A KERNEL LANGUAGE FOR ALGEBRAIC SPECIFICATION AND IMPLEMENTATION
-- EXTENDED ABSTRACT* --

Donald Sannella
Department of Computer Science
University of Edinburgh

Martin Wirsing
Fakultät für Informatik
Universität Passau

Abstract

A kernel specification language called ASL is presented. ASL comprises five fundamental but powerful specification-building operations and has a simple semantics. Behavioural abstraction with respect to a set of observable sorts can be expressed, and (recursive) parameterised specifications can be defined using a more powerful and more expressive parameterisation mechanism than usual. A simple notion of implementation permitting vertical and horizontal composition (i.e. it is transitive and monotonic) is adopted and compared with previous more elaborate notions. A collection of identities is given which can provide a foundation for the development of programs by transformation.

1 Introduction

In recent years there has been a great deal of work on developing the algebraic approach to specification of data types and programs. Guttag [Gut 75] and others began by viewing an abstract data type as a class of heterogeneous algebras and showing how such a type can be specified by a signature (a collection of sorts and operators) together with a set of axioms. For quite simple data types (e.g. natural numbers) such an approach can be used without problems. But it is more convenient to build large algebraic specifications in a structured fashion by combining and modifying smaller specifications. Several specification languages have been developed to support this structured approach, including Clear [BG 77, 80] (cf. [HKR 80]), CIP-L [Bau 81] and LOOK [ZLT 82, ETLZ 82]. Each language provides a certain set of operations for use in building specifications together with a convenient syntax and a formal semantics.

We describe here (section 3) a kernel language for algebraic specification called ASL (a significantly revised version of the ASL in [Wir 82]). This language is nothing more than a collection of five fundamental but powerful specification-building operations. It has a simple semantics in comparison with high-level specification languages like Clear, CIP-L and LOOK. ASL is intended mainly as a kernel language rather than for writing specifications. That is, it provides a solid foundation on top of which high-level specification languages can be built. The semantics of the constructs of such a language would then be expressed by mapping them into ASL expressions.

ASL differs from previous specification languages in a number of important respects:

- ASL is a language for describing classes of *algebras* rather than for building sets of *axioms* (theories) like most other specification languages. Some of the operations of ASL (e.g. **abstract**) cannot be viewed as simple operations on theories.
- An ASL specification may be *loose* (i.e. it may possess non-isomorphic models). Loose specifications are also allowed by Clear, CIP-L and LOOK but not by some previous approaches (e.g. the initial algebra approach [ADJ 76, 78]). Loose specifications can be precise while leaving some freedom of choice (e.g. to the implementator).
- ASL is oriented toward a 'behavioural' approach to specification rather than toward an initial or final algebra approach. Along with [GGM 76] and others, we argue that it is usually irrelevant how values of a sort are represented in an algebra as long as the desired input/output relation is satisfied. ASL includes a very general abstraction operation which can be used to behaviourally abstract from a specification, relaxing interpretation to those algebras which are behaviourally equivalent to a model. This can be used to write 'abstract model' specifications as in [LB 77].

---

*The full version of this paper is available as Report CSR-131-83, Dept. of Computer Science, Univ. of Edinburgh.

- The approach to parameterisation in ASL (section 5) is more general and flexible than in other languages. The main difference is that the signature of the result of applying a parameterised specification may depend on the signature of the actual parameter in a more flexible way than before. Since parameterised specifications may be recursive we can write 'abstract domain equations' as in [HR 80] and [EL 81].

Since ASL is a powerful specification language it is possible to adopt a simple notion of the implementation of one specification by another ($T \rightsquigarrow T'$) which can easily be extended to the case of parameterised specifications. Then the transitivity of implementations (vertical composition -- if $T \rightsquigarrow T'$ and $T' \rightsquigarrow T''$ then $T \rightsquigarrow T''$) follows immediately and the monotonicity of ASL's operations gives horizontal composition (specification-building operations preserve implementations, e.g. if $P \rightsquigarrow P'$ and $A \rightsquigarrow A'$ then $P(A) \rightsquigarrow P'(A')$). These results permit the development of programs from ASL specifications in a gradual and modular fashion. A number of more elaborate notions of implementation can be expressed using ASL, including notions which coincide with or approximate most previously proposed definitions.

An advantage of the kernel language approach is that facts about the basic operations (easy to prove because of the simple semantics) automatically extend to facts concerning the high-level constructs of any language built on top of the kernel. Thus the ASL identities and relations given in section 7 extend to identities and relations on any language built on top of ASL. These could be used as the basis for a methodology of program development by transformation of specifications.

2 Algebraic background

In this section the algebraic definitions which will be needed throughout the rest of the paper are presented.

2.1 Signatures

In order to get a clean mathematical semantics of parameterisation with fixed points of recursive parameterised specifications (see section 5) we need a more elaborate definition of signatures and signature morphisms than the standard one (as in e.g. [BG 80]). First, we need to fix the set of possible sorts and operators to ensure that the possible signatures and signature morphisms form sets rather than classes. Then we extend the usual definition of a signature morphism -- as a total function from one (finite) signature to another -- to a partial function from the (infinite) set of all sorts and operators into that set. Since we think that signature morphisms should be computable we require them to be partial recursive functions. Formally:

Fix arbitrary countably infinite sets Λ of sorts and Γ of operators.

Def: A *signature* Σ is a pair $\langle S,\Omega\rangle$ where $S\subseteq\Lambda$ is a set (of *sorts*) and Ω is a family of subsets of Γ (*operators*) indexed by $S^*\times S$. The index of a set $O\in\Omega$ is the *type* of every element of O. Let the *universal signature* Σ_{univ} be $\langle S_{univ},\Omega_{univ}\rangle$ where $S_{univ}=\Lambda$ and Ω_{univ} is the family $(\Gamma)_{us\in\Lambda^*\times\Lambda}$.

Def: A *signature morphism* σ is a pair $\langle f,g\rangle$ where $f:S_{univ}\rightarrow S_{univ}$ is a partial recursive function and g is a family of partial recursive functions $g_{us}:(\Omega_{univ})_{us}\rightarrow(\Omega_{univ})_{f^*(u)f(s)}$, where $u\in S^*_{univ}$, $s\in S_{univ}$ and $f^*:S^*_{univ}\rightarrow S^*_{univ}$ is the extension of f to strings of sorts. We write $\sigma:\Sigma\rightarrow\Sigma'$ if $\Sigma=\sigma^{-1}(\Sigma')$ (i.e. Σ is the inverse image of Σ'). Then $\sigma:\Sigma\rightarrow\Sigma'$ implies that $\sigma|_{\Sigma}$ (σ restricted to the domain Σ) is a total function into Σ'. Furthermore, we write $\sigma(s)$ for $f(s)$ and $\sigma(\omega)$ for $g_{us}(\omega)$, where $\omega\in\Omega_{us}$.

Note that infinite signatures are permitted; for examples showing how this could be useful see [Wir 82]. Moreover, the definition of signature permits overloading (i.e. several operators having the same name but different types) as does the definition in [BG 80].

According to the above definition, a signature morphism $\sigma:\Sigma\rightarrow\Sigma'$ is almost the same as in [BG 80]; the difference is that a signature morphism σ can simultaneously be $\sigma:\Sigma A\rightarrow\Sigma A'$ and $\sigma:\Sigma B\rightarrow\Sigma B'$ for $\Sigma A\neq\Sigma B$ and $\Sigma A'\neq\Sigma B'$. This difference is important. Signature morphisms are used for renaming the sorts and operators of a specification (via the **derive** operation); since it is possible to define signature morphisms which 'make sense' over a range of signatures, we can (for example) write a parameterised specification which systematically renames the sorts and operators of any specification it is given as an actual parameter, which is impossible in Clear, CIP-L, LOOK, or the ADJ approach to parameterisation [ADJ 78, 80]. This point is discussed at greater length in section 5.

2.2 Algebras

The definitions of a (total) Σ-algebra A with carriers $|A|$ and of a Σ-homomorphism are as usual, except that the carrier $|A|_s$ is required to be non-empty for every $s \in sorts(\Sigma)$. The reason for this requirement is that permitting such degenerate algebras would give rise to problems in later definitions (see the definitions of reachable and $\equiv_W$ in section 2.4). The class of all Σ-algebras will be denoted $Alg(\Sigma)$.

Given a Σ'-algebra A' and an injective signature morphism $\sigma:\Sigma\rightarrow\Sigma'$, we can recover the Σ-algebra buried inside A' (since A' is just an extension of this algebra). The definition extends without modification to the case in which σ is not injective, where the Σ-algebra will contain multiple copies of some of the carriers and operations of A'.

Def: If $\sigma=\langle f,g\rangle$ is a signature morphism $\sigma:\Sigma\rightarrow\Sigma'$ and A' is a Σ'-algebra, then the *σ-restriction* of A', written $A'|_\sigma$ is the Σ-algebra with carrier $|A|_s=|A'|_{f(s)}$ for each $s\in sorts(\Sigma)$, and $\omega_A=g(\omega)_{A'}$ for each $\omega\in opns(\Sigma)$. When σ is obvious we sometimes use the notation $A'|_\Sigma$.

2.3 Terms and the term algebra

Σ-terms, the translation $\sigma(t)$ of a Σ-term t by a signature morphism $\sigma:\Sigma\rightarrow\Sigma'$, and the term algebra $W_\Sigma(X)$ are defined as usual. For some choices of Σ and X, $W_\Sigma(X)$ will have an empty carrier for some sort $s\in sorts(\Sigma)$ (in this case $W_\Sigma(X)$ is not an algebra, strictly speaking). We then say that $W_\Sigma(X)$ is *empty* in s. If we have a Σ-term t and an assignment $\phi:X\rightarrow|A|$ of values in A to variables then the value of t in A under ϕ is denoted $\phi^\#(t)$ (i.e. $\phi^\#:W_\Sigma(X)\rightarrow A$ is the unique homomorphism extending ϕ).

Def: If Σ is a signature then let X_Σ be a sorts(Σ)-indexed set of variables with $(X_\Sigma)_s=\mathbb{N}$ for each $s\in sorts(\Sigma)$. If Σ is obvious we will write X instead of X_Σ. We write x_1, x_2, y, a etc. instead of 1_s, 2_s etc. $\in(X_\Sigma)_s$.

Notation: If Z is an S-indexed set and $S'\subseteq S$, then $Z_{S'}$ denotes the restriction of Z to S'. For example, the notation $|W_\Sigma(X_S)|_{S'}$ refers to the (S'-indexed) set of Σ-terms of sorts in S' containing variables of sorts in S.

2.4 Properties of algebras and W-equivalence

Def: If A is a Σ-algebra and $S\subseteq sorts(\Sigma)$ is a set of sorts, then A is *reachable* on S if for every sort $s\in S$ and every carrier element $a\in|A|_s$ there is a term $t\in|W_\Sigma(X_{S'})|_s$ and assignment $\phi:X_\Sigma\rightarrow|A|$ such that $\phi^\#(t)=a$, where $S'=sorts(\Sigma)-S$. Unreachable carrier elements are called *junk*. If an algebra is reachable on all sorts then it is *finitely generated*.

Equivalently, A is reachable on S iff there exists a surjective homomorphism $f:W_\Sigma(X_{S'})\rightarrow A$.

Def: A *Σ-formula* is a first-order equational formula on Σ; that is, a formula built from Σ-terms using = (term equality), the logical connectives $\neg$, $\wedge$, $\vee$ and $\Rightarrow$ and the quantifiers $\forall$ and $\exists$. Satisfaction of a Σ-formula e by a Σ-algebra A ($A\models e$) is defined as usual.

In fact, any notion of Σ-formula will do; we only need to know when a Σ-formula is satisfied by a Σ-algebra. The definition above gives one example of such a notion. The semantics of ASL can thus be viewed as parameterised by the notions of formula and satisfaction. The semantics of Clear [BG 80] is parameterised by an *institution* [GB 83] -- i.e. by notions of signature, algebra, formula and satisfaction which must satisfy certain properties. The semantics of ASL can be made independent of the notion of algebra and (to some extent) of the notion of signature as well, but the properties which the notions must satisfy are different (see [SW 83] for details).

Def: If A, A' are Σ-algebras and $W\subseteq|W_\Sigma(X)|$ then A and A' are *W-equivalent* ($A\equiv_W A'$) if there are surjective assignments $\phi:X\rightarrow|A|$ and $\phi':X\rightarrow|A'|$ such that $\forall t,t'\in W.\ (\phi^\#(t)=\phi^\#(t') \Leftrightarrow \phi'^\#(t)=\phi'^\#(t'))$.

This definition generalises the various notions of *behavioural equivalence* in the literature. If $OBS \subseteq sorts(\Sigma)$ is a set of *observable sorts* then two Σ-algebras are considered to be behaviourally equivalent with respect to OBS if all computations yielding a result of observable sort give the same result in both algebras. There is some disagreement over which class of inputs to these computations should be considered: $|W_\Sigma(X)|_{OBS}$-equivalence (all inputs) is behavioural equivalence according to [Rei 81] and [GM 82]; $|W_\Sigma(X_{OBS})|_{OBS}$-equivalence (inputs of observable sorts) is behavioural equivalence in the sense of [Sch 82] and [GM 83]; and $|W_\Sigma(\phi)|_{OBS}$-equivalence (no inputs) is the same as behavioural (or I/O) equivalence in [BM 81] and [Kam 83] (and implied by [GGM 76]) except that in these three papers only finitely generated algebras are considered. There are other choices for W which yield interesting equivalences; one of these (used to define the junk operation) is given in the next section.

3 The language ASL and its semantics

ASL is a language for describing classes of algebras. It contains five constructs, each construct embodying a primitive operation on classes of algebras. These are:

- Form a *basic specification* having a given signature Σ and given axioms E. This specifies the class of all Σ-algebras satisfying E.
- Take the *sum* T+T' of two specifications, specifying the class of algebras obtained by combining a model of T with a model of T'. This allows large specifications to be built from smaller specifications.
- Restrict interpretation to those models which are *reachable* on certain sorts. Requiring reachability is the same as restricting by a certain second-order principle which is equivalent to structural induction.
- *Derive* a specification from a richer specification by renaming or forgetting some sorts and operators but otherwise retaining the class of models. This can be used to hide the details of a constructive specification to give a more abstract result.
- *Abstract* away from certain details of the specification, relaxing interpretation to those algebras which are the same as a model with respect to some observability criterion. With an appropriate observability criterion this amounts to *behavioural abstraction* with respect to a set of observable sorts.

These fundamental and mutually independent operations can be composed to give higher-level operations for building specifications in a wide variety of ways. ASL is a *kernel* language which provides a foundation on top of which high-level specification languages such as Clear, CIP-L and LOOK can be built. The semantics of the specification-building constructs of these languages can be expressed by mapping them into ASL expressions. A specification language has been defined on top of a previous version of ASL [Gau 83] and we have informally redefined the specification part of CIP-L on top of ASL (see [Wir 82] for the basic idea). We do not intend that ASL itself be used directly for writing specifications, although in the next section examples are given showing that this is possible.

<u>Syntax</u>

Expr	::=	Basic-Spec | Sum | Reachable | Derive | Abstract
Basic-Spec	::=	< signature, set of formulas >
Sum	::=	Expr + Expr
Reachable	::=	**reachable** Expr **on** set of sorts
Derive	::=	**derive from** Expr **by** signature morphism
Abstract	::=	**abstract** Expr **wrt** set of terms

No special syntax is provided for signatures, sets, formulas or signature morphisms; the usual mathematical notation will be used in examples.

<u>Semantics</u>

The semantics of ASL is defined by two functions

Sig: Expr $\rightarrow$ signature
Mod: Expr $\rightarrow$ class of algebras

such that for any expression T, $Mod[\![T]\!]$ is a class of $Sig[\![T]\!]$-algebras. We use square brackets [ ] to denote classes. The definition of Sig below includes context conditions for each construct; if these are not satisfied then the expression is invalid. It is easy to prove that for any specification T, $Mod[\![T]\!]$ is closed under isomorphism.

$\mathrm{Sig}[\![<\Sigma,E>]\!] = \Sigma$ the operators and sorts used in E are in Σ

$\mathrm{Sig}[\![T+T']\!] = \mathrm{Sig}[\![T]\!] \cup \mathrm{Sig}[\![T']\!]$

$\mathrm{Sig}[\![\textbf{reachable } T \textbf{ on } S]\!] = \mathrm{Sig}[\![T]\!]$ $S \subseteq \mathrm{sorts}(\mathrm{Sig}[\![T]\!])$

$\mathrm{Sig}[\![\textbf{derive from } T \textbf{ by } \sigma]\!] = \Sigma$ $\sigma : \Sigma \to \mathrm{Sig}[\![T]\!]$ (recall that $\Sigma = \sigma^{-1}(\mathrm{Sig}[\![T]\!])$)

$\mathrm{Sig}[\![\textbf{abstract } T \textbf{ wrt } W]\!] = \mathrm{Sig}[\![T]\!]$ $W \subseteq |W_\Sigma(X)|$, where $\Sigma = \mathrm{Sig}[\![T]\!]$

$\mathrm{Mod}[\![<\Sigma,E>]\!] = \{ A \in \mathrm{Alg}(\Sigma) \mid A \models E \}$

$\mathrm{Mod}[\![T+T']\!] = \{ A \in \mathrm{Alg}(\mathrm{Sig}[\![T+T']\!]) \mid A|_{\mathrm{Sig}[\![T]\!]} \in \mathrm{Mod}[\![T]\!] \text{ and } A|_{\mathrm{Sig}[\![T']\!]} \in \mathrm{Mod}[\![T']\!] \}$

$\mathrm{Mod}[\![\textbf{reachable } T \textbf{ on } S]\!] = \{ A \in \mathrm{Mod}[\![T]\!] \mid A \text{ is reachable on } S \}$

$\mathrm{Mod}[\![\textbf{derive from } T \textbf{ by } \sigma]\!] = \{ A|_\sigma \mid A \in \mathrm{Mod}[\![T]\!] \}$

$\mathrm{Mod}[\![\textbf{abstract } T \textbf{ wrt } W]\!] = \{ A \in \mathrm{Alg}(\mathrm{Sig}[\![T]\!]) \mid \exists A_0 \in \mathrm{Mod}[\![T]\!] . (A_0 \equiv_W A) \}$

The + operation is not quite the same as + in Clear, since no account is taken of shared subspecifications. This feature of Clear is designed to make it easy to build specifications without worrying about the names of sorts and operators. Such high-level features have no place in a kernel language like ASL. The same effect can be achieved manually by use of + in conjunction with the **derive** operation.

The **reachable** construct restricts interpretation to models which are reachable on the given set of sorts S. It can be used to express the **data** operation in 'hierarchical' Clear [SW 82] and the **based on** construct of CIP-L. The **data** operation of 'ordinary' Clear [BG 80] and the **constraining** operation of LOOK [ETLZ 82] cannot be fully expressed in ASL because they restrict to the class of *initial* models. We do not view this as a disadvantage. In our opinion the initial algebra approach to specification [ADJ 76] adopted by Clear and LOOK has more problems than advantages; this view seems to be shared by others, e.g. [GGM 76], [Wand 79] and [Bau 81]. Some of these problems are: initial models do not always exist for specifications having axioms which include $\forall$ or $\exists$; to prove that an inequality $t \neq t'$ holds, one must in general prove that the equality $t=t'$ is not provable; implementations have unpleasant properties in the presence of an operation for restricting to initial models [SW 82]; and in the stepwise development of specifications and programs, the set of constructors for a data type is often fixed at an early stage, whereas the inequalities satisfied by the type are only established once all design decisions have been made. No power is lost by abandoning the initial algebra approach [BBTW 81].

The **derive** operation corresponds to **derive** in Clear. This already gives a hint of abstraction because it is possible to construct a specification which employs auxiliary sorts and operators and then use the **derive** operation to forget them, retaining only the semantics of the remaining sorts and operators. But this is not real abstraction, because the structure induced by the auxiliary operators remains (compare the examples List and Impoverished-List in the next section). The real abstraction is done by the **abstract** operation which ignores invisible structure (compare the examples Impoverished-List and Behavioural-Set). The result of abstracting from T with respect to a set W of visible terms is the class of algebras which are W-equivalent to a model of T. No similar operation is found in any other specification language, so far as we are aware.

An interesting use of **abstract** is to express behavioural abstraction with respect to a set of observable sorts:

$\textbf{behaviour } T \textbf{ wrt } OBS =_{def} \textbf{abstract } T \textbf{ wrt } |W_\Sigma(X_{OBS})|_{OBS}$ where $\Sigma = \mathrm{sig}[\![T]\!]$ and $OBS \subseteq \mathrm{sorts}(\Sigma)$

(Please note that **behaviour** is only an abbreviation for a special case of **abstract**; it is not a new operation of ASL.) This gives the class of all algebras which are behaviourally equivalent (with respect to OBS) to a model of T, using a notion of behavioural equivalence due to [Sch 82] and [GM 83] (this is the notion which seems to fit most gracefully with our notion of implementation). This operation can be used to abstract from a concretely-specified input/output behaviour as in the 'abstract model specifications' of [LB 77]. It also allows us to adopt a very simple notion of implementation, as discussed in section 6.

Another use of **abstract** is to express the **junk** operation:

junk T **on** S $=_{def}$ **abstract** T **wrt** $|W_{\Sigma}(X_{S'})|$ where Σ=sig⟦T⟧, S⊆sorts(Σ) and S'=sorts(Σ)-S

This gives those algebras which are the same as models of T except that they may contain arbitrary junk (non-reachable values) in sorts S. It can be seen as a kind of dual to the **reachable** operation. Note that some of the models of **junk** T **on** S will be reachable on S even if none of the models of T are. We can select these by applying the **reachable** operation. This particular combination occurs often so we give it a name:

restrict T **on** S $=_{def}$ **reachable** (**junk** T **on** S) **on** S where S⊆sorts(T)

This gives the class of reachable (on S) subalgebras of models of T which are unchanged for sorts not in S.

The following abbreviations will be convenient in the sequel:

reachable T	$=_{def}$	**reachable** T **on** sorts(T)	restrict to the finitely generated models of T
junk T	$=_{def}$	**junk** T **on** sorts(T)	allow arbitrary junk in all sorts
restrict T	$=_{def}$	**restrict** T **on** sorts(T)	finitely generated subalgebras of models of T

Clear's **enrich** operation (add some sorts, operators and axioms to a specification) can be expressed using the + and basic-spec operations:

enrich T **by sorts** S **opns** F **axioms** E $=_{def}$ T + < < sorts(T) ∪ S, opns(T) ∪ F >, E >

The notation T=T' (where T and T' are ASL expressions) will be used to abbreviate Mod⟦T⟧=Mod⟦T'⟧; similarly, T⊆T' means Sig⟦T⟧=Sig⟦T'⟧ and Mod⟦T⟧⊆Mod⟦T'⟧.

4 Examples

The specifications of booleans, natural numbers, and lists of natural numbers in ASL are much the same (except for syntax) as they would be in CIP-L:

```
Bool =def reachable
        enrich ∅ by
          sorts   bool
          opns    true, false : bool
          axioms  true ≠ false

Nat =def reachable
        enrich ∅ by
          sorts   nat
          opns    0 : nat
                  succ : nat → nat
          axioms  0 ≠ succ(x)
                  succ(x) = succ(y) ⟺ x = y
```

```
List =def reachable
        enrich Bool + Nat by
          sorts   list
          opns    nil : list
                  cons : nat, list → list
                  head : list → nat
                  tail : list → list
                  ∈ : nat, list → bool
          axioms  head(cons(a,l)) = a
                  tail(cons(a,l)) = l
                  a ∈ nil = false
                  a ∈ cons(a,l) = true
                  a ≠ b ⟹ a ∈ cons(b,l) = a ∈ l
```

All models of each of these specifications are isomorphic to the standard model. The axioms in Bool and Nat are required to avoid trivial models, in contrast to Clear and LOOK. The inequations in Bool and Nat together with the axioms of List induce inequations like cons(a,cons(a,l))≠cons(a,l) so it is not necessary to state them explicitly.

Suppose ΣSet denotes the following signature:

```
Sig⟦Bool⟧ ∪ Sig⟦Nat⟧ ∪ ( sorts  set
                         opns   ∅ : set
                                add : nat, set → set
                                ∈ : nat, set → bool )
```

and σ:ΣSet→sig⟦List⟧ is the signature morphism with σ(*set*)=*list*, σ(∅)=*nil*, σ(*add*)=*cons* and $\sigma(x)=x$ for all other sorts and operators x in ΣSet. Then the specification

Impoverished-List $=_{def}$ **derive from** List **by** σ

has exactly the same class of models as List except for the absence of *head* and *tail* and the renaming of the sort *list* and some of the remaining operators. The formulas $add(a,add(a,S)) \neq add(a,S)$ and $a \neq b \Longrightarrow add(a,add(b,S)) \neq add(b,add(a,S))$ still hold in every model of Impoverished-List, although there is no longer any context in which values like [1,2], [2,1] and [2,2,1] can be distinguished.

Behavioural abstraction results in a broader class of models:

Behavioural-Set $=_{def}$ **behaviour** Impoverished-List **wrt** {nat, bool}

Models of Behavioural-Set include the models of Impoverished-List as well as the algebra with a carrier consisting of the set of *bags* of natural numbers (satisfying $add(a,add(b,S))=add(b,add(a,S))$ and $add(a,add(a,S)) \neq add(a,S)$) and the standard model of finite sets with carrier $\mathcal{P}(\mathbb{N})$ (where $add(a,add(b,S))=add(b,add(a,S))$ and $add(a,add(a,S))=add(a,S)$) and all algebras isomorphic to them. All models may include arbitrary junk for the sort *set*. Trivial models (satisfying e.g. $add(a,\emptyset)=\emptyset$) are still excluded. If we form a specification from List which is similar to Behavioural-Set but with the order of **derive** and **behaviour** reversed, the result is identical to Impoverished-List except that its models may contain junk (and different from Behavioural-Set):

junk Impoverished-List **on** {set} = **derive from** (**behaviour** List **wrt** {bool, nat}) **by** σ $\subset$ Behavioural-Set

Behavioural-Set has almost the same class of models as the following more direct specification of sets:

```
Loose-Set =def enrich Bool + Nat by
                 sorts  set                        axioms a ∈ ∅ = false
                 opns   ∅ : set                           a ∈ add(a,S) = true
                        add : nat, set → set              a ≠ b ⟹ a ∈ add(b,S) = a ∈ S
                        ∈ : nat, set → bool
```

The only difference between the models of Behavioural-Set and Loose-Set is that models of Behavioural-Set may contain arbitrary junk of sort *set*, while any junk in models of Loose-Set must satisfy the axioms of Loose-Set:

Behavioural-Set = **junk** Loose-Set **on** {set} = **behaviour** Loose-Set **wrt** {nat, bool}

In order to restrict interpretation to the standard model of sets we must add more information to Loose-Set:

```
Set =def enrich reachable Loose-Set on {set} by
           axioms add(a,add(b,S)) = add(b,add(a,S))
                  add(a,add(a,S)) = add(a,S)
```

The only model of Set (up to isomorphism) is the standard model. The same class of models results if the order of **enrich** and **reachable** is switched. Now

Behavioural-Set = **behaviour** Set **wrt** {bool, nat} = **behaviour** (**junk** Set **on** {set}) **wrt** {**bool, nat**}

In fact, if Set is **enrich** Loose-Set **by axioms** E or **enrich reachable** Loose-Set **on** {set} **by axioms** E where E is any set of axioms consistent with those in Loose-Set, then these identities still hold.

Set can be extended by adding a new value, an 'infinite set' which contains every natural number:

```
Infinite-Set =def reachable enrich junk Set on {set} by
                        opns    infset : set
                        axioms  add(a,infset) = infset
                                a ∈ infset = true            on {set}
```

In every model of Infinite-Set the value of *infset* will be different from every other value of sort *set*. Apart from this new value, the models of Infinite-Set are exactly the models of Set. This kind of extension (in which a new constructor is added to a previously **reachable**-restricted sort) is not possible in Clear, CIP-L or LOOK.

If we use **derive** to forget the operator *infset* the result is almost the same as Set; the only difference is that every model will contain a single junk element. We can apply **restrict** to obtain their reachable subalgebras:

Set = **restrict** (**derive from** Infinite-Set **by** σ) **on** {set}

where σ: Sig⟦ Set ⟧ $\hookrightarrow$ Sig⟦ Infinite-Set ⟧ is the inclusion.

Suppose that Loose-Set is enriched as follows:

Loose-Bag $=_{def}$ **enrich** Loose-Set **by**
opns howmany : nat, set $\rightarrow$ nat
axioms howmany(a, $\emptyset$) = 0
howmany(a, add(a, S)) = succ(howmany(a, S))
a $\neq$ b $\Longrightarrow$ howmany(a, add(b, S)) = howmany(a, S)

Recall that the models of Loose-Set included the standard model where add(a, add(a, S))=add(a, S) as well as models where add(a, add(a, S))$\neq$add(a, S). The models of Loose-Set in which repeated elements are ignored cannot be extended to give models of Loose-Bag; if {a, a}={a} then howmany(a, {a, a})=howmany(a, {a}) so 2=1. The other models of Loose-Set (extended by *howmany*) remain. The original models of Loose-Set (along with models containing arbitrary junk of sort *set*) can be regained by forgetting *howmany* and applying behavioural abstraction:

behaviour (**derive from** Loose-Bag **by** σ) **wrt** {nat, bool} = **junk** Loose-Set **on** {set}

where σ: Sig$[\![$ Loose-Set $]\!]\hookrightarrow$Sig$[\![$ Loose-Bag $]\!]$ is the inclusion.

Although the examples in this section are very small, they illustrate some of the things which can be accomplished using ASL. Some of these things are impossible in any other algebraic specification language, viz behavioural abstraction (as in the construction of Behavioural-Set from List) and the addition of a new element to a **reachable**-restricted sort (as in the construction of Infinite-Set from Set).

5 Parameterised specifications with recursion

The semantics of a nonparameterised specification consists (as described in section 3) of a signature Σ together with a class M of Σ-algebras, that is:

$[\![T]\!] = <\Sigma, M>$ where $\Sigma \subseteq \Sigma_{univ}$ and $M \subseteq Alg(\Sigma)$ such that M is closed under isomorphism

The collection of isomorphism classes of (countable) Σ-algebras forms a set for any Σ. Therefore the collection of possible pairs $<\Sigma, M>$ forms a set, which we will call SEM. If $<\Sigma, M> \in$ SEM, then Sig$<\Sigma, M>=\Sigma$ and Mod$<\Sigma, M>$=M. We will refer to classes of Σ-algebras which are closed under isomorphism as *Σ-model classes*.

The semantics of a parameterised specification is a function taking a member of SEM together with a signature morphism as argument and giving a member of SEM as a result (similar to Clear):

f : SEM X signature morphism $\rightarrow$ SEM

The generalisation to multiple parameters is not difficult but this presentation will be confined to the 1-parameter case. A parameterised specification is written λX:R[σ].B where X is the *formal parameter*, R is the *parameter requirement* (itself a specification), σ is the *formal fitting morphism* and B is the *body* (a specification which normally contains X, may contain σ, and may refer to the sorts and operators of R). Application is written (λX:R[σ].B)(ARG[ρ]) where ρ:Sig$[\![$ R $]\!]\rightarrow$Sig$[\![$ ARG $]\!]$ is the *fitting morphism* which matches the *actual parameter* ARG with R. In contrast to Clear, the fitting morphism ρ is available for use in the body B via the formal fitting morphism σ. The semantics of application is as follows (where B_ρ[ARG/X] is an abbreviation for B[ARG/X, ρ/σ, $\rho(\omega)/\omega$ for all $\omega \in$Sig$[\![$ R $]\!]$] -- the substitution into the body B of ARG for X and ρ for σ, and of $\rho(\omega)$ for ω for every sort or operator ω in Sig$[\![$ R $]\!]$):

$$\mathrm{Sig}[\![(\lambda X{:}R[\sigma].B)(ARG[\rho])]\!] = \mathrm{Sig}[\![B_\rho[ARG/X]]\!]$$

$$\mathrm{Mod}[\![(\lambda X{:}R[\sigma].B)(ARG[\rho])]\!] = \begin{cases} \mathrm{Mod}[\![B_\rho[ARG/X]]\!] & \text{if } [\, A|_\rho \mid A \in \mathrm{Mod}(ARG) \,] \subseteq \mathrm{Mod}[\![R]\!] \\ \mathrm{Alg}(\mathrm{Sig}[\![B_\rho[ARG/X]]\!]) & \text{otherwise} \end{cases}$$

Note that ARG is a semantic object from SEM, not a specification; this is necessary for the semantics of recursion.

This semantics describes a parameterisation mechanism which is more powerful and more expressive than in other languages. Using this we can define parameterised specifications in which the signature of the result depends on the signature of the actual parameter in a more flexible way than previously possible. For example, suppose we want to write a parameterised specification called Copy which produces a specification containing two copies of its actual parameter (i.e. two copies of all its sorts and operators). In Clear, CIP-L, LOOK and the ADJ approach to parameterisation this is impossible; the parameterised specification can only transform the part of the actual parameter which corresponds to the formal parameter. The best we could do is to make two copies of this part of the actual parameter, leaving the rest of the actual parameter alone. We can write Copy in ASL as follows:

Copy $=_{def}$ $\lambda X{:}\phi[\sigma].\ X$ + **derive from** X **by** ρ
  where $\rho: \Sigma_{univ} \to \Sigma_{univ}$ is defined by
    $\rho(s') = s$ for $s' \in \Lambda$
    $\rho(\omega') = \omega \in \Gamma_{s1 \ldots sn \to s}$ for $\omega' \in \Gamma_{s1' \ldots sn' \to s'}$

(this assumes that Λ and Γ are closed under 'priming': $s \in \Lambda \Longrightarrow s' \in \Lambda$ and $\omega \in \Gamma_{us} \Longrightarrow \omega' \in \Gamma_{u's'}$). Copy(Nat[$\phi$]) then has the sorts *nat* and *nat'* and operators *0: nat*, *0': nat'*, *succ: nat→ nat* and *succ': nat'→ nat'*. Note how heavily this specification relies on the definition of signature morphisms in section 2.1.

But this specification is not quite correct; suppose T contains *f: s→s* and *f': s'→s'*. Then Copy(T[ϕ]) will include the operators *f: s→s*, *f': s'→s'* and *f'': s''→s''*. In order to get two copies of each sort and operator, Copy has to take account of the signature of the actual parameter. So in fact we need ρ in Copy to be parameterised by the signature of the actual parameter:

Copy $=_{def}$ $\lambda X{:}\phi[\sigma].\ X$ + **derive from** X **by** ρ(Sig(X))
  where ρ: signature → signature morphism is defined by $\rho(\Sigma) = \rho_\Sigma$
  where $\rho_\Sigma: \Sigma_{univ} \to \Sigma_{univ}$ is in turn defined by
    $\rho_\Sigma(s\underbrace{'\cdots'}_{n+1}) = s$ where n is the maximum number of
    $\rho_\Sigma(\omega\underbrace{'\cdots'}_{n+1}) = \omega$ primes on a sort or operator in Σ

In order to define the semantics of recursive parameterised specifications we need orderings on signatures, on Σ-model classes and on signature morphisms. For these we use signature inclusion, set containment and the 'less defined' relation ($\sqsubseteq$) on partial functions respectively.

Theorem: All operators are monotonic with respect to signature inclusion and containment of model classes.

Note that the operations are also continuous for signatures and (except **reachable**) for model classes.

The monotonicity of the operations implies that every fixed-point equation for signatures or for model classes (on the same signature) has a least solution (taking the usual pointwise extension of an ordering on a set to an ordering on functions on the set). Therefore we define the semantics of recursive parameterised specifications (written $\Upsilon t(\lambda X{:}R[\sigma].B)$ where B may contain t) as the least fixed point of the equation $t = \lambda X{:}R[\sigma].B$; that is:

$Sig[\![\Upsilon t(\lambda X{:}R[\sigma].B)]\!]$ = the function (of type SEM X signature morphism → signature)
  which is the least solution of $Sig[\![t]\!](ARG,\rho) = Sig[\![(\lambda X{:}R[\sigma].B)(ARG[\rho])]\!]$

$Mod[\![\Upsilon t(\lambda X{:}R[\sigma].B)]\!]$ = the function (: SEM X signature morphism → model class)
  which is the least solution (i.e. the 'least' according to $\supseteq$, which is actually the greatest) of
  $Mod[\![t]\!](ARG,\rho) = Mod[\![(\lambda X{:}R[\sigma].B)(ARG[\rho])]\!]$

in the class of functions taking a SEM-object ARG with signature Σ and a signature morphism $\rho: Sig[\![R]\!] \to \Sigma$ and giving a $Sig[\![\Upsilon t(\lambda X{:}R[\sigma].B)]\!](ARG,\rho)$ -model class as result.

Then applying a recursive parameterised specification to an argument is just function application:

$Sig[\![\Upsilon t(\lambda X{:}R[\sigma].B)(ARG[\rho])]\!] = Sig[\![\Upsilon t(\lambda X{:}R[\sigma].B)]\!](ARG,\rho)$
$Mod[\![\Upsilon t(\lambda X{:}R[\sigma].B)(ARG[\rho])]\!] = Mod[\![\Upsilon t(\lambda X{:}R[\sigma].B)]\!](ARG,\rho)$

Generalisation to mutually recursive definitions is possible (see [Wir 82]).

Recursive parameterised specifications can be used to write 'abstract domain equations' as in [HR 80] and

[EL 81]. By monotonicity, every such equation has a solution which can be computed within a finite number of iterations (if specifications are finite and every signature morphism has finite domain).

6 Implementation of specifications

The programming discipline of stepwise refinement advocated by Wirth and Dijkstra suggests that a program be evolved by working gradually via a series of successively lower-level refinements of the specification toward a specification which is so low-level that it can be regarded as a program. This approach guarantees the correctness of the resulting program, provided that each refinement step can be proved correct. A formalisation of this approach requires a definition of the concept of refinement, i.e. of the *implementation* of one specification by another.

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 high-level specification leaves open. The following very simple formal notion of implementation captures this idea; a specification T is implemented by another specification T' if T' incorporates more design decisions than T:

Def: If T and T' are specifications, then T is *implemented* by T', written $T \rightsquigarrow T'$, if $\emptyset \neq T' \subseteq T$.

For example, suppose SetChoose specifies the standard model of sets of natural numbers (like Set in section 4) together with an operator *choose: set* $\rightarrow$ *nat* constrained only by the following axiom:

choose(add(x,S)) $\in$ add(x,S) = true

That is, *choose* will select some arbitrary element of any non-empty set. And suppose SetChoose' is SetChoose augmented by axioms which further constrain *choose* to always select the minimal element. Then $Mod[\![SetChoose']\!] \subseteq Mod[\![SetChoose]\!]$ and so SetChoose $\rightsquigarrow$ SetChoose' (since SetChoose' is satisfiable).

As another example, Behavioural-Set from section 4 (recall Behavioural-Set = **behaviour** Set **wrt {bool, nat}**, where Set specifies the standard model of sets) is implemented by List (lists of natural numbers together with the operator $\in$) once the 'auxiliary' operators *head* and *tail* have been forgotten and the sort *list* and operators *nil* and *cons* renamed as *set*, $\emptyset$ and *add*:

Behavioural-Set $\rightsquigarrow$ **derive from** List **by** σ

where σ:Sig$[\![$Behavioural-Set$]\!] \to$ sig$[\![$List$]\!]$ is a signature morphism with $\sigma(set)=list$, $\sigma(\emptyset)=nil$, $\sigma(add)=cons$ and $\sigma(x)=x$ for all other sorts and operators x in Behavioural-Set. But note:

Set $\not\rightsquigarrow$ **derive from** List **by** σ

since Set itself (before behavioural abstraction) is satisfied only by algebras isomorphic to the standard model. Under most previous notions of implementation (see below) Set $\rightsquigarrow$ **derive from** List **by** σ is a proper implementation. This was necessary because previous specification languages did not permit behavioural abstraction, so the notion of implementation had to capture it.

This notion of implementation extends to give a notion of the implementation of parameterised specifications:

Def: If $P=\lambda X{:}R[\sigma].B$ and $P'=\lambda X{:}R[\sigma].B'$ are parameterised specifications, then P is *implemented* by P', written $P \rightsquigarrow P'$, if for all actual parameters ARG$\in$SEM with fitting morphism ρ:Sig$[\![R]\!] \to$ Sig(ARG) such that $\{A|_\rho \mid A \in Mod(ARG)\} \subseteq Mod[\![R]\!]$, $P(ARG[\rho]) \rightsquigarrow P'(ARG[\rho])$.

This definition can easily be generalised to parameterised specifications with multiple parameters.

An important issue for any notion of implementation is whether implementations can be composed *vertically* and *horizontally* [GB 80]. Implementations can be vertically composed if the implementation relation is transitive ($T \rightsquigarrow T'$

and $T' \rightsquigarrow T''$ implies $T \rightsquigarrow T''$) and they can be horizontally composed if the specification-building operations preserve implementations (i.e. $P \rightsquigarrow P'$ and $A \rightsquigarrow A'$ implies $P(A) \rightsquigarrow P'(A')$; $A \rightsquigarrow A'$ and $B \rightsquigarrow B'$ implies $A+B \rightsquigarrow A'+B'$; and a similar rule holds for each of the remaining operations). Our notion of implementation has both these properties (the proofs are immediate by transitivity of $\subseteq$ and monotonicity of ASL's operations):

Theorem (vertical composition): If $T \rightsquigarrow T'$ and $T' \rightsquigarrow T''$ then $T \rightsquigarrow T''$.

Theorem (horizontal composition): If $A \rightsquigarrow A'$, $B \rightsquigarrow B'$ and $P=\lambda X{:}R[\sigma].C \rightsquigarrow P'=\lambda X{:}R[\sigma].C'$, then:

1. $A + B \rightsquigarrow A' + B'$ iff $A' + B'$ is satisfiable
2. For any $S \subseteq sorts(A)$, **reachable** A **on** S $\rightsquigarrow$ **reachable** A' **on** S iff **reachable** A' **on** S is satisfiable
3. For any $\sigma: \Sigma \rightarrow Sig[\![A]\!]$, **derive from** A **by** $\sigma \rightsquigarrow$ **derive from** A' **by** σ
4. For any $W \subseteq |W_\Sigma(X)|$, **abstract** A **wrt** W $\rightsquigarrow$ **abstract** A' **wrt** W
5. If $\rho: Sig[\![R]\!] \rightarrow Sig[\![A]\!]$ is a fitting morphism such that $\{M|_\rho \mid M \in Mod[\![A]\!]\} \subseteq Mod[\![R]\!]$ and $\{M|_\rho \mid M \in Mod[\![A']\!]\} \subseteq Mod[\![R]\!]$, then $P(A[\rho]) \rightsquigarrow P'(A'[\rho])$

These two results allow large structured specifications to be refined in a gradual and modular fashion. All of the individual small specifications which make up a large specification can be separately refined in several stages to give a collection of lower-level specifications (this is easy because of their small size). When the low-level specifications are put back together, the result is guaranteed to be an implementation of the original specification.

ASL can be used to express a number of other concepts of implementation as well, including notions which coincide with or approximate most previously proposed definitions such as [EKMP 82], [EK 82], [GM 82] and [SW 82]. Only one of these is given below; see [SW 83] for some others.

Def: If A and A' are Σ-algebras, then $A \geqslant A'$ if there exists a surjective homomorphism $f: A \rightarrow A'$. If T and T' are specifications with $Sig[\![T]\!] = Sig[\![T']\!]$, then T' is a *homomorphic image* of T ($T \geqslant T'$) if for every $A \in Mod[\![T]\!]$ there exists an $A' \in Mod[\![T']\!]$ such that $A \geqslant A'$.

Def: If T and T' are specifications, $\sigma: Sig[\![T]\!] \rightarrow Sig[\![T']\!]$ is a signature morphism, $OBS \subseteq sorts(T)$ and **reachable** T = T then $T \underset{FIR}{\overset{\sigma}{\rightsquigarrow}} T'$ if $\phi \neq$ **derive from** T' **by** $\sigma \geqslant$ **junk** T.

This corresponds to the notion of implementation in [Ehr 79], and is a simplified version of the notion in [EK 82]. Observe that Set $\underset{FIR}{\overset{\sigma}{\rightsquigarrow}}$ List where σ is an appropriate signature morphism, and note that this notion may be extended to give a notion of the implementation of parameterised specifications.

When using a powerful specification language one can adopt a simple notion of implementation. Previous languages and specification methods were less powerful (lacking operations like **behaviour**) so a more complex notion of implementation was necessary to handle cases like the implementation of sets by lists above. With ASL such complexity is not required because all such cases can be handled by explicit use of the **behaviour** operation.

One benefit of such a simple notion of implementation is that one can reason about implementations in a formal way using the specification language itself rather than at a metalevel using a metalanguage. For example, the identities in the next section can be used to prove the transitivity of $\underset{FIR}{\rightsquigarrow}$ (see [SW 83]). A second benefit is that with this simple notion the specifier has more freedom to say exactly what is required. For example, in some situations we might really want sets to be implemented only using a representation isomorphic to the standard model (e.g. in cases where the choice of data representation influences the complexity of an algorithm). In ASL one has the freedom not to apply behavioural abstraction in cases such as these. Finally, the simple notion of implementation permits vertical and horizontal composition of implementations, but this is generally not the case for the more complicated notions unless rather strong conditions are imposed (see e.g. [EKMP 82], [SW 82] and [GM 82]).

7 Identities and transformation of specifications

Because the semantics of ASL is simple, it is easy to prove that certain identities and relations between specifications hold. For example (see [SW 83] for some others):

Theorem: 1. **reachable** (**reachable** T **on** S) **on** S' = **reachable** (**reachable** T **on** S') **on** S $\supseteq$ **reachable** T **on** S ∪ S'

2. W' ⊆ W implies **abstract** (**abstract** T **wrt** W) **wrt** W' = **abstract** T **wrt** W'
= **abstract** (**abstract** T **wrt** W') **wrt** W

so a. **behaviour** (**junk** T **on** S) **wrt** OBS = **behaviour** T **wrt** OBS if S ∩ OBS = ∅
b. **junk** (**junk** T **on** S') **on** S = **junk** T **on** S = **junk** (**junk** T **on** S) **on** S' if S' ⊆ S
c. **behaviour** (**behaviour** T **wrt** OBS') **wrt** OBS = **behaviour** T **wrt** OBS if OBS ⊆ OBS'
d. **junk** (**behaviour** T **wrt** OBS) **on** S = **behaviour** T **wrt** OBS if S ∩ OBS = ∅

3. **derive from** (**derive from** T **by** σ) **by** σ' = **derive from** T **by** $\sigma' \cdot \sigma$

4. **behaviour** (**restrict** T **on** S) **wrt** OBS = **behaviour** T **wrt** OBS
if S ∩ OBS = ∅ and $W_\Sigma(X_{S'})$ is non-empty in all sorts of S, where Σ=Sig⟦T⟧, S'=sorts(Σ)-S

5. **abstract** (**derive from** T **by** σ) **wrt** W $\supseteq$ **derive from** (**abstract** T **wrt** σ(W)) **by** σ

6. **abstract** (**abstract** T **wrt** W) **wrt** W' ⊆ **abstract** T **wrt** W ∩ W'
so a. **junk** (**junk** T **on** S) **on** S' ⊆ **junk** T **on** S ∪ S'
b. **behaviour** (**behaviour** T **wrt** OBS) **wrt** OBS' ⊆ **behaviour** T **wrt** OBS ∩ OBS'

7. **reachable** (T + **reachable** T' **on** S) **on** S' = T + **reachable** T' **on** S if S' ⊆ S

8. **reachable** (**enrich** T **by axioms** E) **on** S = **enrich reachable** T **on** S **by axioms** E

9. T ⩾ T' implies **junk** T **on** S ⩾ **junk** T' **on** S

10. T ⩾ T' implies **derive from** T **by** σ ⩾ **derive from** T' **by** σ

11. $\Upsilon t.(\lambda X{:}R[\sigma].B)(ARG[\rho]) = B_\rho[ARG/X][\Upsilon t.(\lambda X{:}R[\sigma].B)/t]$
if $[A|_\rho \mid A \in Mod(ARG)] \subseteq Mod[\![R]\!]$

But it is possible to find counterexamples showing that the following inequations hold:

Fact: 1'. **abstract** (**derive from** T **by** σ) **wrt** W ≠ **derive from** (**abstract** T **wrt** σ(W)) **by** σ

2'. **reachable** (**reachable** T **on** S) **on** S' ≠ **reachable** T **on** S ∪ S'

3'. **abstract** (**abstract** T **wrt** W) **wrt** W' ≠ **abstract** T **wrt** W ∩ W'

4'. **behaviour** (**behaviour** T **wrt** OBS) **wrt** OBS' ≠ **behaviour** (**behaviour** T **wrt** OBS') **wrt** OBS
≠ **behaviour** T **wrt** OBS ∪ OBS'

These properties can be useful for understanding the effects of ASL's operations. For example, properties 2a and 4 together indicate that the **behaviour** operation disregards any junk of invisible sorts.

It is possible to carry out proofs concerning specifications using the above properties. For example, solutions of domain equations can be computed. The following theorem can be proved in this way.

Theorem (vertical composition for $\underset{FIR}{\leadsto}$): $T \underset{FIR}{\overset{\sigma}{\leadsto}} T'$ and $T' \underset{FIR}{\overset{\sigma'}{\leadsto}} T''$ implies $T \underset{FIR}{\overset{\sigma \cdot \sigma'}{\leadsto}} T''$.

These rules provide transformations for changing one specification into another specification which is equivalent (or an implementation, using the rules containing ⊆). Therefore they could be used as the basis of a method for developing data structures and programs from specifications (see e.g. [Bau 81a]).

8 Concluding remarks

The small set of operations in ASL seems to provide a powerful means for writing specifications. But some of the operations we decided not to include are interesting as well. Here are two which together could replace **abstract**:

Def: If A, A' are Σ-algebras and $W \subseteq |W_\Sigma(X)|$ then A is *W-finer* then A' ($A \leqslant_W A'$) if there are surjective assignments $\phi: X \to |A|$ and $\phi': X \to |A'|$ such that $\forall t, t' \in W.\ (\phi^\#(t) = \phi^\#(t') \Longrightarrow \phi'^\#(t) = \phi'^\#(t'))$.

The *W-coarser* relation $\geqslant_W$ is obtained by replacing $\Longrightarrow$ in this definition by $\Longleftarrow$.

$$\begin{array}{ll}
\mathrm{Sig}[\![T\Delta W]\!] = \mathrm{Sig}[\![T\nabla W]\!] = \mathrm{Sig}[\![T]\!] & \text{if } W \subseteq |W_\Sigma(X)| \text{ where } \Sigma = \mathrm{Sig}[\![T]\!] \\
\mathrm{Mod}[\![T\Delta W]\!] = [\ A \in \mathrm{Alg}(\mathrm{Sig}[\![T]\!])\ |\ \exists A_0 \in \mathrm{Mod}[\![T]\!].\ (A \leqslant_W A_0)\ ] & \\
\mathrm{Mod}[\![T\nabla W]\!] = [\ A \in \mathrm{Alg}(\mathrm{Sig}[\![T]\!])\ |\ \exists A_0 \in \mathrm{Mod}[\![T]\!].\ (A \geqslant_W A_0)\ ] &
\end{array}$$

Then **abstract** T **wrt** W $=_{def}$ $T\Delta W + T\nabla W$
hom T **wrt** S $=_{def}$ **behaviour** T **wrt** S + $T\nabla|W_\Sigma(X)|$ where $\Sigma = \mathrm{Sig}[\![T]\!]$
T/eqns $=_{def}$ $\langle \mathrm{Sig}[\![T]\!], \mathrm{eqns}\rangle + T\nabla|W_\Sigma(X)|$ where $\Sigma = \mathrm{Sig}[\![T]\!]$

The **hom** operation is the same as behavioural abstraction except that it only permits models which are coarser than models of T (i.e. in which more terms are identified). An operation permitting only finer models can be defined similarly. T/E is the quotient of T by the equations E as defined in [Wir 82] (not exactly the usual quotient, since everything coarser than the quotient is included as well). Other interesting possibilities are:

$$\begin{array}{ll}
\mathrm{Sig}[\![T \cup T']\!] = \mathrm{Sig}[\![T \cap T']\!] = \mathrm{Sig}[\![T]\!] & \text{if } \mathrm{Sig}[\![T]\!] = \mathrm{Sig}[\![T']\!] \\
\mathrm{Mod}[\![T \cup T']\!] = [\ A \in \mathrm{Alg}(\mathrm{Sig}[\![T]\!])\ |\ \exists A_0 \in \mathrm{Mod}[\![T]\!], A'_0 \in \mathrm{Mod}[\![T']\!].\ (A \text{ is the glb of } A_0 \text{ and } A'_0)\ ] & \\
\mathrm{Mod}[\![T \cap T']\!] = [\ A \in \mathrm{Alg}(\mathrm{Sig}[\![T]\!])\ |\ \exists A_0 \in \mathrm{Mod}[\![T]\!], A'_0 \in \mathrm{Mod}[\![T']\!].\ (A \text{ is the lub of } A_0 \text{ and } A'_0)\ ] &
\end{array}$$

The lub (least upper bound) and glb (greatest lower bound) are with respect to the homomorphic image relation $\geqslant$ defined in section 6. Note that the lub and glb are not defined uniquely but only up to isomorphism.

Then $T\Delta$ $=_{def}$ $\langle \mathrm{Sig}[\![T]\!], \phi\rangle \cap T$ (i.e. $T\Delta|W_\Sigma(X)|$)
$T\nabla$ $=_{def}$ $\langle \mathrm{Sig}[\![T]\!], \phi\rangle \cup T$ (i.e. $T\nabla|W_\Sigma(X)|$)
hom T **wrt** S $=_{def}$ **behaviour** T **wrt** S + $T\nabla$
T/eqns $=_{def}$ $\langle \mathrm{Sig}[\![T]\!], \mathrm{eqns}\rangle + T\nabla$

Parameterised specifications with signature morphisms as parameters are a special case of parameterised specifications as defined here. This allows the expression of e.g. Clear-style procedure application with avoidance of name clashes. But signature morphisms are not yet 'first class citizens'; it is not possible to specify 'requirements' for signature morphism parameters. For example, it should be possible to require that a signature morphism be defined at least (or at most) on a particular domain, or that it extends a given signature morphism. This should be a straightforward extension. Another interesting generalisation would be to allow (recursive) higher-order parameterised specifications.

Inference in ASL specifications is more complex than in a 'flat' equational specification or in an ordinary structured theory as in LCF [GMW 79] or Clear. Besides the usual inference rules which allow theorems to be derived by combining axioms, inference rules are needed which allow theorems in a specification (say T) to be converted to theorems in a larger specification built from T (say T + T') as in [SB 83]. For example:

thm **in** T $\Longrightarrow$ thm **in** T + T'
σ(thm) **in** T $\Longrightarrow$ thm **in derive from** T **by** σ
thm **in** T and $\forall t \in \mathrm{terms(thm)}.[t \in W \text{ and } \forall x \in FV(t)_s . \forall y \in X_s . t[y/x] \in W]$ $\Longrightarrow$ thm **in abstract** T **wrt** W

The last of these implies the following rule:

thm **in** T and thm contains only terms of sorts in OBS with variables of OBS sorts
$\Longrightarrow$ thm **in behaviour** T **wrt** OBS

The **reachable** operation gives rise to an induction principle.

Finally, it would be interesting to build a new high-level specification language on top of ASL, trying to make available most of the power of ASL (e.g. behavioural abstraction) in higher-level specification-building operations while hiding some of the sharp edges (e.g. it probably should not be possible to get the effect of **abstract** T **wrt** W for arbitrary W). The result should be more versatile and expressive than any present specification language.

To avoid confusion, it is important to point out the differences between the present paper and [Wir 82] which also defined a language called ASL (we will refer to the two languages as new ASL and old ASL respectively). Apart from details of syntax, the differences between the two languages are as follows:

- New ASL contains an important new operation (**abstract**) which allows the expression of behavioural abstraction. Old ASL includes a 'quotient' operation which is not provided in new ASL. This change gives a language which is more oriented toward a behavioural approach to specification. The quotient operation was difficult to use in writing specifications [Gau 83] and did not easily extend from equational axioms to general first-order axioms.
- New ASL includes a more general and flexible parameterisation mechanism than old ASL.
- Old ASL is a language for specifying partial algebras, while new ASL (as described here) is for specifying total algebras. There is no difficulty in changing new ASL to specify partial algebras; we restricted attention to total algebras only for simplicity of presentation.

Furthermore, the present paper develops and justifies an elegant and simple notion of implementation of ASL specifications. This notion was mentioned briefly in [Wir 82], but here it is more appropriate because new ASL can express behavioural abstraction.

Acknowledgements

Thanks for useful discussions and helpful comments: from DS to Rocco de Nicola, David Rydeheard, Oliver Schoett and (especially) Rod Burstall; and from MW to Manfred Broy and Marie-Claude Gaudel. This work was supported by the Science and Engineering Research Council and the Sonderforschungsbereich 49, Programmiertechnik, München.

9 References

Note: LNCS n denotes Springer Lecture Notes in Computer Science, Vol. n

[ADJ 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 RC6487. Also in: Current Trends in Programming Methodology, Vol. 4: Data Structuring (R.T. Yeh, ed.), Prentice-Hall, pp. 80-149 (1978).

[ADJ 78] Thatcher, J.W., Wagner, E.G. and Wright, J.B. Data type specification: parameterization and the power of specification techniques. SIGACT 10th Annual Symp. on the Theory of Computing, San Diego, California.

[ADJ 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 ICALP, Noordwijkerhout, Netherlands. LNCS 85, pp. 157-168.

[Bau 81] Bauer, F.L. *et al* (the CIP Language Group) Report on a wide spectrum language for program specification and development (tentative version). Report TUM-I8104, Technische Univ. München.

[Bau 81a] 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.

[BBTW 81] Bergstra, J.A., Broy, M., Tucker, J.V. and Wirsing, M. On the power of algebraic specifications. Proc. 10th MFCS, Strbske Pleso, Czechoslovakia. LNCS 118, pp. 193-204.

[BG 77] Burstall, R.M. and Goguen, J.A. Putting theories together to make specifications. Proc. 5th IJCAI, Cambridge, Massachusetts, pp. 1045-1058.

[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. LNCS 86, pp. 292-332.

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

[Ehr 79] Ehrich, H.-D. On the theory of specification, implementation, and parametrization of abstract data types. Report 82, Univ. of Dortmund. Also in: JACM 29, 1 pp. 206-227 (1982).

[EK 82] Ehrig, H. and Kreowski, H.-J. (1982) Parameter passing commutes with implementation of parameterized data types. Proc. 9th ICALP, Aarhus, Denmark. LNCS 140, pp. 197-211.

[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.

[EL 81] Ehrich, H.-D. and Lipeck, U. Algebraic domain equations. Report 125, Univ. of Dortmund.

[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.

[Gau 83] Gaudel, M.-C. Personal communication with M. Wirsing.

[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, Computer Science Laboratory, SRI International.

[GB 83] Goguen, J.A. and Burstall, R.M. Institutions: logic and specification. Draft report, SRI International.

[GGM 76] Giarratana, V., Gimona, F. and Montanari, U. Observability concepts in abstract data type specification. Proc. 5th MFCS, Gdansk. LNCS 45, pp. 576-587.

[GM 82] Goguen, J.A. and Meseguer, J. Universal realization, persistent interconnection and implementation of abstract modules. Proc. 9th ICALP, Aarhus, Denmark. LNCS 140, pp. 310-323.

[GM 83] Goguen, J.A. and Meseguer, J. An initiality primer. Draft report, SRI International.

[GMW 79] Gordon, M.J., Milner, A.J.R. and Wadsworth, C.P. Edinburgh LCF. LNCS 78.

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

[HKR 80] Hupbach, U.L., Kaphengst, H. and Reichel, H. Initial algebraic specification of data types, parameterized data types, and algorithms. VEB Robotron, Zentrum für Forschung und Technik, Dresden.

[HR 80] Hornung, G. and Raulefs, P. Terminal algebra semantics and retractions for abstract data types. Proc. 7th ICALP, Noordwijkerhout, Netherlands. LNCS 85, pp. 310-323.

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

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

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

[SB 83] Sannella, D.T. and Burstall, R.M. Structured theories in LCF. Proc. 8th CAAP, L'Aquila, Italy. LNCS, to appear.

[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 ICALP, Aarhus, Denmark. LNCS 140, pp. 473-488.

[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.

[Sch 82] Schoett, O. A theory of program modules, their specification and implementation. Draft report, Univ. of Edinburgh.

[Wand 79] Wand, M. Final algebra semantics and data type extensions. JCSS 19 pp. 27-44.

[Wir 82] Wirsing, M. Structured algebraic specifications. Proc. AFCET Symp. on Mathematics for Computer Science, Paris, pp. 93-107.

[ZLT 82] Zilles, S.N., Lucas, P. and Thatcher, J.W. A look at algebraic specifications. Draft report, IBM research.