and set E of Σ-equations, the semantics of the specification **quotient** SP **wrt** E is as follows:

$$\begin{aligned} &Sig[\textbf{quotient } SP \textbf{ wrt } E] = \Sigma \\ &Mod[\textbf{quotient } SP \textbf{ wrt } E] = \{A/E \mid A \in Mod[SP]\} \end{aligned}$$

The **quotient** specification-building operations (one for each Σ and E) are constructors determined by the corresponding quotient functors $\_/E$ (cf. section 2). Intuitively, **quotient** is used to identify the values of certain terms. □

Example 4.8 (extend) If we have a signature morphism $\sigma: \Sigma \to \Sigma'$ then constructors from $\mathbf{Spec}(\Sigma)$ to $\mathbf{Spec}(\Sigma')$ will be called *synthesizing constructors along* σ. The intuition is that they just build new stuff on top of the existing algebras without forgetting anything. One standard way to define such a synthesizing constructor is using the free extension.

Namely, for any signature morphism $\sigma: \Sigma \to \Sigma'$ and equational Σ'-specification SP', there is a free functor $\mathbf{F}_\sigma: \mathbf{PAlg}(\Sigma) \to Mod[SP']$ (the left adjoint to the reduct functor $\_|_\sigma: Mod[SP'] \to \mathbf{PAlg}(\Sigma)$). That this functor always exists is a well-known fact — see [GTW 76] (total algebras) and [BrW 82] (partial algebras). Now, for any Σ-specification SP, **extend** SP **to** SP' **via** σ is a specification defined as follows:

$$\begin{aligned} &Sig[\textbf{extend } SP \textbf{ to } SP' \textbf{ via } \sigma] = \Sigma' \\ &Mod[\textbf{extend } SP \textbf{ to } SP' \textbf{ via } \sigma] = \{\mathbf{F}_\sigma(A) \mid A \in Mod[SP]\} \end{aligned}$$

The **extend** specification-building operations (one for each σ and SP') are constructors determined by the corresponding free functor $\mathbf{F}_\sigma$.

In examples, we will use the standard notation **enrich** SP **by data sorts** S **opns** Ω **axioms** Φ (from CLEAR) to abbreviate **extend** SP **to** $\langle \Sigma', \Phi \rangle$ **via** ι where $\Sigma' =_{def} \Sigma \cup \langle S, \Omega \rangle$ and $\iota\colon \Sigma \hookrightarrow \Sigma'$ is the inclusion, provided Σ' is a signature. Recall (see [BG 80]) that the keyword **data** is significant here. We will use the notation **enrich** SP **by sorts** S **opns** Ω **axioms** Φ with a different meaning later on (see section 5).

Note that in the above, SP may be an arbitrary specification, not necessarily equational. In general $\mathbf{F}_\sigma$ does not have to preserve all the properties required by SP (σ was not required to be a specification morphism $\sigma\colon SP \to SP'$) although it does preserve ground equations deducible from SP. Notice also that this yields the initial algebra construction as a special case (where $\Sigma = \Sigma_\emptyset$). $\square$

Non-example (translate) The **translate** specification-building operation defined in the last section is *not* a constructor. Consider for example the signature morphism $\sigma\colon \Sigma_\emptyset \to \Sigma$ which is the inclusion of the empty signature into some non-empty signature Σ. The (empty) $\Sigma_\emptyset$-specification $\emptyset$ has exactly one model while **translate** $\emptyset$ **by** σ has all of $\mathbf{PAlg}(\Sigma)$ as models, so $\mathbf{translate}_{\sigma\colon \Sigma_\emptyset \to \Sigma}$ is not determined by a function on algebras. Furthermore, if $\sigma'\colon \Sigma \to \Sigma'$ is a signature morphism which is non-injective on sorts (i.e. for some $s, s' \in sorts[\Sigma]$, $\sigma'(s) = \sigma'(s')$ while $s \neq s'$), then those models A of the specification $\langle \Sigma, \emptyset \rangle$ for which $|A|_s \neq |A|_{s'}$ will have no corresponding models in **translate** $\langle \Sigma, \emptyset \rangle$ **by** σ' so $\mathbf{translate}_{\sigma'\colon \Sigma \to \Sigma'}$ is not determined by a function on algebras either. Thus, for $\sigma\colon \Sigma \to \Sigma'$ and a Σ-specification SP, there may be models of SP which give rise to more than one model of **translate** SP **by** σ and other models of SP which give rise to no model of **translate** SP **by** σ. $\square$

Definition 4.9 *A synthesizing constructor* $\kappa\colon \mathbf{Spec}(\Sigma) \to \mathbf{Spec}(\Sigma')$ *is* persistent along *a signature morphism* $\sigma\colon \Sigma \to \Sigma'$, *written* $\kappa\colon \mathbf{Spec}(\Sigma) \xrightarrow{\sigma} \mathbf{Spec}(\Sigma')$, *if* $\kappa\colon \mathbf{PAlg}(\Sigma) \to \mathbf{PAlg}(\Sigma')$ *is (strongly) persistent with respect to* σ, *i.e. for any* Σ*-algebra* A, $\kappa(A)\big|_\sigma = A$.

Example 4.10 (amalgamated union) Given two persistent constructors $\kappa 1\colon \mathbf{Spec}(\Sigma) \xrightarrow{\sigma 1} \mathbf{Spec}(\Sigma 1)$ and $\kappa 2\colon \mathbf{Spec}(\Sigma) \xrightarrow{\sigma 2} \mathbf{Spec}(\Sigma 2)$, let

$$\begin{tikzcd} \Sigma \arrow[r, "\sigma 1"] \arrow[d, "\sigma 2"'] & \Sigma 1 \arrow[d, "\sigma 1'"] \\ \Sigma 2 \arrow[r, "\sigma 2'"'] & \Sigma' \end{tikzcd}$$

be a pushout in **Sign**. For any Σ-algebra A, define $\kappa(A)$ to be the unique Σ'-algebra such that $\kappa(A)\big|_{\sigma 1'} = \kappa 1(A)$ and $\kappa(A)\big|_{\sigma 2'} = \kappa 2(A)$. $\kappa(A)$ is well-defined by the amalgamation lemma since $\kappa 1(A)\big|_{\sigma 1} = A = \kappa 2(A)\big|_{\sigma 2}$. Thus, we have defined a function $\kappa\colon \mathbf{PAlg}(\Sigma) \to \mathbf{PAlg}(\Sigma')$. We denote this function and the corresponding synthesizing constructor (along $\sigma 1;\sigma 1' = \sigma 2;\sigma 2'$) by $\kappa 1 + \kappa 2$; if any doubts may arise, we add $\sigma 1, \sigma 2$ as subscripts to $+$. Intuitively, $\kappa 1 + \kappa 2$ "puts together" the constructions $\kappa 1$ and $\kappa 2$. The assumption of persistency guarantees that this is possible. (See the notion of amalgamated sum in [PB 85] and [EM 85].) $\square$

Fact 4.11 *If* $\kappa 1$: $\mathbf{Spec}(\Sigma) \xrightarrow{\sigma 1} \mathbf{Spec}(\Sigma 1)$ *and* $\kappa 2$: $\mathbf{Spec}(\Sigma) \xrightarrow{\sigma 2} \mathbf{Spec}(\Sigma 2)$ *are persistent constructors then* $\kappa 1 + \kappa 2$: $\mathbf{Spec}(\Sigma) \xrightarrow{\sigma} \mathbf{Spec}(\Sigma')$ *is a persistent constructor along* $\sigma =_{def} \sigma 1;\sigma 1' = \sigma 2;\sigma 2'$.

Proof For any Σ-algebra A, $(\kappa 1 + \kappa 2)(A)\big|_{\sigma 1;\sigma 1'} = ((\kappa 1 + \kappa 2)(A)\big|_{\sigma 1'})\big|_{\sigma 1} = \kappa 1(A)\big|_{\sigma 1} = A$. □

Example 4.12 (translation of a constructor) There is another operator on constructors connected with the pushout in **Sign**. Namely, let

$$\begin{tikzcd} \Sigma \arrow[r, "\sigma 1"] \arrow[d, "\sigma 2"'] & \Sigma 1 \arrow[d, "\sigma 1'"] \\ \Sigma 2 \arrow[r, "\sigma 2'"'] & \Sigma' \end{tikzcd}$$

be a pushout in **Sign**, and suppose $\kappa 1$: $\mathbf{Spec}(\Sigma) \xrightarrow{\sigma 1} \mathbf{Spec}(\Sigma 1)$ is a persistent constructor. Then for any $A2 \in \mathbf{PAlg}(\Sigma 2)$, define $\sigma 2(\kappa 1)(A2)$ to be the unique Σ'-algebra such that $\sigma 2(\kappa 1)(A2)\big|_{\sigma 1'} = \kappa 1(A2\big|_{\sigma 2})$ and $\sigma 2(\kappa 1)(A2)\big|_{\sigma 2'} = A2$. Thus we have defined a function $\sigma 2(\kappa 1)$: $\mathbf{PAlg}(\Sigma 2) \to \mathbf{PAlg}(\Sigma')$ which we call the *translation of* $\kappa 1$ *along* $\sigma 2$. We use the same notation and terminology to refer to the corresponding synthesizing constructor (along $\sigma 2'$). Notice that $\sigma 2(\kappa 1)$ is persistent. Intuitively, $\sigma 2(\kappa 1)$ performs $\kappa 1$ on the "Σ part" of $\Sigma 2$-algebras and leaves the other components unchanged.

Notice that the translation of a constructor is a more elementary operation than the amalgamated union. Namely, using the notation of example 4.10, $\kappa 1 + \kappa 2 = \kappa 2;\sigma 2(\kappa 1) = \kappa 1;\sigma 1(\kappa 2)$. □

As promised at the beginning of this section, we are going to use the notion of a constructor to give a more practically useful definition of implementation.

Definition 4.13 (constructor implementation) *A specification SP* is implemented by *a specification SP′* via *a constructor* κ: $\mathbf{Spec}(Sig[SP']) \to \mathbf{Spec}(Sig[SP])$, *written* $SP \underset{\kappa}{\leadsto} SP'$, *if* $SP \leadsto \kappa(SP')$. *In other words,* $SP \underset{\kappa}{\leadsto} SP'$ *if* κ *transforms every model of SP′ to a model of SP.*

Intuitively speaking, if we want to evaluate a function in *SP*, we are able to do this provided we can evaluate any function in *SP′* since the constructor κ puts together functions in *SP′* to obtain all functions in *SP*. In this sense, κ may be viewed as a program parameterised by the (possibly not yet executable) specification *SP′*. The development of an appropriate syntax for such programs κ is an important and interesting but separate task.

Notice that, using the constructors introduced in examples 4.5-4.8 above, we can reduce many of the notions of implementation in the literature (e.g. [GTW 76], [Ehr 82], [EKMP 82], [SW 82]) to the one above. For example, the implementation notion of [EKMP 82] assumes that κ is the composition of **extend**, **derive**, **restrict** and **quotient** constructors (in that order). Notice also that constructor implementation is a proper generalisation of the notion of refinement of the previous section; $\leadsto$ is just $\underset{id}{\leadsto}$ (constructor implementation via the identity constructor *id*).

Our definition of constructor implementation resembles the notion of implementation given in [Ehr 81] for single algebras. In [Ehr 81], A is implemented by B via a construction F if A is (isomorphic to) a quotient of a subalgebra of $F(B)$. When generalising to loose specifications, the requirement that *some* quotient of *some* subalgebra of $F(B)$ be isomorphic to A may be regarded as a construction only if the subalgebra and quotient are taken uniformly on all models B of the implementing specification. If we do not require uniformity then this amounts to a non-constructive step which will be fully subsumed by the notion of abstractor implementation defined in section 6. There are even closer similarities with the notion of implementation of (parameterised) specifications in [Lip 83]; see section 8.1 for details.

As indicated by fact 4.4, we are able to compose constructors, which easily yields the following important theorem:

Theorem 4.14 (vertical composition) *If* $SP \underset{\kappa}{\rightsquigarrow} SP'$ *and* $SP' \underset{\kappa'}{\rightsquigarrow} SP''$ *then* $SP \underset{\kappa';\kappa}{\rightsquigarrow} SP''$.

Proof By definition, $Mod[SP'] \supseteq Mod[\kappa'(SP'')]$, hence by definition $Mod[SP] \supseteq Mod[\kappa(SP')] = \kappa(Mod[SP']) \supseteq \kappa(Mod[\kappa'(SP'')]) = Mod[(\kappa';\kappa)(SP'')]$. $\square$

Notice that since $\kappa';\kappa$ is an acceptable constructor, there is no reason to require that it has (or may be transformed to) the same form as either κ or κ'. In general this will not be the case. However, in some special cases it turns out that such normal form theorems may be obtained, often under some additional assumptions about the specifications involved (see e.g. [Ehr 81], [EKMP 82], [SW 82], [EWT 83], [Ore 83]). It seems to us that the requirement that the composition of constructors must be forced into some given normal form corresponds to requiring programs to be written in a rather restrictive programming language which does not provide sufficiently powerful modularisation facilities for the job. In some situations, putting a constructor into a normal form can be viewed as an optimization process.

The following simple fact allows us to mechanically strip off outermost constructors if the specification we want to implement happens to be built in this way.

Fact 4.15 *For any constructor* κ: $\mathbf{Spec}(\Sigma) \to \mathbf{Spec}(\Sigma')$ *and* Σ*-specification* SP, $\kappa(SP) \underset{\kappa';\kappa}{\rightsquigarrow} SP'$ *provided that* $SP \underset{\kappa'}{\rightsquigarrow} SP'$.

Proof By definition, $\kappa(SP) \underset{\kappa}{\rightsquigarrow} SP$. Then the desired result follows from the vertical composition theorem. $\square$

An interesting special case of this is the amalgamated union of specifications.

Definition 4.16 *For any two specification morphisms* $\sigma 1$: $SP \to SP1$ *and* $\sigma 2$: $SP \to SP2$, *the* amalgamated union of $SP1$ and $SP2$, *written* $SP1 + SP2$ *(decorated with subscripts* $SP, \sigma 1, \sigma 2$ *on* $+$ *if necessary), is a specification with semantics defined as follows:*

$$Sig[SP1 + SP2] = \Sigma'$$
$$Mod[SP1 + SP2] = Mod[(\textbf{translate } SP1 \textbf{ by } \sigma 1') \cup (\textbf{translate } SP2 \textbf{ by } \sigma 2')]$$

where

$$\begin{tikzcd} Sig[SP] \arrow[r, "\sigma 1"] \arrow[d, "\sigma 2"'] & Sig[SP1] \arrow[d, "\sigma 1'"] \\ Sig[SP2] \arrow[r, "\sigma 2'"'] & \Sigma' \end{tikzcd}$$

is a pushout in **Sign**.

In particular, we can form the disjoint union of any two specifications $SP1$ and $SP2$ by letting $\sigma 1$ and $\sigma 2$ be the inclusions of the empty specification over the empty signature into $Sig[SP1]$ and $Sig[SP2]$ respectively.

Notice that according to this definition, $+$ is a derived specification-building operation which is defined in terms of **translate** and $\cup$.

Theorem 4.17 *If* $SP1 \underset{\kappa 1}{\rightsquigarrow} SP$ *and* $SP2 \underset{\kappa 2}{\rightsquigarrow} SP$ *where both* $\kappa 1$: $\mathbf{Spec}(Sig[SP]) \xrightarrow{\sigma 1} \mathbf{Spec}(Sig[SP1])$ *and* $\kappa 2$: $\mathbf{Spec}(Sig[SP]) \xrightarrow{\sigma 2} \mathbf{Spec}(Sig[SP2])$ *are persistent constructors, then* $SP1 + SP2 \underset{\kappa 1 + \kappa 2}{\rightsquigarrow} SP$.

Proof By definition, we have to show that for $A \in Mod[SP]$, $(\kappa 1 + \kappa 2)(A) \in Mod[SP1 + SP2]$, i.e. that $(\kappa 1 + \kappa 2)(A)|_{\sigma 1'} \in Mod[SP1]$ and $(\kappa 1 + \kappa 2)(A)|_{\sigma 2'} \in Mod[SP2]$, which is obvious since by the definition of $\kappa 1 + \kappa 2$, $(\kappa 1 + \kappa 2)(A)|_{\sigma 1'} = \kappa 1(A) \in Mod[SP1]$ and $(\kappa 1 + \kappa 2)(A)|_{\sigma 2'} = \kappa 2(A) \in Mod[SP2]$. $\square$

This theorem allows us to implement the independent components of a specification separately and then combine their implementations provided that they do not affect the common part.

In the above theorem we required $\kappa 1$ and $\kappa 2$ to be persistent on all $Sig[SP]$-algebras as in the definition of the amalgamated union of constructors. Notice that in this context, however, it is sufficient to require that $\kappa 1$ and $\kappa 2$ are persistent only on models of SP (which may be easier to achieve in practice). Of course formally, $\kappa 1 + \kappa 2$ is then only a constructor on $Mod[SP]$ rather than on $\mathbf{PAlg}(Sig[SP])$ since it may be undefined on some $Sig[SP]$-algebras.

Theorem 4.18 *Let*

$$\begin{tikzcd} \Sigma \arrow[r, "\sigma 1"] \arrow[d, "\sigma 2"'] & \Sigma 1 \arrow[d, "\sigma 1'"] \\ \Sigma 2 \arrow[r, "\sigma 2'"'] & \Sigma' \end{tikzcd}$$

be a pushout in **Sign**, $\kappa 1$: $\mathbf{Spec}(\Sigma) \xrightarrow{\sigma 1} \mathbf{Spec}(\Sigma 1)$ *be a persistent constructor, and* $SP1$, $SP2$ *be* $\Sigma 1$- *and* $\Sigma 2$-*specifications respectively. If* $SP1 \underset{\kappa 1}{\rightsquigarrow}$ **derive from** $SP2$ **by** $\sigma 2$, *then* $SP1 + SP2 \underset{\sigma 2(\kappa 1)}{\rightsquigarrow} SP2$.

Proof For $A2 \in Mod[SP2]$, by definition we have $\sigma 2(\kappa 1)(A2)|_{\sigma 2'} = A2 \in Mod[SP2]$ and $\sigma 2(\kappa 1)(A2)|_{\sigma 1'} = \kappa 1(A2|_{\sigma 2}) \in Mod[SP1]$. Thus $\sigma 2(\kappa 1)(A2) \in Mod[SP1 + SP2]$. □

This gives another way of decomposing a specification and implementing the components separately. Namely, we implement one component using (a part of) the other and then we can proceed with the implementation of the other component.

Notice that again, in this context the requirement of persistency of $\kappa 1$ may be relaxed to persistency on $\sigma 2$-reducts of models of $SP2$.

Summing up, the development process using this notion of implementation would consist of a sequence of implementation steps $SP_0 \underset{\kappa_1}{\rightsquigarrow} SP_1 \underset{\kappa_2}{\rightsquigarrow} \cdots \underset{\kappa_n}{\rightsquigarrow} SP_n$. Intuitively, SP_0, SP_1 etc. do not "grow" as happens when we use the simple refinement notion, where the same development would look like:

$$SP_0 \rightsquigarrow \kappa_1(SP_1) \rightsquigarrow \cdots \rightsquigarrow \kappa_1(\ldots \kappa_n(SP_n)\ldots)$$

Using constructor implementations, we gradually reduce the specification by implementing its parts. Our goal is to end up with an empty specification over the empty signature, i.e. $SP_n = \langle \Sigma_\emptyset, \emptyset \rangle$. Then according to theorem 4.14, the composition of constructors $\kappa_n; \cdots; \kappa_1$ forms a program which implements SP_0. Of course, usually it is sufficient to stop earlier, when we reach a specification containing only definitions of types and functions available in the programming language we intend to use.

This view of the program development process does not give a recipe for construction of the individual implementation steps. This is where human invention is required, although research on program and specification transformation (e.g. [Bau 81b] and [DLS 87]) offers techniques for systematising some of these steps, and work on program synthesis (e.g. [MW 80]) even suggests that some steps may be mechanically constructed. Theorems 4.17 and 4.18 above as well as theorem 8.5 of section 8.1 suggest ways of developing implementation steps in a structured manner by combining more primitive implementation steps.

5 Examples of constructor implementations

In the following examples and those of the sequel we will use the standard specifications of the boolean values *Bool* and the natural numbers *Nat* (which contains *Bool* because of the presence of operations like $\geq: nat, nat \rightarrow bool$). We also use the **enrich** notation of CLEAR, **enrich** SP **by sorts** S **opns** Ω **axioms** Φ, as an abbreviation for (**translate** SP **by** ι) $\cup \langle Sig[SP] \cup \langle S, \Omega \rangle, \Phi \rangle$, where ι is the obvious signature inclusion. All axioms are implicitly universally quantified over all free variables.

We begin with a simple specification of (finite) sets of natural numbers:

$SetNat =_{def}$ **restrict** (**enrich** Nat **by**
  sorts set
  opns $\emptyset: \rightarrow set$
  $add: nat, set \rightarrow set$

$isempty$: $set \to bool$
$\in$: $nat, set \to bool$
axioms $D(\emptyset)$
$D(add(a, S))$
$add(a, add(b, S)) = add(b, add(a, S))$
$add(a, add(a, S)) = add(a, S)$
$isempty(\emptyset) = true$
$isempty(add(a, S)) = false$
$a \in \emptyset = false$
$a \in add(a, S) = true$
$a \neq b \Rightarrow a \in add(b, S) = a \in S$)
on $\{set\}$

We will show below how to implement $SetNat$ by the following specification of bags (multisets) of natural numbers:

$BagNat =_{def}$ **restrict** (**enrich** Nat **by**
sorts bag
opns $\emptyset$: $\to bag$
add: $nat, bag \to bag$
$isempty$: $bag \to bool$
$count$: $nat, bag \to nat$
axioms $D(\emptyset)$
$D(add(a, B))$
$add(a, add(b, B)) = add(b, add(a, B))$
$isempty(\emptyset) = true$
$isempty(add(a, B)) = false$
$count(a, \emptyset) = 0$
$count(a, add(a, B)) = succ(count(a, B))$
$a \neq b \Rightarrow count(a, add(b, B)) = count(a, B)$)
on $\{bag\}$

The constructor implementation of $SetNat$ by $BagNat$ proceeds in three steps:

Extend: $\mathcal{E}_{Bag \to Set}$: **Spec**($Sig[BagNat]$) $\to$ **Spec**($\Sigma BagNat'$) $=_{def}$
λX: $Sig[BagNat]$. **enrich** X **by**
data opns $\in$: $nat, bag \to bool$
axioms $a \in B = count(a, B) > 0$

Derive: $\mathcal{D}_{Bag \to Set}$: **Spec**($\Sigma BagNat'$) $\to$ **Spec**($Sig[SetNat]$) $=_{def}$
λX: $\Sigma BagNat'$. **derive from** X **by** σ

where σ renames the sorts and operations in $Sig[SetNat]$ to those in $\Sigma BagNat'$ by renaming set to bag and leaving the other names as they were (note that $count$ is hidden in this step).

Quotient: $\mathcal{Q}_{Bag\to Set}$: $\mathbf{Spec}(Sig[SetNat]) \to \mathbf{Spec}(Sig[SetNat]) =_{def}$
$\lambda X\colon Sig[SetNat].$ **quotient** X **wrt**
$\{\forall a\colon nat, S\colon set.add(a, add(a, S)) \sim add(a, S)\}$

Notice that any specification-building operation $\omega\colon \mathbf{Spec}(\Sigma) \to \mathbf{Spec}(\Sigma')$ may be identified with the parameterised specification $\lambda X\colon \Sigma.\omega(X)$. This allows us to use the syntax of parameterised specifications to define specification-building operations (constructors in particular) as above.

We now have:

$$SetNat \underset{\mathcal{E}_{Bag\to Set};\mathcal{D}_{Bag\to Set};\mathcal{Q}_{Bag\to Set}}{\rightsquigarrow} BagNat$$

This may be equivalently (and perhaps more directly) presented as a single simple refinement step. Namely, we have just stated that $SetNat$ refines to the following specification:

quotient
  derive from
    enrich
      $BagNat$
    by data opns $\in\colon nat, bag \to bool$
      axioms $a \in B = count(a, B) > 0$
  by σ
wrt $\{\forall a\colon nat, S\colon set.add(a, add(a, S)) = add(a, S)\}$

We hope that this makes the notation we used to define $\mathcal{E}_{Bag\to Set}$, $\mathcal{D}_{Bag\to Set}$ and $\mathcal{Q}_{Bag\to Set}$ clear. We prefer the previous formulation of the same implementation step, since it clearly separates the constructive part of the refined specification ($\mathcal{E}_{Bag\to Set};\mathcal{D}_{Bag\to Set};\mathcal{Q}_{Bag\to Set}$) from its non-constructive, yet-to-be-implemented part ($BagNat$).

Of course, the claim that $BagNat$ implements $SetNat$ via $\mathcal{E}_{Bag\to Set};\mathcal{D}_{Bag\to Set};\mathcal{Q}_{Bag\to Set}$ requires a proof. We have to show that given any model $BAG \in Mod[BagNat]$, $\mathcal{Q}_{Bag\to Set}(\mathcal{D}_{Bag\to Set}(\mathcal{E}_{Bag\to Set}(BAG)))$ is a model of $SetNat$. In this case the proof is relatively straightforward and easy, albeit tedious, based directly on our definitions of the specific constructions involved, arguing in terms of how the constructors transform individual models of $BagNat$. We omit it here, and we omit similar proofs in the sequel. It should be pointed out, however, that if the methodology we present is to be used in practice, some systematic and uniform techniques for constructing proofs of this kind must be developed. This task is separate from the development of the general model-theoretic framework which is the topic of this paper.

Next we implement $BagNat$ by the following specification of lists of natural numbers:

$ListNat =_{def}$ **restrict** (**enrich** Nat **by**
  sorts $list$
  opns $nil\colon \to list$
    $cons\colon nat, list \to list$
    $null\colon list \to bool$

hd: $list \to nat$
tl: $list \to list$
axioms $D(nil)$
$D(cons(a, L))$
$null(nil) = true$
$null(cons(a, L)) = false$
$hd(cons(a, L)) = a$
$tl(cons(a, L)) = L$)
on $\{list\}$

The constructor implementation of $BagNat$ by $ListNat$ proceeds in three steps. The idea is that a finite bag is represented by a list containing at the n^{th} position the number of times n occurs in the bag.

Extend: $\mathcal{E}_{List \to Bag}$: **Spec**($Sig[ListNat]$) $\to$ **Spec**($\Sigma ListNat'$) $=_{def}$
λX: $Sig[ListNat]$.
enrich X **by**
data opns nth: $nat, list \to nat$
put: $nat, list \to list$
axioms $null(L) = true \Rightarrow nth(n, L) = 0$
$null(L) = false \Rightarrow nth(0, L) = hd(L)$
$null(L) = false \Rightarrow nth(succ(n), L) = nth(n, tl(L))$
$null(L) = true \Rightarrow put(0, L) = cons(succ(0), L)$
$null(L) = false \Rightarrow put(0, L) = cons(succ(hd(L)), tl(L))$
$null(L) = true \Rightarrow put(succ(n), L) = cons(0, put(n, L))$
$null(L) = false \Rightarrow put(succ(n), L) = cons(hd(L), put(n, tl(L)))$

where $\Sigma ListNat'$ is the extension of $Sig[ListNat]$ by the operation names nth: $nat, list \to nat$ and put: $nat, list \to list$.

Derive: $\mathcal{D}_{List \to Bag}$: **Spec**($\Sigma ListNat'$) $\to$ **Spec**($Sig[BagNat]$) $=_{def}$
λX: $\Sigma ListNat'$. **derive from** X **by** σ

where σ renames the sorts and operations in $Sig[BagNat]$ to those in $\Sigma ListNat'$ by renaming bag to $list$, $\emptyset$ to nil, add to put, $count$ to nth and $isempty$ to $null$ and leaving the other names as they were. Note that hd, tl and $cons$ are hidden in this step.

Restrict: $\mathcal{R}_{List \to Bag}$: **Spec**($Sig[BagNat]$) $\to$ **Spec**($Sig[BagNat]$) $=_{def}$
λX: $Sig[BagNat]$. **restrict** X **on** $\{bag\}$

Intuitively, this removes from models those bags which cannot be constructed using $\emptyset$ and add, i.e. lists with trailing 0's. This last step is necessary only because we started with the explicit requirement that models of $BagNat$ (and initially of $SetNat$) are reachable.

We now have:

$$BagNat \underset{\mathcal{E}_{List\to Bag};\mathcal{D}_{List\to Bag};\mathcal{R}_{List\to Bag}}{\leadsto} ListNat$$

Putting this together with the previous example using the vertical composition theorem, we get:

$$SetNat \underset{\mathcal{E}_{List\to Bag};\mathcal{D}_{List\to Bag};\mathcal{R}_{List\to Bag}\ ;\ \mathcal{E}_{Bag\to Set};\mathcal{D}_{Bag\to Set};\mathcal{Q}_{Bag\to Set}}{\leadsto} ListNat$$

Although both of the implementations we composed above are in the form required in [EKMP 82] (**extend-derive-restrict-quotient**) the result implementation is not in this form. In this case the result may be converted to an implementation of this form but this does not matter in our framework; the implementation is acceptable as it is.

6 Abstractors and implementations

It is often possible to abstract away from some of the details of the user's original specification without violating the real intention behind it. This is the idea behind the specification technique known in software engineering as *abstract model specification* [LB 77], in which the user defines in a more or less concrete fashion a model which gives the desired results with the intention that any program giving the same answers is acceptable. An example (not from software engineering) is in [AMRW 85] where the semantics of a set of basic operations on transition systems (which are sufficient to define e.g. SMoLCS [AR 83]) is described by first presenting an operational semantics and then abstracting in two different ways to yield the input-output semantics and strong equivalence semantics. Our specification of sets of natural numbers in the last section may be regarded in the same way — we do not really care whether an algebra satisfies all the axioms given there. An algebra is an acceptable realisation of this specification as long as the membership relation behaves properly (i.e. gives the right answers for every choice of argument).

This theme has been discussed in [GGM 76], [BM 81], [Rei 81], [GM 82], [Sch 86], [Kam 83], [MG 83], [ST 85a,87a] and elsewhere; the idea goes back (at least) to work on automata theory in the 1950's [Moo 56].

To formalize these ideas we will consider another class of specification-building operations called abstractors. Intuitively, any equivalence relation on Σ-algebras determines a specification-building operation which relaxes interpretation of any Σ-specification SP by admitting as a model any Σ-algebra which is equivalent to a model of SP. Seen another way, the abstractor closes the class of models of a specification under this equivalence.

Definition 6.1 *An* abstractor determined by *an equivalence relation* $\equiv\ \subseteq \mathbf{PAlg}(\Sigma) \times \mathbf{PAlg}(\Sigma)$ *is a specification-building operation* $\alpha_\equiv\colon \mathbf{Spec}(\Sigma) \to \mathbf{Spec}(\Sigma)$ *where for any Σ-specification SP,*

$$Sig[\alpha_\equiv(SP)] = \Sigma$$
$$Mod[\alpha_\equiv(SP)] = \{A \in \mathbf{PAlg}(\Sigma) \mid \exists A' \in Mod[SP].A \equiv A'\}$$

In the sequel we will omit the subscript $\equiv$ when there is no danger of confusion. Also, if α is known we denote the abstraction equivalence *which determines it by $\equiv_\alpha$.*

A few easy facts follow immediately from this definition.

Fact 6.2 *Abstractors are monotonic.* □

Fact 6.3 *Abstractors preserve and reflect consistency of specifications. That is, for any abstractor* α: $\mathbf{Spec}(\Sigma) \to \mathbf{Spec}(\Sigma)$ *and* Σ*-specification* SP, SP *is consistent iff* $\alpha(SP)$ *is consistent.* □

Fact 6.4 *Abstractors are idempotent, i.e. for any abstractor* α: $\mathbf{Spec}(\Sigma) \to \mathbf{Spec}(\Sigma)$, $\alpha(\alpha(SP))$ *has the same class of models as* $\alpha(SP)$. □

Remark In general, abstractors are not closed under composition, i.e. there are abstractors $\alpha 1$: $\mathbf{Spec}(\Sigma) \to \mathbf{Spec}(\Sigma)$ and $\alpha 2$: $\mathbf{Spec}(\Sigma) \to \mathbf{Spec}(\Sigma)$ such that the composition $\alpha 1;\alpha 2$: $\mathbf{Spec}(\Sigma) \to \mathbf{Spec}(\Sigma)$ is not an abstractor.

Counterexample Let $\mathbf{PAlg}(\Sigma) = C_1 \cup C_2 \cup C_3 \cup C_4$ where $C_1, \ldots, C_4$ are disjoint classes of Σ-algebras. Consider the equivalences

$$\begin{aligned} &\equiv\ = (C_1 \times C_1) \cup (C_2 \times C_2) \cup (C_3 \times C_3) \cup (C_4 \times C_4) \\ &\equiv_1\ = \ \equiv \cup\, (C_1 \times C_2) \cup (C_2 \times C_1) \cup (C_3 \times C_4) \cup (C_4 \times C_3) \\ &\equiv_2\ = \ \equiv \cup\, (C_2 \times C_3) \cup (C_3 \times C_2) \end{aligned}$$

$\alpha_{\equiv_1};\alpha_{\equiv_2}$ is not idempotent, in contradiction to fact 6.4: $\alpha_{\equiv_2}(\alpha_{\equiv_1}(C_1)) = C_1 \cup C_2 \cup C_3 \neq \mathbf{PAlg}(\Sigma) = \alpha_{\equiv_2}(\alpha_{\equiv_1}(\alpha_{\equiv_2}(\alpha_{\equiv_1}(C_1))))$.

□

This fact is neither surprising nor disturbing; we will not in fact have occasion to compose abstractors.

Example 6.5 (observational abstraction) For any Σ-specification SP and set W of ground Σ-terms, the semantics of the specification **abstract** SP **wrt** W is as follows [SW 83]:

$$\begin{aligned} &Sig[\mathbf{abstract}\ SP\ \mathbf{wrt}\ W] = \Sigma \\ &Mod[\mathbf{abstract}\ SP\ \mathbf{wrt}\ W] = \{A \in \mathbf{PAlg}(\Sigma) \mid \exists A' \in Mod[SP].A \equiv_W A'\} \end{aligned}$$

where for any two algebras $A, A' \in \mathbf{PAlg}(\Sigma)$, $A \equiv_W A'$ iff:

- for all $t \in W$, $A \models D(t)$ iff $A' \models D(t)$, and
- for all $s \in sorts(\Sigma)$ and all $t, t' \in W_s$, $A \models t = t'$ iff $A' \models t = t'$.

Intuitively, W is the set of Σ-terms which represent computations the user is allowed to perform. We do not want to distinguish between algebras in which all these computations give the same results. A similar idea in the context of concurrent processes appears in [deNH 84].

This can be generalised in two ways. First, instead of a set of Σ-terms for which we can "observe" definedness and equality, we can consider more complicated observations: arbitrary Σ-sentences. Two

Σ-algebras are equivalent with respect to a set of Σ-sentences if they satisfy exactly the same sentences from this set. This more general notion of observational equivalence was introduced and analysed in [ST 87a] (cf. [Pep 83]). Second, since we have only considered ground terms here, the equivalence takes into account only reachable subparts of algebras. To take "junk" into account, one can consider equivalence with respect to a set of Σ-terms (or more generally, Σ-formulae) with free variables; see [SW 83] and [ST 87a] for details. For simplicity, we discuss only the simplest version of observational equivalence here. $\square$

Example 6.6 (behavioural abstraction) An important special case of observational abstraction is behavioural abstraction. For any Σ-specification SP and set $OBS \subseteq sorts(\Sigma)$ of sorts, the semantics of the specification **behaviour** SP **wrt** OBS is as follows [SW 83], [ST 86a,87a]:

$$Sig[\mathbf{behaviour}\ SP\ \mathbf{wrt}\ OBS] = \Sigma$$
$$Mod[\mathbf{behaviour}\ SP\ \mathbf{wrt}\ OBS] = \{A \in \mathbf{PAlg}(\Sigma) \mid \exists A' \in Mod[SP]. A \equiv_{OBS} A'\}$$

where the equivalence $\equiv_{OBS}$ is just $\equiv_W$ for W the set of all ground Σ-terms of sorts in OBS. Intuitively, OBS is the set of external sorts, visible to the user. The result of any computation leading to any of these sorts is observable. $\square$

The above considerations indicate that often we are satisfied with implementing a given specification up to an abstraction equivalence. This leads to the following notion:

Definition 6.7 (abstractor implementation) *A Σ-specification SP is implemented by a Σ'-specification SP'* wrt *an abstractor α: $\mathbf{Spec}(\Sigma) \to \mathbf{Spec}(\Sigma)$* via *a constructor κ: $\mathbf{Spec}(\Sigma') \to \mathbf{Spec}(\Sigma)$, written $SP \underset{\kappa}{\overset{\alpha}{\rightsquigarrow}} SP'$, if $\alpha(SP) \rightsquigarrow \kappa(SP')$. In other words, $SP \underset{\kappa}{\overset{\alpha}{\rightsquigarrow}} SP'$ if κ transforms every model of SP' to an algebra which is $\equiv_\alpha$-equivalent to a model of SP.*

Every constructor implementation $SP \underset{\kappa}{\rightsquigarrow} SP'$ is also an abstractor implementation $SP \underset{\kappa}{\overset{\alpha_\equiv}{\rightsquigarrow}} SP'$ where $\equiv$ is the identity relation on $\mathbf{PAlg}(Sig[SP])$. Also, if $SP \underset{\kappa}{\overset{\alpha_\equiv}{\rightsquigarrow}} SP'$ is an abstractor implementation then so is $SP \underset{\kappa}{\overset{\alpha_{\equiv'}}{\rightsquigarrow}} SP'$ for any $\equiv' \supseteq \equiv$.

If in the above definition, α is behavioural abstraction, then intuitively speaking we are implementing the behaviour of SP rather than SP itself. This subsumes the notions of implementation in [GM 82], [Sch 86] and [BMPW 86].

The abstractor α cannot be chosen arbitrarily; the choice depends on the specification SP and the context in which it is to be used. If α abstracts too much then the implementation will be useless — for example if $\equiv$ is the total equivalence on $\mathbf{PAlg}(\Sigma)$ then $Mod[\alpha_\equiv(SP)] = \mathbf{PAlg}(\Sigma)$ and so $SP \underset{\kappa}{\overset{\alpha_\equiv}{\rightsquigarrow}} SP'$ for *any* SP' and constructor κ: $\mathbf{Spec}(Sig[SP']) \to \mathbf{Spec}(Sig[SP])$.

Let us consider now the problem of vertical composability of abstractor implementations. Suppose $SP \underset{\kappa}{\overset{\alpha}{\rightsquigarrow}} SP'$ and $SP' \underset{\kappa'}{\overset{\alpha'}{\rightsquigarrow}} SP''$. We would like to be able to conclude that $SP \underset{\kappa';\kappa}{\overset{\alpha}{\rightsquigarrow}} SP''$. Note that according to the above argument we assume that α was chosen appropriately for the context in which SP is to be used and so we do not want to change it even when composing implementations.

In general, there is no hope for such a result. If α' is too "liberal", there is no reason to expect that κ transforms any $\alpha'(SP')$-model to a model of $\alpha(SP)$. However, the following theorem does hold:

Theorem 6.8 (vertical composition) *If* $SP \overset{\alpha}{\underset{\kappa}{\rightsquigarrow}} SP'$ *and* $SP' \overset{\alpha'}{\underset{\kappa'}{\rightsquigarrow}} SP''$ *then* $SP \overset{\alpha}{\underset{\kappa';\kappa}{\rightsquigarrow}} SP''$ *provided* κ preserves the abstraction equivalences, *i.e. for any two algebras* $A1, A2 \in \mathbf{PAlg}(Sig[SP'])$ *if* $A1 \equiv_{\alpha'} A2$ *then* $\kappa(A1) \equiv_\alpha \kappa(A2)$.

Proof By definition, $\alpha'(SP') \rightsquigarrow \kappa'(SP'')$. Then $\kappa(\alpha'(SP')) \rightsquigarrow (\kappa';\kappa)(SP'')$ follows from the monotonicity of κ. By vertical composability of $\rightsquigarrow$ it suffices to show that $\alpha(SP) \rightsquigarrow \kappa(\alpha'(SP'))$, i.e. $Mod[\alpha(SP)] \supseteq Mod[\kappa(\alpha'(SP'))] = \kappa(Mod[\alpha'(SP')])$. Now, for any model $A' \in Mod[\alpha'(SP')]$, there is $A1' \in Mod[SP']$ such that $A' \equiv_{\alpha'} A1'$. Since κ preserves the abstraction equivalences, $\kappa(A') \equiv_\alpha \kappa(A1')$. Now, $\kappa(A1') \in Mod[\alpha(SP)]$ since $SP \overset{\alpha}{\underset{\kappa}{\rightsquigarrow}} SP'$ and so $\kappa(A') \in Mod[\alpha(SP)]$. □

A methodological conclusion from this theorem is that the development process should proceed as follows: starting from a specification SP considered in a context for which an abstractor α is appropriate, we (abstractor) implement SP, say $SP \overset{\alpha}{\underset{\kappa}{\rightsquigarrow}} SP'$. The next step should be to establish the appropriate abstractor up to which SP' may be considered by "pushing $\equiv_\alpha$ through κ". Namely, from the above theorem it follows that this should be the abstractor determined by the equivalence $\kappa^{-1}(\equiv_\alpha)$ where for $A, A' \in \mathbf{PAlg}(Sig[SP'])$, $A\ \kappa^{-1}(\equiv_\alpha)\ A'$ iff $\kappa(A) \equiv_\alpha \kappa(A')$ (it is trivial to show that $\kappa^{-1}(\equiv_\alpha)$ defined in this way is an equivalence). Then, we can proceed with the development of SP' in the context of the abstractor determined by $\kappa^{-1}(\equiv_\alpha)$. (Actually, any equivalence finer than $\kappa^{-1}(\equiv_\alpha)$ will do.) Similar ideas in the context of concurrent processes appear in [Lar 86].

Corollary 6.9 *If* $SP_0 \overset{\alpha_1}{\underset{\kappa_1}{\rightsquigarrow}} \cdots \overset{\alpha_n}{\underset{\kappa_n}{\rightsquigarrow}} SP_n$ *and* $\equiv_{\alpha_2} \subseteq \kappa_1^{-1}(\equiv_{\alpha_1})$ *and* $\cdots$ *and* $\equiv_{\alpha_n} \subseteq \kappa_{n-1}^{-1}(\equiv_{\alpha_{n-1}})$ *then* $SP_0 \overset{\alpha_1}{\underset{\kappa_n;\cdots;\kappa_1}{\rightsquigarrow}} SP_n$. □

In practice, it is often convenient to use a sharper version of the above results. It is not really necessary for constructors to preserve the abstraction equivalences on all algebras; the results hold if the constructors preserve the equivalences between models of the appropriate specifications (e.g. in the vertical composition theorem it is sufficient that $\kappa(A1) \equiv_\alpha \kappa(A2)$ for any $A1 \in \mathbf{PAlg}(Sig[SP'])$ and $A2 \in Mod[SP']$ such that $A1 \equiv_{\alpha'} A2$).

The requirement in the vertical composition theorem that the constructors preserve abstraction equivalences is just the same as the requirement in [Sch 86] that constructors in implementation steps (which correspond to *implementation cells*, in his terminology) be *stable*. A difference between the approach in [Sch 86] and ours is that he considers a fixed abstraction equivalence between all algebras of a given signature.

In the rest of this section, we show that vertical composition and the above methodological remarks may work in practice. On one hand, the constructors we have introduced do preserve appropriate (observational) equivalences; and on the other hand, we show how to push standard observational equivalences in a satisfactory way through the constructors we have defined. By "in a satisfactory way" we mean that although in some cases we do not characterise the result of pushing an equivalence through a constructor exactly, we describe instead a finer equivalence which is also sufficient as mentioned already.

Lemma 6.10 (derive) *For any signature morphism* $\sigma\colon \Sigma 1 \to \Sigma 2$ *and set* W *of ground* $\Sigma 2$*-terms,* $\mathcal{D}_\sigma^{-1}(\equiv_W) = \equiv_{\sigma(W)}$*, where* $\mathcal{D}_\sigma\colon \mathbf{Spec}(\Sigma 2) \to \mathbf{Spec}(\Sigma 1) =_{def} \lambda X\colon \Sigma 2.$ **derive from** X **by** σ*. (We justified this notation in the last section.)*

Proof Recall that $\mathcal{D}_\sigma$ (viewed as a function) is the reduct functor $\_|_\sigma\colon \mathbf{PAlg}(\Sigma 2) \to \mathbf{PAlg}(\Sigma 1)$. Let $A, A' \in \mathbf{PAlg}(\Sigma 2)$. We have $A \equiv_{\sigma(W)} A'$ iff: for all $t \in W$, $A \models D(\sigma(t))$ iff $A' \models D(\sigma(t))$ and for all $t_1, t_2 \in W_s$, $A \models \sigma(t_1) = \sigma(t_2)$ iff $A' \models \sigma(t_1) = \sigma(t_2)$. By the satisfaction lemma, this is equivalent to: for all $t \in W$, $A|_\sigma \models D(t)$ iff $A'|_\sigma \models D(t)$ and for all $t_1, t_2 \in W_s$, $A|_\sigma \models t_1 = t_2$ iff $A'|_\sigma \models t_1 = t_2$, i.e. $A|_\sigma \equiv_W A'|_\sigma$. □

Lemma 6.11 (restrict) *For any signature* Σ*,* $S \subseteq sorts(\Sigma)$*, set* W *of ground* Σ*-terms and* Σ*-algebra* A*,* $A \equiv_W \mathcal{R}_S(A)$*, where* $\mathcal{R}_S\colon \mathbf{Spec}(\Sigma) \to \mathbf{Spec}(\Sigma) =_{def} \lambda X\colon \Sigma.$ **restrict** X **on** S*.*

Proof Obvious since for any $A \in \mathbf{PAlg}(\Sigma)$ and $t \in W$, $A \models D(t)$ iff $\mathcal{R}_S(A) \models D(t)$ and moreover if $A \models D(t)$ then the values of t in A and in $\mathcal{R}_S(A)$ are the same. □

The above lemma gives directly a characterisation of the result of pushing observational equivalence through **restrict** constructors. Perhaps more importantly, it directly implies that **restrict** steps may be skipped if we use abstractor implementations.

Corollary 6.12 *Under the assumptions of lemma 6.11,* $\mathcal{R}_S^{-1}(\equiv_W) = \equiv_W$*.* □

Corollary 6.13 *Under the assumptions of lemma 6.11, if* α *is the abstractor determined by* $\equiv_W$*, then for any* Σ*-specifications* SP *and* SP'*,* $SP \underset{\mathcal{R}_S}{\overset{\alpha}{\rightsquigarrow}} SP'$ *implies* $SP \underset{id}{\overset{\alpha}{\rightsquigarrow}} SP'$*.* □

It is worth pointing out that the above corollary also allows us to throw out **restrict** steps "in the middle" of the development process (provided that the intermediate equivalence used in this step satisfies the assumptions of lemma 6.11). Namely, given $SP \underset{\kappa';\mathcal{R}_S;\kappa}{\overset{\alpha}{\rightsquigarrow}} SP'$, if this implementation can be decomposed into $SP \underset{\kappa}{\overset{\alpha}{\rightsquigarrow}} SP1 \underset{\kappa';\mathcal{R}_S}{\overset{\alpha'}{\rightsquigarrow}} SP'$ where $\equiv_{\alpha'} \subseteq \kappa^{-1}(\equiv_\alpha)$ (and, say, $SP1 = \mathcal{R}_S(\kappa'(SP'))$) and $\equiv_{\alpha'}$ is observational equivalence with respect to a set of ground terms, then $SP \underset{\kappa}{\overset{\alpha}{\rightsquigarrow}} SP1 \underset{\kappa'}{\overset{\alpha'}{\rightsquigarrow}} SP'$ and hence $SP \underset{\kappa';\kappa}{\overset{\alpha}{\rightsquigarrow}} SP'$. This means that corollary 6.12 becomes superfluous since instead of using it to push equivalences through **restrict** steps we can just skip these steps entirely. This corresponds nicely to standard programming practice. If we happen to produce a data type with some junk elements, we are not forced to remove them before using the data type. Instead, we just use the data type as usual, pretending that the junk elements are not there.

The situation with **quotient** steps is similar. No program can ever force two different existing data values to be the same. At best, we can pretend that they are the same. This is possible only provided that they exhibit the same observable behaviour. Thus, we can remove **quotient** steps whenever they do not glue together elements having different observable behaviour.

Definition 6.14 *For any signature Σ, set E of Σ-equations, set W of ground Σ-terms and Σ-algebra A, we say that E is* observably trivial on A *(*wrt W*) if $A \equiv_W A/E$. We say that E is observably trivial on a Σ-specification SP if it is observably trivial on each model of SP. We say that E is* behaviourally trivial on A *(resp. SP)* wrt *a set $OBS \subseteq sorts(\Sigma)$ of observable sorts if it is observably trivial on A (resp. SP) with respect to the set of all ground terms of sorts in OBS.*

As for **restrict**, the above definition leads directly to a (trivial) characterisation of the result of pushing observational equivalence through the **quotient** constructor in the context of specifications which guarantee observable triviality of the equations by which we quotient. More importantly, however:

Lemma 6.15 (quotient) *Under the assumptions of definition 6.14, if α is the abstractor determined by $\equiv_W$ and SP, SP' are Σ-specifications such that E is observably trivial on SP' wrt W and $SP \underset{\mathcal{Q}_E}{\overset{\alpha}{\rightsquigarrow}} SP'$ where $\mathcal{Q}_E$: $\mathbf{Spec}(\Sigma) \to \mathbf{Spec}(\Sigma) =_{def} \lambda X{:}\ \Sigma.$ $\mathbf{quotient}$ X $\mathbf{wrt}$ E, then $SP \underset{id}{\overset{\alpha}{\rightsquigarrow}} SP'$.* □

Proof Trivial. □

The above definition of observable triviality does not give any hints on how to prove that a set of equations is indeed observably trivial on a given specification. We do not study this problem here, just as we do not treat techniques for proving implementation steps correct. We formulate the following easy lemma to indicate what kind of results may be expected and useful here.

Lemma 6.16 *Consider a signature Σ, set E of Σ-equations, set $OBS \subseteq sorts(\Sigma)$ of observable sorts and a* reachable, total *Σ-algebra A. Let $\sim^E$ be the least congruence on ground Σ-terms generated by E (i.e. $\sim^E =_{def} \sim^E_{T_\Sigma}$). If for any two ground terms t, t' of an observable sort, $A \models t = t'$ whenever $t \sim^E t'$ then E is behaviourally trivial on A wrt OBS.*

Proof It is easy to see that since A is reachable and all terms have a defined value in A,

$$\sim^E_A = \{\langle t_A, t'_A\rangle \mid t \sim^E t'\}.$$

Hence, by our assumption, the congruence $\sim^E_A$ is the identity relation on the carriers of A of observable sorts and thus indeed $A \equiv_{OBS} A/E$. □

Definition 6.17 *For any signature morphism $\sigma{:}\ \Sigma \to \Sigma'$, constructor κ: $\mathbf{Spec}(\Sigma) \to \mathbf{Spec}(\Sigma')$ and sets W and W' of ground Σ- and ground Σ'-terms respectively, κ is* observably sufficiently complete *(*wrt W, W'*) if for any term $t' \in W'$, either for all $A \in \mathbf{PAlg}(\Sigma)$, $\kappa(A) \not\models D(t')$ or there exists a term $t \in W$ such that for all $A \in \mathbf{PAlg}(\Sigma)$, $\kappa(A) \models t' = \sigma(t)$.*

Typically, we will consider sets W and W' such that observable sufficient completeness is a weaker condition than sufficient completeness, which corresponds to the case where W' is the set of all ground Σ'-terms of the sorts $\sigma(S)$ for $S =_{def} sorts(\Sigma)$ and W is the set of all ground Σ-terms.

Definition 6.18 *For any signature morphism $\sigma\colon \Sigma \to \Sigma'$, constructor $\kappa\colon$ $\mathbf{Spec}(\Sigma) \to \mathbf{Spec}(\Sigma')$ and set W of ground Σ-terms, κ is* observably persistent *(wrt W) if for all terms $t1, t2 \in W$ of the same sort and any $A \in \mathbf{PAlg}(\Sigma)$, $\kappa(A) \models \sigma(t1) = \sigma(t2)$ iff $A \models t1 = t2$ and $\kappa(A) \models D(\sigma(t1))$ iff $A \models D(t1)$.*

Notice that observable persistency is a weaker condition than the standard persistency, i.e. that for any $A \in \mathbf{PAlg}(\Sigma)$, $\kappa(A)|_\sigma = A$. Namely, the satisfaction lemma implies that if κ is persistent then it is observably persistent.

The following lemma applies to all synthesizing constructors, including for example the **extend** constructor.

Lemma 6.19 (synthesize) *For any signature morphism $\sigma\colon \Sigma \to \Sigma'$ which is injective on sorts, constructor $\kappa\colon$ $\mathbf{Spec}(\Sigma) \to \mathbf{Spec}(\Sigma')$ and sets W and W' of ground Σ- and Σ'-terms respectively, if κ is observably sufficiently complete wrt W, W' and observably persistent wrt W then $\kappa^{-1}(\equiv_{W'}) \supseteq \equiv_W$. Moreover, if in addition W is a minimal set such that observable sufficient completeness holds then $\kappa^{-1}(\equiv_{W'}) = \equiv_W$.*

Proof Let $A1, A2 \in \mathbf{PAlg}(\Sigma)$. Assume $A1 \equiv_W A2$; we prove that $\kappa(A1) \equiv_{W'} \kappa(A2)$.

1. For any $t' \in W'$, $\kappa(A1) \models D(t')$ iff $\kappa(A2) \models D(t')$: If t' is defined in no algebra $\kappa(A)$ for $A \in \mathbf{PAlg}(\Sigma)$ then the equivalence is obvious. Otherwise, let $t \in W$ be such that for any $A \in \mathbf{PAlg}(\Sigma)$, $\kappa(A) \models t' = \sigma(t)$. Now $\kappa(A1) \models D(t')$ iff $\kappa(A1) \models D(\sigma(t))$ iff $A1 \models D(t)$ (by observable persistency) iff $A2 \models D(t)$ iff $\kappa(A2) \models D(\sigma(t))$ iff $\kappa(A2) \models D(t')$.

2. For any $t1', t2' \in W'$ of the same sort, $\kappa(A1) \models t1' = t2'$ iff $\kappa(A2) \models t1' = t2'$:

   "$\Leftarrow$": If $t1'$ and $t2'$ are undefined in $\kappa(A2)$ then they are undefined in $\kappa(A1)$ as well by (1) and so $\kappa(A1) \models t1' = t2'$. So assume both $t1'$ and $t2'$ are defined in $\kappa(A2)$. Let $t1, t2 \in W$ be such that for all $A \in \mathbf{PAlg}(\Sigma)$, $\kappa(A) \models t1' = \sigma(t1)$ and $\kappa(A) \models t2' = \sigma(t2)$. Since σ is injective on sorts, $t1$ and $t2$ are of the same sort. We have $\kappa(A2) \models \sigma(t1) = \sigma(t2)$ hence $A2 \models t1 = t2$ by observable persistency and so $A1 \models t1 = t2$ and so $\kappa(A1) \models \sigma(t1) = \sigma(t2)$ which finally implies $\kappa(A1) \models t1' = t2'$.

   "$\Rightarrow$": By symmetry.

Moreover, notice that if W is a minimal set of ground Σ-terms such that observable sufficient completeness holds, then for all $t \in W$ there exists $t' \in W'$ such that for all $A \in \mathbf{PAlg}(\Sigma)$, $\kappa(A) \models t' = \sigma(t)$; otherwise we could have removed t from W without violating observable sufficient completeness and observable persistency. We can now prove $\kappa(A1) \equiv_{W'} \kappa(A2)$ implies $A1 \equiv_W A2$ using the same arguments as above. □

Notice that although in order for κ to be a well-defined constructor we require that it is defined on all Σ-algebras, in the development process κ will be applied to a particular specification SP in the context of an abstractor α. In this situation it is sufficient to show that κ is observably sufficiently complete and observably persistent only on models of $\alpha(SP)$.

As remarked already, constructor implementation using the **derive**, **restrict**, **quotient** and **extend** constructors subsumes many of the notions of implementation in the literature. The above lemmas imply that the extension of any of these notions to a corresponding notion of abstractor implementation goes through smoothly.

Lemma 6.20 (amalgamated union) *Let* $\kappa 1\colon \mathbf{Spec}(\Sigma) \xrightarrow{\sigma 1} \mathbf{Spec}(\Sigma 1)$ *and* $\kappa 2\colon \mathbf{Spec}(\Sigma) \xrightarrow{\sigma 2} \mathbf{Spec}(\Sigma 2)$ *be persistent constructors,* $W, W1, W2$ *be sets of ground* Σ-*,* $\Sigma 1$- *and* $\Sigma 2$-*terms respectively such that* $\kappa 1$ *is observably sufficiently complete wrt* $W, W1$ *and* $\kappa 2$ *is observably sufficiently complete wrt* $W, W2$*. Recall that* $\kappa =_{def} \kappa 1 + \kappa 2\colon \mathbf{Spec}(\Sigma) \to \mathbf{Spec}(\Sigma')$*, where*

$$\begin{tikzcd}
\Sigma \arrow[r, "\sigma 1"] \arrow[d, "\sigma 2"'] & \Sigma 1 \arrow[d, "\sigma 1'"] \\
\Sigma 2 \arrow[r, "\sigma 2'"'] & \Sigma'
\end{tikzcd}$$

is a pushout in **Sign***, is a persistent synthesizing constructor (along* $\sigma 1;\sigma 1' = \sigma 2;\sigma 2'$*) such that for* $A \in \mathbf{PAlg}(\Sigma)$*,* $\kappa(A)$ *is the unique* Σ'*-algebra such that* $\kappa(A)|_{\sigma 1'} = \kappa 1(A)$ *and* $\kappa(A)|_{\sigma 2'} = \kappa 2(A)$*. Under these assumptions,* κ *is observably sufficiently complete wrt* W, W' *where* $W' =_{def} \sigma 1'(W1) \cup \sigma 2'(W2)$*.*

Proof Let $t' \in W'$. Suppose $t' = \sigma 1'(t1)$ for $t1 \in W1$ (the case $t' = \sigma 2'(t2)$ for $t2 \in W2$ is symmetric) and that $\kappa(A) \models D(t')$ for some $A \in \mathbf{PAlg}(\Sigma)$. Then also $\kappa 1(A) \models D(t1)$ and so, since $\kappa 1$ is observably sufficiently complete, there exists $t \in W$ such that for any $A \in \mathbf{PAlg}(\Sigma)$, $\kappa 1(A) \models t1 = \sigma 1(t)$. Now for any $A \in \mathbf{PAlg}(\Sigma)$, $\kappa(A) \models \sigma 1'(t1) = \sigma 1'(\sigma 1(t))$ iff $\kappa(A)|_{\sigma 1'} \models t1 = \sigma 1(t)$ by the satisfaction lemma. However, by the definition of κ, $\kappa(A)|_{\sigma 1'} = \kappa 1(A)$ and so $\kappa(A) \models t' = \sigma 1'(\sigma 1(t))$. $\Box$

Notice that we have assumed that $\kappa 1$ and $\kappa 2$ are persistent constructors as required in the definition of $\kappa 1 + \kappa 2$. However, as noted in the remarks after that definition, the constructor $\kappa 1 + \kappa 2$ will in practice be applied to a particular Σ-specification SP in which case it is sufficient that $\kappa 1$ and $\kappa 2$ be persistent and observably sufficiently complete only on models of SP (up to the relevant abstraction equivalence).

Corollary 6.21 *Under the assumptions of lemma 6.20,* $\kappa^{-1}(\equiv_{W'}) \supseteq \equiv_W$*.*

Proof By lemma 6.20, $\kappa = \kappa 1 + \kappa 2$ is observably sufficiently complete. Moreover, it is observably persistent since it is persistent by fact 4.11. The result follows directly by lemma 6.19. $\Box$

Lemma 6.22 (translation of a constructor) *Let*

$$\begin{tikzcd}
\Sigma \arrow[r, "\sigma 1"] \arrow[d, "\sigma 2"'] & \Sigma 1 \arrow[d, "\sigma 1'"] \\
\Sigma 2 \arrow[r, "\sigma 2'"'] & \Sigma'
\end{tikzcd}$$

be a pushout in **Sign***, let* $W, W1, W2$ *be sets of ground* Σ-*,* $\Sigma 1$- *and* $\Sigma 2$-*terms respectively, and let* $\kappa 1$: $\mathbf{Spec}(\Sigma) \xrightarrow{\sigma 1} \mathbf{Spec}(\Sigma 1)$ *be a persistent constructor. If* $\kappa 1$ *is observably sufficiently complete wrt* $W, W1$ *and* $\sigma 2(W) \subseteq W2$ *then* $\sigma 2(\kappa 1)$: $\mathbf{Spec}(\Sigma 2) \to \mathbf{Spec}(\Sigma')$ *is observably sufficiently complete wrt* $W2, W'$ *where* $W' = \sigma 1'(W1) \cup \sigma 2'(W2)$.

Proof Let $t' \in W'$. If $t' = \sigma 2'(t2)$ for $t2 \in W2$, there is nothing to prove. Otherwise, $t' = \sigma 1'(t1)$ for some $t1 \in W1$. If for some $A2 \in \mathbf{PAlg}(\Sigma 2), \sigma 2(\kappa 1)(A2) \models D(t')$ then also $\kappa 1(A2|_{\sigma 2}) \models D(t1)$ and so, by observable sufficient completeness of $\kappa 1$, there exists $t \in W$ such that for any $A \in \mathbf{PAlg}(\Sigma)$, $\kappa 1(A) \models t1 = \sigma 1(t)$. Hence, for any $A2 \in \mathbf{PAlg}(\Sigma 2)$, $\sigma 2(\kappa 1)(A2) \models \sigma 1'(t1) = \sigma 1'(\sigma 1(t))$ by the satisfaction lemma, since $\sigma 2(\kappa 1)(A2)|_{\sigma 1'} = \kappa 1(A2|_{\sigma 2})$. Now, notice that $\sigma 2'(\sigma 2(t)) = \sigma 1'(\sigma 1(t))$, and so $\sigma 2(\kappa 1)(A2) \models t' = \sigma 2'(\sigma 2(t))$. Moreover, $\sigma 2(t) \in W2$. □

Corollary 6.23 *Under the assumptions of lemma 6.22,* $\sigma 2(\kappa 1)^{-1}(\equiv_{W'}) \supseteq \equiv_{W2}$.

Proof By lemma 6.22, $\sigma 2(\kappa 1)$ is observably sufficiently complete. Moreover, it is observably persistent since it is persistent (see example 4.12 in section 4). The result follows directly by lemma 6.19. □

Notice that again in practice it is sufficient to require that $\kappa 1$ be persistent and observably sufficiently complete only on (the relevant parts of) models of the specification its translation is applied to in the development process.

7 Examples of abstractor implementations

Recall from section 5 the development of the (constructor) implementation of sets of natural numbers by lists of natural numbers:

$$SetNat \underset{\mathcal{E}_{Bag\to Set};\mathcal{D}_{Bag\to Set};\mathcal{Q}_{Bag\to Set}}{\rightsquigarrow} BagNat \underset{\mathcal{E}_{List\to Bag};\mathcal{D}_{List\to Bag};\mathcal{R}_{List\to Bag}}{\rightsquigarrow} ListNat$$

As argued in the last section, we do not really need an exact implementation of $SetNat$; all we are interested in is the behaviour that $SetNat$ determines, i.e. we want to implement the specification **behaviour** $SetNat$ **wrt** $\{nat, bool\}$. Let

$$\mathcal{A}_{Set}\colon \mathbf{Spec}(Sig[SetNat]) \to \mathbf{Spec}(Sig[SetNat]) =_{def}$$
$$\lambda X\colon Sig[SetNat].\ \mathbf{behaviour}\ X\ \mathbf{wrt}\ \{nat, bool\}$$

be the abstractor. We then have (trivially):

$$SetNat \underset{\mathcal{E}_{Bag\to Set};\mathcal{D}_{Bag\to Set};\mathcal{Q}_{Bag\to Set}}{\overset{\mathcal{A}_{Set}}{\rightsquigarrow}} BagNat$$

We can now use lemma 6.15 simplify this implementation. Since by lemma 6.16 the equation used in the quotient step $\mathcal{Q}_{Bag\to Set}$ is behaviourally trivial on $\mathcal{D}_{Bag\to Set}(\mathcal{E}_{Bag\to Set}(BagNat))$ with respect to $\{nat, bool\}$ (notice that the *count* operation is no longer available here), we have:

$$SetNat \underset{\mathcal{E}_{Bag\to Set};\mathcal{D}_{Bag\to Set}}{\overset{\mathcal{A}_{Set}}{\rightsquigarrow}} BagNat$$

The way we have arrived at this implementation step is misleading. Ordinarily, we would not proceed by first developing a constructor implementation, then upgrading it to an abstractor implementation, and then simplifying the result. Our task from the beginning would be to implement $SetNat$ up to $\mathcal{A}_{Set}$ and we would not have to use an explicit **quotient** at all. On the other hand, the proof that this is an implementation might well involve a quotient construction in order to show that for every $BAG \in Mod[BagNat]$, $\mathcal{D}_{Bag\to Set}(\mathcal{E}_{Bag\to Set}(BAG))$ is $\mathcal{A}_{Set}$-equivalent to a model of $SetNat$.

We want to proceed further with the development by exploring the possibilities for an abstractor implementation of $BagNat$ which can be composed with the above. To do this, we need to determine the appropriate abstraction equivalence for $BagNat$ in this context by pushing $\equiv_{\mathcal{A}_{Set}}$ through $\mathcal{E}_{Bag\to Set};\mathcal{D}_{Bag\to Set}$.

Recall that $\equiv_{\mathcal{A}_{Set}}$ is observational equivalence wrt the set W_{Set} of all ground $Sig[SetNat]$-terms of sorts nat and $bool$. By lemma 6.10, $\mathcal{D}^{-1}_{Bag\to Set}(\equiv_{\mathcal{A}_{Set}})$ is observational equivalence on $\mathbf{PAlg}(\Sigma BagNat')$ with respect to the same set of terms (the signature morphism used in this step is the inclusion on operation names). Notice that this set is strictly included in the set of all ground $\Sigma BagNat'$-terms of sorts nat and $bool$; for example, it does not contain terms like $count(0, \emptyset)$, $count(0, add(0, \emptyset))$, $count(succ(0), \emptyset)$, $isempty(add(count(0, \emptyset), \emptyset))$, etc.

The next step is more interesting. To use lemma 6.19 we need (intuitively) to find a way of replacing each observable term from W_{Set} by a (provably) equal $Sig[BagNat]$-term. In fact, this will be sufficient since $\mathcal{E}_{Bag\to Set}$ is persistent (on models of $BagNat$). There is no trouble with observable $Sig[Nat]$-terms – these will remain unchanged. The same holds for terms of the form $isempty(B)$ where B is (syntactically!) a ground $Sig[SetNat]$-term of sort set, i.e. a ground $Sig[BagNat]$-term of sort bag not containing an occurrence of $count$. The only other terms in W_{Set} are of the form $n \in B$ where n is a ground $Sig[Nat]$-term and B is a ground term of sort bag (to which set was renamed in the previous step). By the construction of $\mathcal{E}_{Bag\to Set}$, we have that for any $BAG \in Mod[BagNat]$, $\mathcal{E}_{Bag\to Set}(BAG) \models n \in B = count(n, B) > 0$.

Thus the appropriate set W_{Bag} of observable terms contains all ground $Sig[Nat]$-terms and all $Sig[BagNat]$-terms of the forms $isempty(B)$ and $count(n, B) > 0$ where n is a ground $Sig[Nat]$-term and B is a ground $Sig[SetNat]$-term of sort set.

Now, by lemma 6.19, $\mathcal{E}^{-1}_{Bag\to Set}(\equiv_{W_{Set}}) = \equiv_{W_{Bag}}$ (since W_{Bag} is a minimal set of ground $Sig[BagNat]$-terms such that $\mathcal{E}_{Bag\to Set}$ is observably sufficiently complete wrt W_{Bag}, W_{Set} and $\mathcal{E}_{Bag\to Set}$ is persistent).

Now our job is to implement $BagNat$ in the context of the abstraction equivalence $\equiv_{W_{Bag}}$. Let $\mathcal{A}_{Bag}$ be the abstractor determined by this equivalence. The constructor implementation developed in section 5 yields the abstractor implementation:

$$BagNat \underset{\mathcal{E}_{List\to Bag};\mathcal{D}_{List\to Bag};\mathcal{R}_{List\to Bag}}{\overset{A_{Bag}}{\rightsquigarrow}} ListNat$$

The assumptions of corollary 6.12 are satisfied and so we may eliminate the **restrict** step:

$$BagNat \underset{\mathcal{E}_{List\to Bag};\mathcal{D}_{List\to Bag}}{\overset{A_{Bag}}{\rightsquigarrow}} ListNat$$

Then similarly as above we can push the abstraction equivalence through $\mathcal{D}_{List\to Bag}$ and then $\mathcal{E}_{List\to Bag}$ to obtain a relevant abstraction equivalence for $ListNat$. In this case, a suitable equivalence turns out to be $\equiv_{W_{List}}$ where W_{List} contains:

- all ground $Sig[Nat]$-terms,
- all terms $null(L)$ where L is a ground $Sig[ListNat]$-term of sort $list$ such that $Mod[ListNat] \models D(L)$, and
- all ground $Sig[ListNat]$-terms of the form $hd(L) > 0$ where L is a ground $Sig[ListNat]$-term of sort $list$ such that $Mod[ListNat] \models D(hd(L))$.

Notice that although for any ground $Sig[ListNat]$-term L of sort $list$, if $Mod[ListNat] \models D(L)$ then $Mod[ListNat] \models L = cons(n_1, \ldots, cons(n_k, nil)\ldots)$ for some ground $Sig[Nat]$-terms $n_1, \ldots, n_k$, this need not be the case for algebras which are observably equivalent to models of $ListNat$ wrt W_{List}. What is true in any such algebra A is that for any such term L, either $A \models null(L) = true$ or $A \models null(L) = false$ and in the latter case $Mod[ListNat] \models D(tl(L))$ and $Mod[ListNat] \models D(hd(L))$. This is already sufficient to reduce terms like $put(n_1, \ldots, put(n_k, nil)\ldots)$ to ground $Sig[ListNat]$-terms of sort $list$ defined in every model of $ListNat$. As a consequence of this, $null(put(n_1, \ldots, put(n_k, nil)\ldots))$ reduces to $null(L)$ and $nth(n, put(n_1, \ldots, put(n_k, nil)\ldots)) > 0$ reduces to $hd(L') > 0$ for ground $Sig[ListNat]$-terms L and L', which is our goal.

It is easy to see that $\mathcal{E}_{List\to Bag}$ is persistent on algebras observably equivalent to a model of $ListNat$ wrt W_{List}. Notice however that if we had replaced axioms like $null(L) = false \Rightarrow nth(0, L) = hd(L)$ by $nth(0, cons(n, L)) = n$ then we would lose persistency since in algebras which are observably equivalent to models of $ListNat$ wrt W_{List} but are not themselves models of $ListNat$, equations like $cons(n, L) = cons(n', L)$ may hold even if $n \neq n'$.

By lemma 6.19, $(\mathcal{E}_{List\to Bag};\mathcal{D}_{List\to Bag})^{-1}(\equiv_{W_{Bag}}) \supseteq \equiv_{W_{List}}$. Thus we are interested in implementing $ListNat$ in the context of the abstraction equivalence $\equiv_{W_{List}}$. Let us stress that this is a different task from just implementing $ListNat$. By pushing the original abstraction equivalence through the constructors used in the implementation of $SetNat$ by $ListNat$ we have determined a degree of freedom in implementing $ListNat$. In fact, it is just because of this that we can implement $ListNat$ using simply lists of booleans. This is by no means a universally useful implementation of $ListNat$ — it is tailor-made to work in this particular context.

$$
\begin{array}{l}
ListBool =_{def} \textbf{restrict} \; (\; \textbf{enrich } Bool \textbf{ by} \\
\qquad \begin{array}{ll}
\textbf{sorts} & list \\
\textbf{opns} & nil \colon \to list \\
 & cons \colon bool, list \to list \\
 & null \colon list \to bool \\
 & hd \colon list \to bool \\
 & tl \colon list \to list \\
\textbf{axioms} & D(nil) \\
 & D(cons(a, L)) \\
 & null(nil) = true \\
 & null(cons(a, L)) = false \\
 & hd(cons(a, L)) = a \\
 & tl(cons(a, L)) = L \;)
\end{array} \\
\quad \textbf{on } \{list\}
\end{array}
$$

Now the abstractor implementation of $ListNat$ by $ListBool + Nat$ proceeds in the following two steps:

Extend: $\mathcal{E}_{ListBool \to ListNat}$: $\mathbf{Spec}(Sig[ListBool + Nat]) \to \mathbf{Spec}(\Sigma ListBool') =_{def}$

$$
\begin{array}{l}
\lambda X \colon Sig[ListBool + Nat]. \\
\quad \textbf{enrich } X \textbf{ by} \\
\qquad \begin{array}{ll}
\textbf{data opns} & cons' \colon nat, list \to list \\
 & hd' \colon list \to nat \\
\textbf{axioms} & cons'(0, L) = cons(false, L) \\
 & cons'(succ(n), L) = cons(true, L) \\
 & hd(L) = false \Rightarrow hd'(L) = 0 \\
 & hd(L) = true \Rightarrow hd'(L) = succ(0)
\end{array}
\end{array}
$$

Derive: $\mathcal{D}_{ListBool \to ListNat}$: $\mathbf{Spec}(\Sigma ListBool') \to \mathbf{Spec}(Sig[ListNat]) =_{def}$

$$\lambda X \colon \Sigma ListBool'. \; \textbf{derive from } X \textbf{ by } \sigma$$

where σ renames the sorts and operations in $Sig[ListNat]$ to those in $\Sigma ListBool'$ by renaming $cons$ to $cons'$ and hd to hd' and leaving the other names unchanged. Note that this hides the original hd and $cons$ operations on lists of booleans.

The idea behind this implementation is that since we do not want to observe the values of elements of lists but only test whether or not they are greater than 0, we can replace all the non-zero values by $true$ and 0 by $false$.

Let $\mathcal{A}_{List}$ be the abstractor determined by the abstraction equivalence $\equiv_{W_{List}}$. We have:

$$ListNat \underset{\mathcal{E}_{ListBool \to ListNat}; \mathcal{D}_{ListBool \to ListNat}}{\overset{A_{List}}{\leadsto}} ListBool + Nat$$

Putting these three abstractor implementations together (the condition of the vertical composition theorem is satisfied because of the way we developed the implementations) we get:

$$SetNat \underset{\mathcal{E}_{ListBool \to ListNat}; \mathcal{D}_{ListBool \to ListNat} \;;\; \mathcal{E}_{List \to Bag}; \mathcal{D}_{List \to Bag} \;;\; \mathcal{E}_{Bag \to Set}; \mathcal{D}_{Bag \to Set}}{\overset{A_{Set}}{\leadsto}} ListBool + Nat$$

Of course, we could have implemented $SetNat$ by $ListBool + Nat$ in a much more direct way (without going through $BagNat$ and $ListNat$) but the point of this example was to show the details of the intermediate implementations rather than to discover a clever implementation of $SetNat$.

8 Parameterisation and implementations

In the same way as the simple notion of refinement on specifications gave rise to a notion of refinement for parameterised specifications, the definitions of constructor and abstractor implementation in sections 4 and 6 extend to notions of constructor and abstractor implementation for parameterised specifications. We begin with the former.

8.1 Parameterisation and constructor implementations

Definition 8.1 *For any parameterised specification* P: $\mathbf{Spec}(\Sigma_{par}) \to \mathbf{Spec}(\Sigma_{res})$ *and specification-building operation* ω: $\mathbf{Spec}(\Sigma_{res}) \to \mathbf{Spec}(\Sigma)$, $\omega(P)$ *is a parameterised specification defined by* $\omega(P) =_{def} \lambda X\colon \Sigma_{par}.\omega(P(X))\colon \mathbf{Spec}(\Sigma_{par}) \to \mathbf{Spec}(\Sigma)$.

Definition 8.2 (constructor implementation) *For any parameterised specifications with a common parameter signature* P: $\mathbf{Spec}(\Sigma_{par}) \to \mathbf{Spec}(\Sigma)$ *and* P': $\mathbf{Spec}(\Sigma_{par}) \to \mathbf{Spec}(\Sigma')$ *and constructor* κ: $\mathbf{Spec}(\Sigma') \to \mathbf{Spec}(\Sigma)$, P is implemented by P' via κ, *written* $P \underset{\kappa}{\rightsquigarrow\!\!\!>} P'$, *if* $P \rightsquigarrow\!\!\!> \kappa(P')$.

This subsumes the notion of implementation of parameterised specifications in [SW 82]. It resembles the one in [Lip 83], where a parameterised specification is a (strongly) persistent functor. According to [Lip 83], P is implemented by P' via a construction F (another persistent functor, obtained by composing certain specification-building operations) if there is some P'' and (persistent) natural transformations i: $P'' \dot{\to} P';F$ and s: $P'' \dot{\to} P$ such that i and s are componentwise injective and surjective respectively. In our framework, this corresponds roughly to an implementation via the composition of a persistent constructor, a **restrict** step and a **quotient** step (in that order).

Although there are several other definitions of implementation of parameterised specifications in the literature (see e.g. [EK 82], [GM 82] and [Gan 83]) it is difficult to compare them with ours because our definition extends the definition for the non-parameterised case in the usual way that a relation is extended from elements to functions (that is, pointwise). In contrast, [EK 82] defines implementation of parameterised specifications by comparing their bodies and then proves that this implies our notion of implementation. This is arguably preferable from the point of view of proving correctness of implementations (see section 10 for some brief comments on this point) but we prefer to adopt the natural definition and treat the problem of proving correctness separately.

As in the non-parameterised case, vertical composition is easy to show:

Theorem 8.3 (vertical composition) *For any parameterised specifications* P, P', P'' *with common parameter signature* Σ_{par}, *if* $P \underset{\kappa}{\rightsquigarrow\!\!\!>} P'$ *and* $P' \underset{\kappa'}{\rightsquigarrow\!\!\!>} P''$ *then* $P \underset{\kappa';\kappa}{\rightsquigarrow\!\!\!>} P''$.

Proof Pointwise, using vertical composition for the non-parameterised case. □

Similarly as in fact 4.15, we can mechanically strip off outermost constructors from parameterised specifications:

Fact 8.4 *For any parameterised specifications P and P' and constructor κ on the result signature of P, $\kappa(P) \underset{\kappa';\kappa}{\rightsquigarrow} P'$ provided that $P \underset{\kappa'}{\rightsquigarrow} P'$.*

Proof As for fact 4.15. □

Constructor implementations do not compose horizontally. In fact, the standard formulation of the horizontal composition property is not even well-formed in this case. Namely, if $P\colon \mathbf{Spec}(\Sigma_{par}) \to \mathbf{Spec}(\Sigma_{res})$ is a parameterised specification, SP is a Σ_{par} specification and $SP \underset{\kappa}{\rightsquigarrow} SP'$, then in general $Sig[SP'] \neq \Sigma_{par}$ and so $P(SP')$ is not even well-defined.

The following theorem plays the role of horizontal composition for constructor implementations:

Theorem 8.5 (horizontal composition) *Given a parameterised specification P with parameter signature Σ_{par} and a Σ_{par}-specification SP, if $P \underset{\kappa}{\rightsquigarrow} P'$ and $SP \underset{\mu}{\rightsquigarrow} SP'$ then $P(SP) \underset{\kappa}{\rightsquigarrow} P'(\mu(SP'))$.*

Proof $SP \underset{\mu}{\rightsquigarrow} SP'$ means $SP \rightsquigarrow \mu(SP')$, hence since all specification-building operations are monotonic, by an easy induction on the definition of P' we can show that $P'(SP) \rightsquigarrow P'(\mu(SP'))$. Since by definition, $P \underset{\kappa}{\rightsquigarrow} P'$ implies $P(SP) \underset{\kappa}{\rightsquigarrow} P'(SP)$, the vertical composition theorem for non-parameterised specifications implies $P(SP) \underset{\kappa}{\rightsquigarrow} P'(\mu(SP'))$. □

Although this is not horizontal composition as formulated in [GB 80], it is perfectly adequate for our purposes. It guarantees that in the case of a specification formed by applying a parameterised specification P to a Σ-specification SP, the developments of P and SP may proceed independently and the results be successfully combined. If $P \underset{\kappa_1}{\rightsquigarrow} P_1 \underset{\kappa_2}{\rightsquigarrow} \cdots \underset{\kappa_n}{\rightsquigarrow} P_n$ and $SP \underset{\mu_1}{\rightsquigarrow} SP_1 \underset{\mu_2}{\rightsquigarrow} \cdots \underset{\mu_m}{\rightsquigarrow} SP_m$ then $P(SP) \underset{\kappa_n;\cdots;\kappa_1}{\rightsquigarrow} P_n((\mu_m;\cdots;\mu_1)(SP_m))$. We aim at reducing the parameter specification to the empty specification and the parameterised specification to the identity. If $SP_m = \langle \Sigma_\emptyset, \emptyset \rangle$ and $P_n = \lambda X\colon \Sigma.X$ then the composition of constructors $\mu_m;\cdots;\mu_1;\kappa_n;\cdots;\kappa_1$ forms a program which implements $P(SP)$.

8.2 Parameterisation and abstractor implementations

Definition 8.6 (abstractor implementation) *For any parameterised specifications with a common parameter signature $P\colon \mathbf{Spec}(\Sigma_{par}) \to \mathbf{Spec}(\Sigma)$ and $P'\colon \mathbf{Spec}(\Sigma_{par}) \to \mathbf{Spec}(\Sigma')$, abstractor $\alpha\colon \mathbf{Spec}(\Sigma) \to \mathbf{Spec}(\Sigma)$ and constructor $\kappa\colon \mathbf{Spec}(\Sigma') \to \mathbf{Spec}(\Sigma)$,* P is implemented by P' wrt α via *κ, written $P \overset{\alpha}{\underset{\kappa}{\rightsquigarrow}} P'$, if $\alpha(P) \rightsquigarrow \kappa(P')$.*

Vertical composition of abstractor implementations of parameterised specifications is just the same as in the non-parameterised case:

Theorem 8.7 (vertical composition) *For any parameterised specifications P, P', P'' with common parameter signature Σ_{par}, if $P \underset{\kappa}{\overset{\alpha}{\rightsquigarrow}} P'$ and $P' \underset{\kappa'}{\overset{\alpha'}{\rightsquigarrow}} P''$ then $P \underset{\kappa';\kappa}{\overset{\alpha}{\rightsquigarrow}} P''$ provided that κ preserves the abstraction equivalences.*

Proof Pointwise, using vertical composition for the non-parameterised case. □

Applicability of this result in program development requires proving that the constructors we use preserve the appropriate abstraction equivalences. For this, lemmas 6.10-6.22 of section 6 are applicable just as in the non-parameterised case.

Unfortunately, the horizontal composition theorem for abstractor implementations does not hold in general, even in the form suggested by the horizontal composition theorem for constructor implementations. This is shown by the following counterexample:

Counterexample

Let $\Sigma =_{def}$ **sorts** p, obs
opns $a, b\colon \to p$
$c, d\colon \to obs$

Let $P =_{def} \lambda X\colon \Sigma.$**enrich** X **by** **opns** $f\colon p \to obs$
axioms $f(a) = c$

Let $SP =_{def}$ **sorts** p, obs
opns $a, b\colon \to p$
$c, d\colon \to obs$
axioms $a = b$

Let $\alpha'_{obs}\colon \mathbf{Spec}(\Sigma') \to \mathbf{Spec}(\Sigma')$ denote the abstractor determined by behavioural equivalence on Σ'-algebras with respect to the observable sort obs, where $\Sigma' =_{def} \Sigma \cup$ **opns** $f\colon p \to obs$. Let $\alpha_{obs}\colon \mathbf{Spec}(\Sigma) \to \mathbf{Spec}(\Sigma)$ denote the abstractor determined by behavioural equivalence on Σ-algebras with respect to obs. Then it is easy to see that $Mod[\alpha_{obs}(SP)] = \mathbf{PAlg}(\Sigma)$, hence $SP \underset{id_\Sigma}{\overset{\alpha_{obs}}{\rightsquigarrow}} \langle \Sigma, \emptyset \rangle$. Thus, one would expect that $P(SP) \underset{id_{\Sigma'}}{\overset{\alpha'_{obs}}{\rightsquigarrow}} P(id_\Sigma(\langle \Sigma, \emptyset \rangle))$ if a horizontal composition theorem were to hold. Unfortunately this is not the case: $Mod[\alpha'_{obs}(P(SP))] \not\supseteq Mod[P(\langle \Sigma, \emptyset \rangle)]$. To see this, notice that $Mod[P(SP)] \models (a = b) \wedge (f(a) = c)$, and hence $Mod[P(SP)] \models f(b) = c$. Thus $Mod[\alpha'_{obs}(P(SP))] \models f(b) = c$ as well. On the other hand, $Mod[P(\langle \Sigma, \emptyset \rangle)] \not\models f(b) = c$. □

This example shows that for horizontal composition to hold, parameter specifications cannot in general be abstracted from since parameterised specifications can make essential use of non-observable parts of the parameter. In the above example, the fact that $Mod[SP] \models a = b$ but $Mod[\alpha_{obs}(SP)] \not\models a = b$ allowed (intuitively) P to distinguish between the abstract and non-abstract form of SP.

One way to circumvent this is to restrict attention to parameterised specifications which use their arguments in an abstract way, so that if we change the argument to an equivalent one we get a result which is equivalent. Formally:

Definition 8.8 *Let $\alpha\colon \mathbf{Spec}(\Sigma) \to \mathbf{Spec}(\Sigma)$ be an abstractor. We say that two Σ-specifications $SP1$ and $SP2$ are α-*equivalent *if $Mod[\alpha(SP1)] = Mod[\alpha(SP2)]$.*

Theorem 8.9 (horizontal composition) *If* $P \overset{\alpha}{\underset{\kappa}{\rightsquigarrow}} P'$ *and* $SP \overset{\alpha'}{\underset{\kappa'}{\rightsquigarrow}} SP'$ *then* $P(SP) \overset{\alpha}{\underset{\kappa}{\rightsquigarrow}} P'(\kappa'(SP'))$ *provided that either* P *or* P' preserves α'-equivalence, *i.e. for any* α'*-equivalent specifications* $SP1, SP2$ *over the (common) parameter signature of* P *and* P'*, either* $P(SP1)$ *and* $P(SP2)$ *are* α*-equivalent or* $P'(SP1)$ *and* $P'(SP2)$ *are* $\kappa^{-1}(\equiv_\alpha)$*-equivalent.*

Proof Since $SP \overset{\alpha'}{\underset{\kappa'}{\rightsquigarrow}} SP'$, by monotonicity of $\kappa(P')$, $\kappa(P'(\alpha'(SP))) \rightsquigarrow \kappa(P'(\kappa'(SP')))$.

If P preserves α'-equivalence then $P(\alpha'(SP))$ and $P(SP)$ are α-equivalent which implies that $\alpha(P(SP)) \rightsquigarrow \alpha(P(\alpha'(SP)))$, and since $P \overset{\alpha}{\underset{\kappa}{\rightsquigarrow}} P'$ entails $\alpha(P(\alpha'(SP))) \rightsquigarrow \kappa(P'(\alpha'(SP)))$, we can indeed conclude $\alpha(P(SP)) \rightsquigarrow \kappa(P'(\kappa'(SP')))$.

If P' preserves α'-equivalence then so does $\kappa(P')$, i.e. for any two α'-equivalent specifications $SP1, SP2$ over the parameter signature, $\kappa(P'(SP1))$ and $\kappa(P'(SP2))$ are α-equivalent (by the definition of $\kappa^{-1}(\equiv_\alpha)$). In particular, we have $\alpha(\kappa(P'(SP))) \rightsquigarrow \alpha(\kappa(P'(\alpha'(SP))))$ and so trivially $\alpha(\kappa(P'(SP))) \rightsquigarrow \kappa(P'(\alpha'(SP)))$. Moreover, since $P \overset{\alpha}{\underset{\kappa}{\rightsquigarrow}} P'$ entails $\alpha(P(SP)) \rightsquigarrow \kappa(P'(SP))$, which trivially implies $\alpha(P(SP)) \rightsquigarrow \alpha(\kappa(P'(SP)))$, we can conclude that $\alpha(P(SP)) \rightsquigarrow \kappa(P'(\kappa'(SP')))$. $\square$

The requirement that P preserves α'-equivalence in the above theorem is guaranteed in either of the following three cases:

1. P is given in the form $\lambda X\colon \Sigma.SP1[\alpha'(X)]$, i.e. P explicitly abstracts from its argument before using it.

2. P is built entirely from constructors which preserve the relevant abstraction equivalences.

3. The abstractor α' is trivial, i.e. for any specification SP, $Mod[\alpha'(SP)] = Mod[SP]$.

The last case amounts to the following:

Corollary 8.10 *If* $P \overset{\alpha}{\underset{\kappa}{\rightsquigarrow}} P'$ *and* $SP \underset{\kappa'}{\rightsquigarrow} SP'$ *then* $P(SP) \overset{\alpha}{\underset{\kappa}{\rightsquigarrow}} P'(\kappa'(SP'))$. $\square$

Recall that a constructor implementation $SP \underset{\kappa'}{\rightsquigarrow} SP'$ is an abstractor implementation $SP \overset{\alpha'}{\underset{\kappa'}{\rightsquigarrow}} SP'$ where the abstractor α' is trivial. Notice however that when we push the corresponding equivalence (which is the identity) through κ' and the constructors used in the further implementation of SP' (see theorem 6.8 and subsequent discussion), the resulting abstraction equivalences may determine non-trivial abstractors again and so the use of techniques of abstractor implementations may be essential further on.

The other sufficient condition for the horizontal composability of abstractor implementations in theorem 8.9, namely the requirement that the implementing specification P' preserves α'-equivalence, seems more important and useful from the point of view of program development methodology. (Thanks to Oliver Schoett for making this point.) Intuitively, P' is to be more specific and "smaller" than P and so it may be easier to formulate it in such a way that it preserves the abstraction equivalence. In particular, suppose that we have managed to implement P entirely constructively (which is, after all, the goal of the development process), i.e. we have $P \overset{\alpha}{\underset{\kappa_n;\cdots;\kappa_1}{\rightsquigarrow}} \lambda X\colon \Sigma.X$ (where

Σ is the parameter signature of P). Then the requirement that $\lambda X\colon \Sigma.X$ preserves α'-equivalence, as formulated in theorem 8.9, reduces to the requirement that the composite constructor $\kappa_n; \cdots ;\kappa_1$ preserves the corresponding abstraction equivalence on Σ-algebras. This is a reasonable requirement to impose, as the programming language used to encode constructors should guarantee this anyway (see [Sch 86] for full discussion).

The above horizontal composition theorem may be used in modular program development just as presented for constructor implementations of parameterised specifications. We have to ensure however that the constructors used in the implementation preserve the relevant equivalences.

9 Institutions and implementations

In the previous sections we have chosen to present the development of our implementation notions, theorems and methodology in the framework of partial first-order logic with equality. This was mostly in order to take advantage of the reader's intuition; we made use of very few properties of partial algebras or the form of sentences. This means that in place of full partial first-order logic with equality we could have used partial equational logic or even some higher-order logic. Moreover, instead of partial algebras we could have used for example total algebras [GTW 76] or continuous algebras [GTWW 77], [TW 86]. We could even change the notions of signature and of algebra to deal with errors [GDLE 84], coercions [GJM 85], [Gog 83], or Milner-style polymorphism [Mil 78].

The notion of an *institution* [GB 84a] provides a tool for dealing with any of these different notions of a logical system for writing specifications. An institution comprises definitions of signature, model (algebra), sentence and a satisfaction relation satisfying a few minimal consistency conditions. (For a similar but more logic-oriented approach see [Bar 74].) By basing our definitions (of specification, implementation, etc.) on an arbitrary institution we can avoid choosing particular definitions of these underlying notions and do everything at an adequately general level. We have presented our approach to specifications in an arbitrary institution at an intuitive level in [ST 85a] and with full technical details in [ST 86a].

Definition 9.1 *An* institution **INS** *consists of:*

- *a category* $\mathbf{Sign_{INS}}$ *(of signatures);*
- *a functor* $\mathbf{Sen_{INS}}\colon \mathbf{Sign_{INS}} \to \mathbf{Set}$ *(where* $\mathbf{Set}$ *is the category of all sets;* $\mathbf{Sen_{INS}}$ *gives for any signature* Σ *the set* $\mathbf{Sen_{INS}}(\Sigma)$ *of* Σ*-sentences and for any signature morphism* $\sigma\colon \Sigma \to \Sigma'$ *the function* $\mathbf{Sen_{INS}}(\sigma)\colon \mathbf{Sen_{INS}}(\Sigma) \to \mathbf{Sen_{INS}}(\Sigma')$ *translating* Σ*-sentences to* Σ'*-sentences);*
- *a functor* $\mathbf{Mod_{INS}}\colon \mathbf{Sign_{INS}} \to \mathbf{Cat}^{op}$ *(where* $\mathbf{Cat}$ *is the category of all categories;* $\mathbf{Mod_{INS}}$ *gives for any signature* Σ *the category* $\mathbf{Mod_{INS}}(\Sigma)$ *of* Σ*-models and for any signature morphism* $\sigma\colon \Sigma \to \Sigma'$ *the* σ*-reduct functor* $\mathbf{Mod_{INS}}(\sigma)\colon \mathbf{Mod_{INS}}(\Sigma') \to \mathbf{Mod_{INS}}(\Sigma)$ *translating* Σ'*-models to* Σ*-models); and*

- *a satisfaction relation* $\models_{\mathbf{INS},\Sigma} \subseteq |\mathbf{Mod_{INS}}(\Sigma)| \times \mathbf{Sen_{INS}}(\Sigma)$ *for each signature* Σ.

such that for any signature morphism $\sigma\colon \Sigma \to \Sigma'$ *the translations* $\mathbf{Mod_{INS}}(\sigma)$ *of models and* $\mathbf{Sen_{INS}}(\sigma)$ *of sentences preserve the satisfaction relation, i.e. for any* $\varphi \in \mathbf{Sen_{INS}}(\Sigma)$ *and* $M' \in |\mathbf{Mod_{INS}}(\Sigma')|$,

$$M' \models_{\mathbf{INS},\Sigma'} \mathbf{Sen_{INS}}(\sigma)(\varphi) \quad \Longleftrightarrow \quad \mathbf{Mod_{INS}}(\sigma)(M') \models_{\mathbf{INS},\Sigma} \varphi \qquad \textit{(Satisfaction condition)}$$

To be useful as the underlying institution of a specification methodology, an institution must provide some tools for "putting things together". Thus, we additionally require that the category **Sign** has pushouts and initial objects (i.e. is finitely cocomplete) and moreover that **Mod** preserves pushouts and initial objects (and hence finite colimits), i.e. that **Mod** translates pushouts and initial objects in **Sign** to pullbacks and terminal objects (respectively) in **Cat**. For a brief discussion of these requirements see [ST 86a]. For notational convenience we omit subscripts like **INS** and Σ whenever possible, and for any signature morphism $\sigma\colon \Sigma \to \Sigma'$ we denote $\mathbf{Sen}(\sigma)$ simply by σ and $\mathbf{Mod}(\sigma)$ by $\_|_\sigma$.

All of the logical systems mentioned above fit into the mould of an institution. In particular, partial first-order logic with equality forms an institution **PFOEQ** as follows:

- $\mathbf{Sign_{PFOEQ}}$ is **Sign**
- For a signature Σ, $\mathbf{Sen_{PFOEQ}}(\Sigma)$ is the set of partial first-order Σ-sentences; for a signature morphism $\sigma\colon \Sigma \to \Sigma'$, $\mathbf{Sen_{PFOEQ}}(\sigma)$ is the translation of Σ-sentences to Σ'-sentences, defined in the obvious way.
- For a signature Σ, $\mathbf{Mod_{PFOEQ}}(\Sigma)$ is the category $\mathbf{PAlg}(\Sigma)$; for a signature morphism $\sigma\colon \Sigma \to \Sigma'$, $\mathbf{Mod_{PFOEQ}}(\sigma)$ is the σ-reduct functor $\_|_\sigma\colon \mathbf{PAlg}(\Sigma') \to \mathbf{PAlg}(\Sigma)$.
- For a signature Σ, $\models_{\mathbf{PFOEQ},\Sigma}$ is the satisfaction relation as defined in section 2.

The satisfaction condition is just the satisfaction lemma of section 2. Moreover, $\mathbf{Sign_{PFOEQ}}$ is finitely cocomplete (as mentioned in section 2) and $\mathbf{Mod_{PFOEQ}}$ translates finite colimits in $\mathbf{Sign_{PFOEQ}}$ to limits in **Cat**.

It would now be appropriate to repeat the preceeding sections in the context of an arbitrary institution, generalising from **PFOEQ**. Of course, we are not going to bore the reader with this — we will just give a brief summary of how this can be done and where some problems lie.

The contents of section 3 generalises immediately to an arbitrary institution. The definitions and results there were introduced in [ST 86a] in the framework of an arbitrary institution **INS** in exactly the form they appear here (replacing **PAlg** by $\mathbf{Mod_{INS}}$, etc.). The examples of specification-building operations (**translate**, $\cup$) are defined exactly the same way there.

The general concept of a constructor (section 4) may be formulated in an arbitrary institution as well, again as a specification-building operation determined by a function on models in this institution. This yields immediately the concept of a constructor implementation for specifications in an arbitrary institution. Moreover, the vertical composition theorems (theorems 4.14 and 8.3) and the horizontal composition theorem (theorem 8.5) hold without modification. We can directly generalise to an arbitrary institution the definitions of the constructors **derive** and **extend** (the latter requires the free functors involved to exist, i.e. the institution to be liberal [GB 84a], though) but our definitions of **restrict** and **quotient** use "non-institutional" properties of the partial algebra framework. Fortunately, the definition of a reachable subalgebra may be presented in an institution-independent way using standard notions of category theory (see [ST 86a], [Tar 85]). This definition may be used to define the **restrict** constructor in an arbitrary institution. It is not yet clear to us how the **quotient** constructor we have presented here can be generalised to work in an arbitrary institution; it seems that some of the ideas presented in [Tar 85] may lead to a satisfactory solution of this problem. Next, it is easy to see that the definitions of the amalgamated union and the translation of constructors are directly applicable and work as expected in an arbitrary institution. In particular, theorems 4.17 and 4.18 hold in this more general framework.

The notion of an abstractor (section 6) generalises directly to the framework of an arbitrary institution, where it is determined by an equivalence on the category of models over a given signature. This immediately yields the notion of abstractor implementation in an arbitrary institution. Moreover, the vertical composition theorems (theorems 6.8 and 8.7) and the horizontal composition theorem (theorem 8.9) and its corollary hold under exactly the same assumptions as before. As discussed in detail in [ST 87a], the most straightforward generalisation to an arbitrary institution of the notion of observational equivalence with respect to a set of terms (and hence of behavioural equivalence with respect to a set of observable sorts) is the concept of observational equivalence with respect to a set of sentences in this institution, already mentioned briefly in section 6.

Definition 9.2 *For any signature $\Sigma \in |\mathbf{Sign_{INS}}|$, set $\Phi \subseteq \mathbf{Sen_{INS}}(\Sigma)$ of Σ-sentences and Σ-models $A, B \in |\mathbf{Mod_{INS}}(\Sigma)|$, A and B are* observably equivalent wrt Φ, *written $A \equiv_\Phi B$, if for all $\varphi \in \Phi$, $A \models_{\mathbf{INS},\Sigma} \varphi$ iff $B \models_{\mathbf{INS},\Sigma} \varphi$.*

In **PFOEQ**, for any signature Σ and set W of ground Σ-terms, the former observational equivalence $\equiv_W$ is the same as observational equivalence with respect to the set of sentences consisting of the sentence $D(t)$ for each term t in W and the sentence $t = t'$ for each pair t, t' of terms in W of the same sort. Notice however that the set of observable sentences needed to express behavioural equivalence varies from one institution to another; in fact, even the basic idea of an observable sort cannot be expressed directly in an institution-independent way (recall that signatures are arbitrary objects which do not have to include sorts or operations). Again, the lemmas on how specific constructors preserve observational equivalence must be examined one at a time. With appropriate reformulation, lemmas 6.10 (for **derive**), 6.19 (for synthesizing constructors), 6.20 (for amalgamated union) and 6.22 (for translation of constructors) and all their corollaries still hold in an arbitrary institution. Just as an example, let us restate and prove lemma 6.19 in this framework:

Lemma 6.19′ (synthesize) *For any signature morphism $\sigma: \Sigma \to \Sigma'$, constructor κ: $\mathbf{Spec}(\Sigma) \to \mathbf{Spec}(\Sigma')$ and sets $\Phi \subseteq \mathbf{Sen}(\Sigma)$ and $\Phi' \subseteq \mathbf{Sen}(\Sigma')$ of Σ- and Σ'-sentences respectively, if:*

- κ *is* observably sufficiently complete *(*wrt Φ, Φ'*), i.e. for any $\varphi' \in \Phi'$ there exists $\varphi \in \Phi$ such that for all $M \in |\mathbf{Mod}(\Sigma)|$, $\kappa(M) \models_{\Sigma'} \varphi'$ iff $\kappa(M) \models_{\Sigma'} \sigma(\varphi)$; and*

- κ *is* observably persistent *(*wrt Φ*), i.e. for any $\varphi \in \Phi$ and $M \in |\mathbf{Mod}(\Sigma)|$, $M \models_\Sigma \varphi$ iff $\kappa(M) \models_{\Sigma'} \sigma(\varphi)$*

then $\kappa^{-1}(\equiv_{\Phi'}) \supseteq \equiv_\Phi$. Moreover, if in addition Φ is a minimal set of sentences such that observable sufficient completeness holds then $\kappa^{-1}(\equiv_{\Phi'}) = \equiv_\Phi$.

Proof Let $M1, M2 \in |\mathbf{Mod}(\Sigma)|$. Assume $M1 \equiv_\Phi M2$; we prove that $\kappa(M1) \equiv_{\Phi'} \kappa(M2)$. Let $\varphi' \in \Phi'$; consider $\varphi \in \Phi$ such that for every $M \in |\mathbf{Mod}(\Sigma)|$, $\kappa(M) \models_{\Sigma'} \varphi'$ iff $\kappa(M) \models_{\Sigma'} \sigma(\varphi)$. Then $\kappa(M1) \models_{\Sigma'} \varphi'$ iff $\kappa(M1) \models_{\Sigma'} \sigma(\varphi)$ iff $M1 \models_\Sigma \varphi$ iff $M2 \models_\Sigma \varphi$ iff $\kappa(M2) \models_{\Sigma'} \sigma(\varphi)$ iff $\kappa(M2) \models_{\Sigma'} \varphi'$. Moreover, if Φ is minimal then for any $\varphi \in \Phi$ there exists $\varphi' \in \Phi'$ such that for all $M \in |\mathbf{Mod}(\Sigma)|$, $\kappa(M) \models_{\Sigma'} \varphi'$ iff $\kappa(M) \models_{\Sigma'} \sigma(\varphi)$ and so $M1 \equiv_\Phi M2$ provided that $\kappa(M1) \equiv_{\Phi'} \kappa(M2)$ by the same chain of equivalences as above. □

Notice that in this context the condition of observable sufficient completeness with respect to Φ, Φ' may be relaxed slightly. Namely, for any $\varphi' \in \Phi'$ it is also sufficient to find $\varphi \in \Phi$ such that for all $M \in |\mathbf{Mod}(\Sigma)|$, $\kappa(M) \models_{\Sigma'} \varphi'$ iff $\kappa(M) \not\models_{\Sigma'} \sigma(\varphi)$. Referring to our example in section 7, this would allow us to replace observations of the form $count(n, B) > 0 = true$ by $count(n, B) = 0$.

Since we did not define the **quotient** constructor at all, lemma 6.15 cannot be considered. As for lemma 6.11 (for **restrict**), it does not hold in general since there are sentences which are not preserved under submodels (e.g. existential sentences in first-order logic). If however we restrict the form of observable sentences appropriately (e.g. to infinitary conditional equations as defined in the framework of so-called *abstract algebraic institutions* in [Tar 86a]) so that they are preserved under submodels, the lemma and its corollaries hold.

Summing this up, the notions, results and methodology presented in the previous sections (with the single exception of the **quotient** constructor) carry over to the framework of an arbitrary institution.

This generalisation is important not only because it allows us to develop programs from specifications in different institutions. Even in the process of developing a single program it may be convenient to use different institutions at different stages of development. After all, we proceed from a high-level user-oriented specification to a low-level computer-oriented program. It seems natural that different logical tools are necessary to express properties at these very different levels. Thus we need a means of switching from one institution to another during the development process. This problem was mentioned in [Tar 86b]. The following notion seems adequate for this purpose:

Definition 9.3 *For any two institutions* **INS1** *and* **INS2***, a* semi-institution morphism from **INS1** to **INS2***, γ:* **INS1** $\to$ **INS2***, consists of:*

- *a functor γ_{Sign}: $\mathbf{Sign_{INS1}} \to \mathbf{Sign_{INS2}}$, and*

- *a natural transformation γ: $\mathbf{Mod_{INS1}} \dot\to \gamma_{Sign};\mathbf{Mod_{INS2}}$, i.e. a natural family of functors γ_Σ: $\mathbf{Mod_{INS1}}(\Sigma) \to \mathbf{Mod_{INS2}}(\gamma_{Sign}(\Sigma))$ for $\Sigma \in |\mathbf{Sign_{INS1}}|$.*

Intuitively, γ: **INS1** $\rightarrow INS2$ translates signatures and models of **INS1** to signatures and models of **INS2**. Notice that this is not quite an *institution morphism* as defined in [GB 84a]; an institution morphism from **INS1** to **INS2** would additionally translate sentences in **INS2** to sentences in **INS1** (preserving the satisfaction relation). This is not necessary for our purposes and moreover in many cases we want to deal with it is unachievable. For example, if we want to specify programs using the institution **PFOEQ** and then implement them in an institution **PEQ** of partial equational logic — which may be viewed as an applicative programming language — then we could use the trivial semi-institution morphism γ: **PEQ** $\rightarrow$ **PFOEQ** (which is identity on signatures and models) which cannot be extended to an institution morphism: there would be no way to translate existential quantifiers into equations, for example.

Any semi-institution morphism γ: **INS1** $\rightarrow$ **INS2** determines a constructor which maps specifications in **INS1** to specifications in **INS2**.

Definition 9.4 (change institution) *For any Σ-specification SP (over* **INS1***) and semi-institution morphism γ:* **INS1** $\rightarrow$ **INS2***, the semantics of the specification* **change institution of** *SP* **via** γ *(over* **INS2***) is as follows:*

$$Sig[\textbf{change institution of } SP \textbf{ via } \gamma] = \gamma_{Sign}(\Sigma)$$
$$Mod[\textbf{change institution of } SP \textbf{ via } \gamma] = \gamma_\Sigma(Mod[SP])$$

The **change institution** *specification-building operations (one for each γ:* **INS1** $\rightarrow$ **INS2** *and $\Sigma \in |\mathbf{Sign_{INS1}}|$) are constructors determined by the functors γ_Σ: $\mathbf{Mod_{INS1}}(\Sigma) \rightarrow \mathbf{Mod_{INS2}}(\gamma_{Sign}(\Sigma))$.*

With this definition, we can use **change institution** just as any other constructor in constructor implementations. For example, we can implement specifications in **INS2** by specifications in **INS1**. All of the composition theorems continue to hold in the presence of this constructor.

In order to use **change institution** in abstractor implementations, we need a way of pushing abstraction equivalences through it.

Lemma 9.5 (change institution) *For any semi-institution morphism γ:* **INS1** $\rightarrow$ **INS2***, signature $\Sigma \in |\mathbf{Sign_{INS1}}|$ and sets $\Phi 1 \subseteq \mathbf{Sen_{INS1}}(\Sigma)$ and $\Phi 2 \subseteq \mathbf{Sen_{INS2}}(\gamma_{Sign}(\Sigma))$,*

if for any $\varphi 2 \in \Phi 2$ there exists $\varphi 1 \in \Phi 1$ such that

for all $M1 \in |\mathbf{Mod_{INS1}}(\Sigma)|$, $\gamma_\Sigma(M1) \models_{\mathbf{INS2},\gamma_{Sign}(\Sigma)} \varphi 2$ iff $M1 \models_{\mathbf{INS1},\Sigma} \varphi 1$,

then $\gamma_\Sigma^{-1}(\equiv_{\Phi 2}) \supseteq \equiv_{\Phi 1}$.

Moreover, if in addition $\Phi 1$ is a minimal set of sentences which satisfy the assumptions then $\gamma_\Sigma^{-1}(\equiv_{\Phi 2}) = \equiv_{\Phi 1}$.

Proof Let $M1, M1' \in \mathbf{Mod_{INS1}}(\Sigma)$. Assume $M1 \equiv_{\Phi 1} M1'$; we prove $\gamma_\Sigma(M1) \equiv_{\Phi 2} \gamma_\Sigma(M1')$. Let $\varphi 2 \in \Phi 2$, and consider $\varphi 1 \in \Phi 1$ such that for every $M \in \mathbf{Mod_{INS1}}(\Sigma)$, $\gamma_\Sigma(M) \models_{\mathbf{INS2},\gamma_{Sign}(\Sigma)} \varphi 2$ iff $M \models_{\mathbf{INS1},\Sigma} \varphi 1$. Then $\gamma_\Sigma(M1) \models_{\mathbf{INS2},\gamma_{Sign}(\Sigma)} \varphi 2$ iff $M1 \models_{\mathbf{INS1},\Sigma} \varphi 1$ iff $M1' \models_{\mathbf{INS1},\Sigma} \varphi 1$ iff $\gamma_\Sigma(M1') \models_{\mathbf{INS2},\gamma_{Sign}(\Sigma)} \varphi 2$. Moreover, if $\Phi 1$ is minimal then for any $\varphi 1 \in \Phi 1$ there exists $\varphi 2 \in \Phi 2$

such that for every $M \in \mathbf{Mod_{INS1}}(\Sigma)$, $\gamma_\Sigma(M) \models_{\mathbf{INS2},\gamma_{Sign}(\Sigma)} \varphi 2$ iff $M \models_{\mathbf{INS1},\Sigma} \varphi 1$ and so $M1 \equiv_{\Phi 1} M1'$ provided $\gamma_\Sigma(M1) \equiv_{\Phi 2} \gamma_\Sigma(M1')$ by the same chain of equivalences. □

In case γ: **INS1** $\rightarrow$ **INS2** may be extended to an institution morphism, then in the above lemma $\Phi 1$ may be defined as the translation of sentences from $\Phi 2$.

To illustrate the above ideas, we now briefly outline a simple example of an abstractor implementation of the $BagNat$ specification (see section 5) by an imperative program over $ListNat$. Since we want to implement a specification in **PFOEQ** by an imperative program, this will involve a change of institution. We begin with a sketch of an institution of a simple imperative programming language. The institution **IMP** will be parameterised by an algebra DT of primitive (built-in) data types and functions of the language over a (**PFOEQ**) signature ΣDT. The components of $\mathbf{IMP}_{DT}$ are as follows:

- Signatures are sets of functional procedure names with types of the form $s_1, \ldots, s_n \rightarrow s$ where $s_1, \ldots, s_n, s$ are sorts of ΣDT. Signature morphisms are maps between these sets of names which preserve types.

- Sentences over a given signature Π are procedure definitions of the form

  $$p(x_1\colon s_1, \ldots, x_n\colon s_n) = \textit{while-program};\ \mathbf{result}\ \textit{expr}\colon s$$

  where p: $s_1, \ldots, s_n \rightarrow s$ is a procedure name in Π, $expr$ is a ΣDT-term (with variables) of sort s, and *while-program* is a statement in a deterministic programming language such as e.g. TINY [Gor 79] containing expressions of this form. With a bit of additional complication we could also allow expressions to contain procedure calls.

- A model M over a signature Π assigns to each procedure name p: $s_1, \ldots, s_n \rightarrow s$ in Π and every sequence $v_1, \ldots, v_n$ of DT values of sorts $s_1, \ldots, s_n$ respectively a computation $M(p)(v_1, \ldots, v_n)$ which is either:

  Divergence: an infinite sequence of states (variable valuations);

  Unsuccessful termination: a finite sequence of states; or

  Successful termination: a finite sequence of states and a value $v \in |DT|_s$.

  It is easy to see that any model M determines, for any procedure name p: $s_1, \ldots, s_n \rightarrow s$ in Π, a partial function p_M: $|DT|_{s_1} \times \cdots \times |DT|_{s_n} \rightarrow |DT|_s$.

- Given a signature Π, a Π-model M and a Π-sentence φ of the form

  $$p(x_1\colon s_1, \ldots, x_n\colon s_n) = \textit{while-program};\ \mathbf{result}\ \textit{expr}\colon s$$

  M satisfies φ if $M(p)(v_1, \ldots, v_n)$ is the computation of *while-program* starting in the state $[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$, and if the computation terminates successfully in a state in which $expr$ has a defined value then $M(p)(v_1, \ldots, v_n)$ contains this value as well. The computations of *while-program* are defined by an operational semantics.

For any primitive data type DT, there is a semi-institution morphism γ: $\mathbf{IMP}_{DT} \to \mathbf{PFOEQ}$ which maps any $\mathbf{IMP}_{DT}$-signature Π to the **PFOEQ** signature $\Sigma DT \cup \Pi$ and any Π-model M to the $\gamma_{Sign}(\Pi)$-algebra which is the expansion of DT by the partial functions p_M associated with procedure names p in Π. (Notice that we cannot hope for translation of **PFOEQ** sentences to $\mathbf{IMP}_{DT}$-sentences here so γ cannot be extended to an institution morphism.)

Consider **IMP** built over an initial model of $ListNat$. Using the same idea as in the implementations of $BagNat$ by $ListNat$ in sections 5 and 7, we can implement $BagNat$ by the empty specification (over the empty signature $\emptyset$) in the institution $\mathbf{IMP}_{ListNat}$ in three steps:

Synthesize: $\mathcal{S}_{\emptyset\to Bag}$: $\mathbf{Spec}_{\mathbf{IMP}_{ListNat}}(\emptyset) \to \mathbf{Spec}_{\mathbf{IMP}_{ListNat}}(\Pi 1) =_{def}$

$\lambda X\colon \emptyset.$ **procedures** $nth\colon nat, list \to nat$

$put\colon nat, list \to list$

axioms $nth(n\colon nat, L\colon list) =$

```
while n>0 and not(null(L)) do
   n:=n-1; L:=tl(L) od;
if null(L) then r:=0 else r:=hd(L) fi;
```

result $r\colon nat$

$put(n\colon nat, L\colon list) =$

```
L1:=nil;
while n>0 and not(null(L)) do
   L1:=cons(hd(L),L1);
   n:=n-1; L:=tl(L) od;
if null(L) then
   v:=0;
   while n>0 do
      n:=n-1; L1:=cons(0,L1) od;
else v:=hd(L); L:=tl(L) fi;
L:=cons(succ(v),L);
while not(null(L1)) do
   L:=cons(hd(L1),L); L1:=tl(L) od;
```

result $L\colon list$

The idea behind these programs is exactly the same as in the **extend** step $\mathcal{E}_{List\to Bag}$ of sections 5 and 7.

Changing institutions: $\mathcal{C}_{\emptyset\to Bag}$: $\mathbf{Spec}_{\mathbf{IMP}_{List}}(\Pi 1) \to \mathbf{Spec}_{\mathbf{PFOEQ}}(\gamma_{Sign}(\Pi 1)) =_{def}$

$\lambda X\colon \Pi 1.$ **change institution of** X **via** γ

Derive: as before

$\mathcal{D}_{\emptyset\to Bag}$: $\mathbf{Spec}_{\mathbf{PFOEQ}}(\gamma_{Sign}(\Pi 1)) \to \mathbf{Spec}(Sig[BagNat]) =_{def}$

$\lambda X\colon \gamma_{Sign}(\Pi 1).$ **derive from** X **by** σ

where σ renames the sorts and operations in $Sig[BagNat]$ to those in $\gamma_{Sign}(\Pi 1)$ by renaming *bag* to *list*, $\emptyset$ to *nil*, *add* to *put*, *count* to *nth* and *isempty* to *null* and leaving the other names as they were. Note that *hd*, *tl* and *cons* are hidden in this step.

Now, similarly as in section 7, we have:

$$BagNat \underset{\mathcal{S}_{\emptyset\to Bag};\mathcal{C}_{\emptyset\to Bag};\mathcal{D}_{\emptyset\to Bag}}{\overset{A_{Bag}}{\leadsto\!\!\!\!\leadsto}} \langle\emptyset,\emptyset\rangle$$

and with the addition of a **restrict** step (as before) we can obtain a constructor implementation here as well.

The **change institution** constructor determined by a semi-institution morphism can be used not only in the development process to implement specifications by specifications in a different institution as above, but also (like any other specification-building operation) as a tool for building specifications. In particular, specifications built using **change institution** may be used as components of other, more complex specifications. Just as in the case of multiplex institutions [GB 84a] where sentences and theories in one institution can be included in specifications over another institution, we can move specifications between institutions connected by semi-institution morphisms. For example, we could build a specification in the institution **PFOEQ** in which some parts would be "specified" using while-programs of the institution $\mathbf{IMP}_{DT}$.

10 Concluding remarks

We have presented a view of the program development process as a sequence of refinement steps leading from a high-level specification to a program. A key concept here is that of an implementation of one specification by another. We started by recalling a simple notion of refinement from [SW 83] and used it to define two more general notions: constructor implementations and abstractor implementations, which subsume most (if not all) of the notions of implementation in the literature. We proved some basic facts about these notions, in particular vertical and (modified) horizontal composition theorems, and studied how they may be used in the practice of program development. The methodology, although presented in the framework of partial algebras with first-order axioms, was shown to generalise to an arbitrary logical system (institution). Moreover, we indicated a way of changing institutions in the course of program development which allows us to formally treat (for example) implementation of algebraic specifications by imperative programs.

A number of important problems connected with the ideas presented remain to be considered. First, we do not discuss here any methods for proving correctness of refinements; methods for proving theorems in specifications, especially in the context of observational abstraction [ST 86a,87a], are relevant to this problem. This would be especially important in the case of parameterised specifications since definition 3.3 suggests checking an infinite number of cases, one for each argument specification. Intuitively, $P \leadsto\!\!\!> P'$ should be deducible from the refinement relation between their bodies. However, since bodies of parameterised specifications in our approach may contain free variables it is not quite obvious how to define this refinement and some additional techniques are necessary.

This is in contrast with a different approach to parameterisation based on pushouts in the category of specifications as used in [BG 80] and [Ehr 82]. In this approach, a parameterised specification is a specification morphism P: $SP_{par} \hookrightarrow SP_{res}$ including the formal parameter specification into the result specification. To apply P to an actual parameter specification SP_{act}, we have to provide another specification morphism which "fits" SP_{act}-models into SP_{par}-models, σ: $SP_{par} \rightarrow SP_{act}$. The result of applying P to SP_{act} using σ, written $P(SP_{act}[\sigma])$, is defined (up to isomorphism) as the pushout object in the category of specifications of P and σ:

$$\begin{tikzcd} SP_{par} \arrow[r, "P"] \arrow[d, "\sigma"'] & SP_{res} \arrow[d, "\sigma'"] \\ SP_{act} \arrow[r, "P'"'] & P(SP_{act}[\sigma]) \end{tikzcd}$$

In this approach, given two parameterised specifications $P1$: $SP_{par} \hookrightarrow SP1_{res}$ and $P2$: $SP_{par} \hookrightarrow SP2_{res}$ (having the same parameter resp. result signatures), $P1 \rightsquigarrow P2$ iff $SP1_{res} \rightsquigarrow SP2_{res}$. This easily follows from the fact that $P(SP_{act}[\sigma])$ may be defined as

$$(\textbf{translate}\ SP_{act}\ \textbf{by}\ P') \cup (\textbf{translate}\ SP_{res}\ \textbf{by}\ \sigma')$$

which involves SP_{res} as a "constant" specification. This is in contrast to the approach to parameterisation we have used, in which the body specification "essentially" involves the parameter. For example, we can write parameterised specifications like:

$$\lambda X{:}\ \Sigma.\textbf{restrict}\ (\textbf{derive from}\ X\ \textbf{by}\ \sigma{:}\ \Sigma' \rightarrow \Sigma)\ \textbf{on}\ sorts(\Sigma')$$

which is not expressible in the pushout approach. Notice also that the fact that application can be defined using **translate** and $\cup$ as above directly implies that both vertical and horizontal composition results hold for parameterised specifications using the pushout approach (for the simple notion of refinement); for constructor implementations the details are yet to be investigated — the notion of translation of a constructor and its properties seem useful in this context.

Solutions to the problem of horizontal composition of pushout-based parameterised specifications were given in [Gan 83], [GM 82], [EK 82] among others, for the particular notions of implementation considered in these papers. These are only examples taken from the large body of technical work in the literature on different specific notions of implementation. Viewed in our approach, each of these notions corresponds to a restriction on the choice of constructors and abstractors which may be used. In this paper we have tried to unify and generalise the many different notions of implementation in the literature. This quest for generality yields a uniform framework in which we can compare different approaches. More importantly, we can investigate which of the problems encountered under different notions of implementation are inherent to the very concept of what an implementation should be and which are just technicalities caused by the imposed restrictions, and conversely, which results and properties are consequences of such restrictions and which are inherent to the nature of implementations. We have not yet tried to pursue this line of investigation in a systematic manner.

One issue we have so far omitted is the problem of inconsistent specifications. According to our definition, any inconsistent specification refines any specification over the same signature. Note, however, that any program determines a model, and so if we succeed in refining a specification to a program then the original specification must have been consistent. This means that checking consistency is not necessary to ensure correctness of the development process. However, an inconsistent specification is a blind alley (worse, it can be refined forever) and so to be cautious it is advisable to check for consistency at each stage. On the other hand, even a consistent specification may have no computable model and so we cannot in general avoid blind alleys in program development anyway.

Once we have successfully gone through the program development process starting from a specification SP, by vertically and horizontally composing all the implementation steps we arrive at an implementation of the form $SP \underset{\kappa_{j+1};\cdots;\kappa_n}{\overset{\alpha}{\rightsquigarrow}} \kappa_j(\ldots\kappa_1(\langle\Sigma_\emptyset,\emptyset\rangle)\ldots)$. Now the constructor $\kappa_1;\cdots;\kappa_n$ amounts to a program which realises SP up to the abstraction equivalence corresponding to α. In what we have presented here, $\kappa_1,\ldots,\kappa_n$ are just functions rather than actual pieces of programs in the usual sense. We did not introduce any particular syntax for defining constructors apart from the one used in examples. It would be interesting to develop a programming language which would provide facilities for defining and composing constructors (this would probably require restricting the notion of constructor we use, as implied in section 3). A good starting point seems to be Standard ML [Mil 85] with modules [MacQ 85], where constructors could be defined as Standard ML functors (i.e. parameterised modules). For example, the constructor $\lambda X\colon \Sigma.$ **derive from** X **by** $\sigma\colon \Sigma' \to \Sigma$ can be coded in Standard ML as follows:

```
functor DERIVE(X : SIG) : SIG' = X
```

provided that σ is a signature inclusion; if not then the right-hand side must be modified to include the appropriate renamings. Similarly, the **extend** constructor $\mathcal{E}_{Bag\to Set}$ (see section 5)

$\lambda X\colon Sig[BagNat]$. **enrich** X **by**
  data opns $\in\colon nat, bag \to bool$
    axioms $a \in B = count(a,B) > 0$

can be coded as:

```
functor EXTEND(X : SIGBAGNAT) : SIGBAGNAT' =
     struct
        open X;
        infix isin;
        fun a isin B = count(a,B) > 0
     end;
```

(in general, this only works if the axioms are equational and in the form required by Standard ML). We have already used Standard ML modules as a structuring mechanism in the Extended ML wide-spectrum specification/programming language [ST 85b,86b]. It would be interesting to investigate how the ideas on program development presented in this paper may be applied in Extended ML where abstractors are always used in a fixed way; especially intriguing is the relation between modules in Extended ML and their use as constructors as outlined above.

Acknowledgements

Many of the ideas presented in this paper evolved in close collaboration with Martin Wirsing. Our thanks to Oliver Schoett for many discussions on the subject of this paper, to Hartmut Ehrig for his criticism which stimulated us to write these ideas down, to an anonymous TAPSOFT referee who directed our attention to [Lip 83] and to the anonymous *Acta Informatica* referees whose remarks helped us to improve the presentation. Thanks to Teresa for (gastronomic) care. This work was supported by grants from the Alvey Directorate and the Polish Academy of Sciences.

11 References

[**AM 75**] Arbib, M.A. and Manes, E.G. *Arrow, Structures and Functors: the Categorical Imperative.* Academic Press (1975).

[**AMRW 85**] Astesiano, E., Mascari, G.F., Reggio, G. and Wirsing, M. On the parameterized algebraic specification of concurrent systems. *Proc. 10th Colloq. on Trees in Algebra and Programming*, Joint Conf. on Theory and Practice of Software Development (TAPSOFT), Berlin. Springer LNCS 185, pp. 342-358 (1985).

[**AR 83**] Astesiano, E. and Reggio, G. A unifying viewpoint for the constructive specification of cooperation, concurrency and distribution. Quaderni CNET no. 115, ETS Pisa (1983).

[**Bar 74**] Barwise, K.J. Axioms for abstract model theory. *Annals of Math. Logic 7* pp. 221-265 (1974).

[**Bau 81a**] Bauer, F.L. *et al* (the CIP Language Group) Report on a wide spectrum language for program specification and development. Report TUM-I8104, Technische Univ. München (1981). See also: *The Wide Spectrum Language CIP-L.* Springer LNCS 183 (1985).

[**Bau 81b**] Bauer, F.L. *et al* (the CIP Language Group) Programming in a wide spectrum language: a collection of examples. *Science of Computer Programming 1* pp. 73-114 (1981).

[**BM 81**] Bergstra, J.A. and Meyer, J.J. I/O computable data structures. *SIGPLAN Notices 16*, 4 pp. 27-32 (1981).

[**BBC 86**] Bernot, G., Bidoit, M. and Choppy, C. Abstract implementations and correctness proofs. *Proc. Symp. on Theoretical Aspects of Computer Science*, Saarbrücken. Springer LNCS 210, pp. 236-251 (1986).

[**BW 85**] Bloom, S.L. and Wagner, E.G. Many-sorted theories and their algebras with some applications to data types. In: *Algebraic Methods in Semantics* (M. Nivat and J.C. Reynolds, eds.), Cambridge Univ. Press, pp. 133-168 (1985).

[**BMPW 86**] Broy, M., Möller, B., Pepper, P. and Wirsing, M. Algebraic implementations preserve program correctness. *Science of Computer Programming 7*, pp. 35-53 (1986).

[**BrW 82**] Broy, M. and Wirsing, M. Partial abstract types. *Acta Informatica 18* pp. 47-64 (1982).

[**Bur 86**] Burmeister, P. *A Model Theoretic Approach to Partial Algebras.* Akademie-Verlag (1986).

[**BG 77**] Burstall, R.M. and Goguen, J.A. Putting together theories to make specifications. *Proc. 5th Intl. Joint Conf. on Artificial Intelligence*, Cambridge, pp. 1045-1058 (1977).

[**BG 80**] Burstall, R.M. and Goguen, J.A. The semantics of Clear, a specification language. *Proc. of Advanced Course on Abstract Software Specifications*, Copenhagen. Springer LNCS 86, pp. 292-332 (1980).

[**BG 82**] Burstall, R.M. and Goguen, J.A. Algebras, theories and freeness: an introduction for computer scientists. *Proc. 1981 Marktoberdorf NATO Summer School.* Reidel (1982).

[**BMS 80**] Burstall, R.M., MacQueen, D.B. and Sannella, D.T. HOPE: an experimental applicative language. *Proc. 1980 LISP Conference*, Stanford, pp. 136-143 (1980).

[**DLS 87**] Dubois, E., Levy, N. and Souquieres, J. Formalising restructuring operators in a specification process. *Proc. ESEC '87*, Strasbourg (1987).

[**deNH 84**] de Nicola, R. and Hennessy, M.C.B. Testing equivalences for processes. *Theoretical Computer Science 34*, pp. 83-133 (1984).

[**Ehr 81**] Ehrich, H.-D. On realization and implementation. *Proc. 10th Intl. Symp. on Mathematical Foundations of Computer Science*, Strbske Pleso, Czechoslovakia. Springer LNCS 118, pp. 271-280 (1981).

[**Ehr 82**] Ehrich, H.-D. On the theory of specification, implementation, and parametrization of abstract data types. *Journal of the Assoc. for Computing Machinery 29* pp. 206-227 (1982).

[**EFH 83**] Ehrig, H., Fey, W. and Hansen, H. ACT ONE: an algebraic specification language with two levels of semantics. Report Nr. 83-03, Institut für Software und Theoretische Informatik, Technische Univ. Berlin (1983).

[EK 82] Ehrig, H. and Kreowski, H.-J. Parameter passing commutes with implementation of parameterized data types. *Proc. 9th Intl. Colloq. on Automata, Languages and Programming*, Aarhus, Denmark. Springer LNCS 140, pp. 197-211 (1982).

[EKMP 82] Ehrig, H., Kreowski, H.-J., Mahr, B. and Padawitz, P. Algebraic implementation of abstract data types. *Theoretical Computer Science 20* pp. 209-263 (1982).

[EKTWW 80] Ehrig, H., Kreowski, H.-J., Thatcher, J.W., Wagner, E.G. and Wright, J.B. Parameterized data types in algebraic specification languages (short version). *Proc. 7th Intl. Colloq. on Automata, Languages and Programming*, Noordwijkerhout, Netherlands. Springer LNCS 85, pp. 157-168 (1980).

[EM 85] Ehrig, H. and Mahr, B. *Fundamentals of Algebraic Specification I: Equations and Initial Semantics*. EATCS Monographs on Theoretical Computer Science, Springer (1985).

[ETLZ 82] Ehrig, H., Thatcher, J.W., Lucas, P. and Zilles, S.N. Denotational and initial algebra semantics of the algebraic specification language LOOK. Draft report, IBM research (1982).

[EWT 83] Ehrig, H., Wagner, E.G. and Thatcher, J.W. Algebraic specifications with generating constraints. *Proc. 10th Intl. Colloq. on Automata, Languages and Programming*, Barcelona. Springer LNCS 154, pp. 188-202 (1983).

[FGJM 85] Futatsugi, K., Goguen, J.A., Jouannaud, J.-P. and Meseguer, J. Principles of OBJ2. *Proc. 12th ACM Symp. on Principles of Programming Languages*, New Orleans, pp. 52-66 (1985).

[Gan 83] Ganzinger, H. Parameterized specifications: parameter passing and implementation with respect to observability. *TOPLAS 5*, 3 pp. 318-354 (1983).

[GGM 76] Giarratana, V., Gimona, F. and Montanari, U. Observability concepts in abstract data type specification. *Proc. 5th Intl. Symp. on Mathematical Foundations of Computer Science*, Gdansk. Springer LNCS 45, pp. 576-587 (1976).

[Gog 83] Gogolla, M. Algebraic specifications with partially ordered sorts and declarations. Fb. 169, Abteilung Informatik, Univ. of Dortmund (1983).

[GDLE 84] Gogolla, M., Drosten, K., Lipeck, U. and Ehrich, H.-D. Algebraic and operational semantics of specifications allowing exceptions and errors. *Theoretical Computer Science 34*, pp. 289-313 (1984).

[GB 80] Goguen, J.A. and Burstall, R.M. CAT, a system for the structured elaboration of correct programs from structured specifications. Technical report CSL-118, SRI International (1980).

[GB 84a] Goguen, J.A. and Burstall, R.M. Introducing institutions. *Proc. Logics of Programming Workshop* (E. Clarke and D. Kozen, eds.), Carnegie-Mellon University. Springer LNCS 164, pp. 221-256 (1984).

[**GB 84b**] Goguen, J.A. and Burstall, R.M. Some fundamental algebraic tools for the semantics of computation. Part 1: Comma categories, colimits, signatures and theories. *Theoretical Computer Science 31*, pp. 175-210 (1984).

[**GB 86**] Goguen, J.A. and Burstall, R.M. A study in the foundations of programming methodology: specifications, institutions, charters and parchments. *Proc. Workshop on Category Theory and Computer Programming*, Guildford. Springer LNCS 240, pp. 313-333 (1986).

[**GJM 85**] Goguen, J.A., Jouannaud, J.-P. and Meseguer, J. Operational semantics for order-sorted algebra. *Proc. 12th Intl. Colloq. on Automata, Languages and Programming*, Nafplion, Greece. Springer LNCS 194, pp. 221-231 (1985).

[**GM 82**] Goguen, J.A. and Meseguer, J. Universal realization, persistent interconnection and implementation of abstract modules. *Proc. 9th Intl. Colloq. on Automata, Languages and Programming*, Aarhus. Springer LNCS 140, pp. 265-281 (1982).

[**GTW 76**] Goguen, J.A., Thatcher, J.W. and Wagner, E.G. An initial algebra approach to the specification, correctness, and implementation of abstract data types. IBM research report RC 6487 (1976). Also in: *Current Trends in Programming Methodology, Vol. 4: Data Structuring* (R.T. Yeh, ed.), Prentice-Hall, pp. 80-149 (1978).

[**GTWW 77**] Goguen, J.A., Thatcher, J.W., Wagner, E.G. and Wright, J.B. Initial algebra semantics and continuous algebras. *JACM 24*, 1 pp. 68-95 (1977).

[**Gor 79**] Gordon, M.J. *Denotational descriptions of Programming Languages.* Springer (1979).

[**Gut 75**] Guttag, J.V. The specification and application to programming of abstract data types. Ph.D. thesis, Univ. of Toronto (1975).

[**GH 83**] Guttag, J.V. and Horning, J.J. Preliminary report on the Larch Shared Language. Report CSL-83-6, Computer Science Laboratory, Xerox PARC (1983).

[**Kam 83**] Kamin, S. Final data types and their specification. *TOPLAS 5*, 1 pp. 97-121 (1983).

[**Lar 86**] Larsen, K. Context-dependent bisimulation between processes. Ph.D. thesis, Dept. of Computer Science, Univ. of Edinburgh (1986).

[**Lip 83**] Lipeck, U. Ein algebraischer Kalkül für einer strukturierten Entwurf von Datenabstraktionen. Ph.D. thesis, Abteilung Informatik, Universität Dortmund (1983).

[**LB 77**] Liskov, B.H. and Berzins, V. An appraisal of program specifications. Computation Structures Group memo 141-1, Laboratory for Computer Science, MIT (1977).

[**MacL 71**] MacLane, S. *Categories for the Working Mathematician.* Springer (1971).

[**MacQ 85**] MacQueen, D.B. Modules for Standard ML. *Polymorphism 2*, 2 (1985). See also: *Proc. 1984 ACM Symp. on LISP and Functional Programming*, Austin, Texas, pp. 198-207.

[**MW 80**] Manna, Z. and Waldinger, R. A deductive approach to program synthesis. *ACM Trans. on Prog. Lang. and Systems 2* pp. 92-121 (1980).

[**MG 83**] Meseguer, J. and Goguen, J.A. Initiality, induction and computability. *Algebraic Methods in Semantics* (M. Nivat and J. Reynolds, eds.), Cambridge Univ. Press, pp. 459-541 (1983).

[**Mil 78**] Milner, R.G. A theory of type polymorphism in programming. *Journal of Computer and Systems Sciences 17*, 3 pp. 348-375 (1978).

[**Mil 85**] Milner, R.G. The Standard ML core language. *Polymorphism 2*, 2 (1985). See also: A proposal for Standard ML. *Proc. 1984 ACM Symp. on LISP and Functional Programming*, Austin, Texas, pp. 184-197.

[**Moo 56**] Moore, E.F. Gedanken-experiments on sequential machines. In: *Automata Studies* (C.E. Shannon and J. McCarthy, eds.), Princeton Univ. Press, pp. 129-153 (1956).

[**Ore 83**] Orejas, F. Characterizing composability of abstract implementations. *Proc. Intl. Conf. on Foundations of Computation Theory*, Borgholm, Sweden. Springer LNCS 158, pp. 335-346 (1983).

[**PB 85**] Parisi-Presicce, F. and Blum, E.K. The semantics of shared submodules specifications. *Proc. 10th Colloq. on Trees in Algebra and Programming*, Joint Conf. on Theory and Practice of Software Development (TAPSOFT), Berlin. Springer LNCS 185, pp. 359-373 (1985).

[**Pep 83**] Pepper, P. On the correctness of type transformations. Talk at 2nd Workshop on Theory and Applications of Abstract Data Types, Passau (1983).

[**Plo 77**] Plotkin, G.D. LCF considered as a programming language. *Theoretical Computer Science 5*, pp. 223-255 (1977).

[**Rei 81**] Reichel, H. Behavioural equivalence – a unifying concept for initial and final specification methods. *Proc. 3rd Hungarian Computer Science Conference*, Budapest, pp. 27-39 (1981).

[**ST 85a**] Sannella, D.T. and Tarlecki, A. Some thoughts on algebraic specification. *Proc. 3rd Workshop on Theory and Applications of Abstract Data Types*, Bremen. Springer Informatik-Fachberichte Vol. 116, pp. 31-38 (1985).

[**ST 85b**] Sannella, D.T. and Tarlecki, A. Program specification and development in Standard ML. *Proc. 12th ACM Symp. on Principles of Programming Languages*, New Orleans, pp. 67-77 (1985).

[ST 86a] Sannella, D.T. and Tarlecki, A. Specifications in an arbitrary institution. Report CSR-184-85, Dept. of Computer Science, Univ. of Edinburgh; to appear in *Information and Control.* See also: Building specifications in an arbitrary institution, *Proc. Intl. Symposium on Semantics of Data Types*, Sophia-Antipolis. Springer LNCS 173, pp. 337-356 (1984).

[ST 86b] Sannella, D.T. and Tarlecki, A. Extended ML: an institution-independent framework for formal program development. *Proc. Workshop on Category Theory and Computer Programming*, Guildford. Springer LNCS 240, pp. 364-389 (1986).

[ST 87a] Sannella, D.T. and Tarlecki, A. On observational equivalence and algebraic specification. *Journal of Computer and Systems Sciences 34*, pp. 150-178 (1987). Extended abstract in: *Proc. 10th Colloq. on Trees in Algebra and Programming*, Joint Conf. on Theory and Practice of Software Development (TAPSOFT), Berlin. Springer LNCS 185, pp. 308-322 (1986).

[ST 87b] Sannella, D.T. and Tarlecki, A. Toward formal development of programs from algebraic specifications: implementations revisited (extended abstract). *Proc. 12th Colloq. on Trees in Algebra and Programming*, Joint Conf. on Theory and Practice of Software Development (TAPSOFT), Pisa. Springer LNCS 249, pp. 96-110 (1987).

[SW 82] Sannella, D.T. and Wirsing, M. Implementation of parameterised specifications. Report CSR-103-82, Dept. of Computer Science, Univ. of Edinburgh. Extended abstract in: *Proc. 9th Intl. Colloq. on Automata, Languages and Programming*, Aarhus. Springer LNCS 140, pp. 473-488 (1982).

[SW 83] Sannella, D.T. and Wirsing, M. A kernel language for algebraic specification and implementation. Report CSR-131-83, Dept. of Computer Science, Univ. of Edinburgh. Extended abstract in: *Proc. Intl. Conf. on Foundations of Computation Theory*, Borgholm, Sweden. Springer LNCS 158, pp. 413-427 (1983).

[Sch 86] Schoett, O. Data abstraction and the correctness of modular programming. Ph.D. thesis, Univ. of Edinburgh (1986).

[Tar 85] Tarlecki, A. On the existence of free models in abstract algebraic institutions. *Theoretical Computer Science 37* pp. 269-304 (1985).

[Tar 86a] Tarlecki, A. Quasi-varieties in abstract algebraic institutions. *Journal of Computer and Systems Sciences 33* pp. 333-360 (1986).

[Tar 86b] Tarlecki, A. Software-system development — an abstract view. *Proc. 10th IFIP World Computer Congress*, Dublin. North-Holland, pp. 685-688 (1986).

[TW 86] Tarlecki, A. and Wirsing, M. Continuous abstract data types. *Fundamenta Informaticae 9*, pp. 95-126 (1986). Extended abstract: Continuous abstract data types - basic machinery and results. *Proc. Intl. Conf. on Fundamentals of Computation Theory*, Cottbus, GDR. Springer LNCS 199 (1985), pp. 431-441.

[**Wand 82**] Wand, M. Specifications, models, and implementations of data abstractions. *Theoretical Computer Science 20* pp. 3-32 (1982).

[**Wir 86**] Wirsing, M. Structured algebraic specifications: a kernel language. *Theoretical Computer Science* 42, pp. 123-249.

[**Zil 74**] Zilles, S.N. Algebraic specification of data types. Computation Structures Group memo 119, Laboratory for Computer Science, MIT (1974).