IMPLEMENTATION OF PARAMETERISED SPECIFICATIONS

-- Extended Abstract* --

Donald Sannella
Department of Computer Science
University of Edinburgh

Martin Wirsing
Institut für Informatik
Technische Universität München

Abstract

A new notion is given for the implementation of one specification by another. Unlike most previous notions, this generalises to handle parameterised specifications as well as loose specifications (having an assortment of non-isomorphic models). Examples are given to illustrate the notion. The definition of implementation is based on a new notion of the simulation of a theory by an algebra. For the bulk of the paper we employ a variant of the Clear specification language [BG 77] in which the notion of a data constraint is replaced by the weaker notion of a hierarchy constraint. All results hold for Clear with data constraints as well, but only under more restrictive conditions.

We prove that implementations compose vertically (two successive implementation steps compose to give one large step) and that they compose horizontally under application of (well-behaved) parameterised specifications (separate implementations of the parameterised specification and the actual parameter compose to give an implementation of the application).

1. Introduction

Algebraic specifications can be viewed as abstract programs. Some specifications are so completely abstract that they give no hint of a method for finding an answer. For example, the function inv:matrix->matrix for inverting an nxn matrix can be specified as follows:

```
inv(A) x A = I
A x inv(A) = I
```

(provided that matrix multiplication and the identity nxn matrix have already been specified). Other specifications are so concrete that they resemble programs. For example:

```
reverse(nil) = nil
reverse(cons(a,l)) = append(reverse(l),cons(a,nil))
```

(this specification of the reverse function on lists amounts to an executable program in the HOPE functional programming language [BMS 80]).

It is usually easiest to specify a problem at a relatively abstract level. We can then work gradually and systematically toward a low-level 'program' which satisfies the specification. This will normally involve the introduction of auxiliary functions, particular data representations and so on. This approach to program development is related to the well-known programming discipline of stepwise refinement advocated by Wirth [Wir 71] and Dijkstra [Dij 72].

---

*The full version of this paper is available as Report CSR-103-82, Department of Computer Science, University of Edinburgh.

A formalisation of this programming methodology depends on some precise notion of the implementation of a specification by a lower-level specification. Previous notions have been given for the implementation of non-parameterised ([GTW 78], [Nou 79], [Hup 80], [EKP 80], [Ehr 82]) and parameterised ([Gan 81], [Hup 81])** specifications, but none of these approaches deals fully with 'structured' algebraic specifications (as in Clear [BG 77] or CIP-L [Bau 81]) which may be constructed in a hierarchical fashion and may be loose (with an assortment of non-isomorphic models). We present a definition of implementation which agrees with our intuitive notions built upon programming experience and which handles such loose hierarchical specifications, based on a new (and seemingly fundamental) concept of the simulation of a theory by an algebra. We show how this definition extends to give a definition of the implementation of parameterised specifications. An example of an implementation is given and several other examples are sketched.

We work within the framework of the Clear specification language [BG 77] which allows large specifications to be built from small easy-to-understand bits. For most of the paper we employ a variant of Clear in which the notion of a data constraint is replaced by the weaker notion of a hierarchy constraint. The result is still a viable specification language, although specifications tend to be somewhat longer than in ordinary Clear. We later show that all results hold for 'ordinary' Clear (with data constraints), but only under more restrictive conditions.

The 'putting-together' theme of Clear and the ideas incorporated in CAT [GB 80] (a proposed system for systematic program development using Clear) lead us to wonder if implementations can be put together as well. We prove that if P is implemented by P' (where P and P' are 'well-behaved' parameterised specifications) and A is implemented by A', then P(A) is implemented by P'(A').

We prove that implementations compose in another dimension as well. If a high-level specification A is implemented by a lower-level specification B which is in turn implemented by a still lower-level specification C (and an extra compatibility condition is satisfied), then A is implemented by C. These two results allow large specifications to be refined in a gradual and modular fashion, a little bit at a time.

2. Preliminaries -- Clear and data/hierarchy constraints

This section is devoted to a very brief review of the notions underlying Clear along with a discussion of data and hierarchy constraints. See [BG 77] for an informal description of Clear, [San 81] and [BG 80] for its formal semantics, and [WB 81] for more about hierarchy constraints. For the usual notions of signature Σ, signature morphism (inclusion) $\sigma=\langle f,g\rangle$ (f maps sorts, g maps operators), Σ-algebra $A=\langle A,\alpha\rangle$ (A_s is the carrier for sort s, $\alpha(\omega)$ is the function associated with operator ω), homomorphism, Σ-equation, and satisfaction of a set of Σ-equations by a Σ-algebra see [BG 80]. The following less conventional definitions are taken (with minor changes) from the same source.

Def: If $\sigma=\langle f,g\rangle$ is a signature morphism $\sigma:\Sigma\rightarrow\Sigma'$ and $A'=\langle A',\alpha'\rangle$ is a Σ'-algebra, then the Σ-restriction of A' (along σ), written $A'|_{\Sigma}^{\sigma}$ is the Σ-algebra $\langle A,\alpha\rangle$ where $A_s=A'_{f(s)}$ and $\alpha(\omega)=\alpha'(g(\omega))$. Normally σ is obvious from context, in which case the notation $A'|_{\Sigma}$ may be used.

Def: A simple Σ-theory presentation is a pair $\langle\Sigma,E\rangle$ where Σ is a signature and E is a set of Σ-equations. This is simple because no constraints (see below) are included.

Def: A Σ-algebra A satisfies a simple theory presentation $\langle\Sigma,E\rangle$ if A satisfies E. Then A is called a model of $\langle\Sigma,E\rangle$. A theory presentation T specifies a set of

**Also [EK 82] which we discovered while writing the final version.

algebras, namely the set of its models denoted Models(T). A theory is called satisfiable if it has at least one model.

Def: If E is a set of Σ-equations, let E^* be the set of all Σ-algebras which satisfy E. If M is a set of Σ-algebras, let M^* be the set of all Σ-equations which are satisfied by each algebra in M. The closure of a set E of Σ-equations is the set E^{**}, written $\bar{E}$. E is closed if $E=\bar{E}$.

Def: A simple Σ-theory T is a simple theory presentation $\langle\Sigma,E\rangle$ where E is closed. The simple Σ-theory presented by the presentation $\langle\Sigma,E\rangle$ is $\langle\Sigma,\bar{E}\rangle$.

Def: A simple theory morphism (inclusion) $\sigma : \langle\Sigma,E\rangle \longrightarrow \langle\Sigma',E'\rangle$, where $\langle\Sigma,E\rangle$ and $\langle\Sigma',E'\rangle$ are simple theories, is a signature morphism (inclusion) $\sigma:\Sigma\longrightarrow\Sigma'$ such that $\sigma(e)\in E'$ for each $e\in E$.

Note that any algebra satisfying a simple theory (presentation) is a model of that simple theory. This is in contrast to the 'initial algebra approach' of ADJ [GTW 78] in which only the initial algebra of a theory is accepted as a model. Clear permits us to write loose specifications such as the following:

```
const ApproxSqrt = enrich Nat by
                     opns sqrt : nat -> nat
                     eqns x - (sqrt(x))^2 ≤ x/2 + 1  = true    enden
```

This specifies an approximate square-root function on the natural numbers. Any function with at least the specified accuracy will do (for example, sqrt(100) may be 7, 8, 9 or 10). Under the initial algebra approach such a specification yields a single model with extra values of sort nat.

Even in Clear, we often want to restrict the class of models. For instance, if no restriction is present in the above example then trivial models (where all natural numbers are equal to 0) and models with extra values of sort nat (other than $succ^n(0)$ for $n>0$) are allowed. This facility is provided by Clear's data operation, which may be applied when enriching a theory by some new sorts, operators and equations. Each application of the data operation contributes a data constraint to the theory which results from the enrichment. Unfortunately, implementations (section 3) do not seem to have nice properties in the presence of data constraints. Accordingly we use for the bulk of this paper a variant of Clear in which data constraints are replaced by hierarchy constraints, contributed by the 'data' operation (see [BDPPW 79] and [WB 81]). Hierarchy constraints are weaker than data constraints so specifications tend to be somewhat longer than in ordinary Clear (as in the terminal algebra approach, it is sometimes necessary to add extra operators to avoid trivial models). We show later that all results hold for Clear with data constraints, but only under more restrictive conditions. We now give some definitions concerning data and hierarchy constraints; note that in most respects the two kinds of constraints are identical.

Def: A Σ-data (hierarchy) constraint c is a pair $\langle i,\sigma\rangle$ where $i:T\hookrightarrow T'$ is a simple theory inclusion and $\sigma:\text{signature}(T')\longrightarrow\Sigma$ is a signature morphism.

A data (hierarchy) constraint is a description of an enrichment (the theory inclusion goes from the theory to be enriched to the enriched theory) together with a signature morphism 'translating' the constraint to the signature Σ.

A signature morphism from Σ to another signature Σ' can be applied to a Σ-constraint, translating it to a Σ'-constraint, just as it can be applied to a Σ-equation to give a Σ'-equation.

Def: If $\sigma':\Sigma\longrightarrow\Sigma'$ is a signature morphism and $\langle i,\sigma\rangle$ is a Σ-data (hierarchy) constraint, then σ' applied to $\langle i,\sigma\rangle$ gives the Σ'-data (hierarchy) constraint $\langle i,\sigma.\sigma'\rangle$.

A data (hierarchy) constraint imposes a restriction on a set of Σ-algebras, just as an equation does. The only difference between a data constraint and a hierarchy

constraint is in this restriction; compare the "no confusion" and "no crime" conditions in the following definitions.

Def: A Σ-algebra A satisfies a Σ-data constraint $\langle i:T \hookleftarrow T', \sigma:sig(T') \to \Sigma \rangle$ if (letting $A_{target} = A|^{\sigma}_{sig(T')}$ and $A_{source} = A|^{i.\sigma}_{sig(T)}$) A_{target} is a model of T' and:

- "No confusion": A_{target} does not satisfy any $sig(T')$-equation e with variables only in sorts of T for any injective assignment of variables to A_{source} values unless e is in $\overline{eqns(T') \cup A_{source}}^{*}$.
- "No junk": Every element in A_{target} is the value of a T'-term which has variables only in sorts of T, for some assignment of A_{source} values.

Without loss of generality we assume that every theory contains the theory Bool (with sort bool and constants true and false) as a primitive subtheory.

Def: A Σ-algebra A satisfies a Σ-hierarchy constraint $\langle i:T \hookleftarrow T', \sigma:sig(T') \to \Sigma \rangle$ if (with A_{target} and A_{source} as above) A_{target} is a model of T' and:

- "No crime": $A \models true \neq false$.
- "No junk": As above.

Since data (hierarchy) constraints behave just like equations, they can be added to the equation set in a simple theory presentation to give a data (hierarchical) theory presentation.

Def: A data (hierarchical) Σ-theory presentation is a pair $\langle \Sigma, EC \rangle$ where Σ is a signature and EC is a set of Σ-equations and Σ-data (hierarchy) constraints.

The notions of data (hierarchical) theory, satisfaction (of a data or hierarchical theory), closure and data (hierarchical) theory morphism follow as in the 'simple' case. The denotation of a (hierarchical) Clear specification is a data (hierarchical) theory $\langle \Sigma, EC \rangle$, specifying all Σ-algebras which satisfy the equations and data (hierarchy) constraints in EC. For the remainder of the paper (except where noted at the end of section 5) all discussion will concern only hierarchical Clear. We will use terms like 'theory' in place of longer terms like 'hierarchical theory'.

Def: A sort or operator of a theory is called constrained if it is in $\sigma(sig(T') - sig(T))$ for some constraint $\langle i:T \hookleftarrow T', \sigma:sig(T') \to \Sigma \rangle$ in that theory.

Lack of space permits only a single brief example to illustrate data and hierarchy constraints. Consider the following specification:

```
const Triv = enrich Bool by
               sorts element   enden

const T = enrich Triv by
            data sorts newelem
                 opns f : element -> newelem   enden
```

T includes a data constraint $\langle Triv \hookleftarrow T, id \rangle$. Given a sig(T)-algebra, we can check if it satisfies this constraint. For example:

$A_{element} = \{0,1,2\}$ $A_{newelem} = \{a,b,c\}$ $f(0)=a$ $f(1)=b$ $f(2)=a$

(with the usual interpretation of Bool). This fails to satisfy the "no confusion" condition (consider the equation $f(x)=f(y)$ under the injective assignment $[x \mapsto 0, y \mapsto 2]$). It also violates the "no junk" condition (the element c is not the value of any term). But if the function f is altered so that $f(2)=c$ then the constraint is satisfied. In general, any algebra satisfying this data constraint will have both

carriers of the same cardinality with f 1-1 and onto.

Changing data above to 'data' changes the data constraint to a hierarchy constraint. The following algebra is then a model of T, although it does not satisfy the "no confusion" condition:

$A_{element} = \{0,1,2\}$ $A_{newelem} = \{a\}$ $f(0) = f(1) = f(2) = a$

(again with the usual interpretation of Bool). It is necessary to add some new operators (e.g. ==:element,element->bool and ==:newelem,newelem->bool) and equations (e.g. f(x)==f(y) = x==y) to retain the original class of models.

The data constraints described here are a special case of those discussed in [BG 80], where the theory inclusion is replaced by an arbitrary theory morphism; essentially the same concept is described by Reichel [Rei 80] (cf. [KR 71]). General data constraints never actually arise in Clear. The definition of data constraint satisfaction given above is an attempt to capture, in this special case, the definition of [BG 80] using a different approach.

A consequence of the inclusion of data or hierarchy constraints in Clear theories is that no complete proof system can exist for Clear (see [MS 82]). This means that the model-theoretic closure of a set of equations and constraints (as defined above) is not always the same as its proof-theoretic closure.

For later results we need a generalisation of Guttag's notion of sufficient completeness [GH 78] and of the classical notion of conservativeness from logic:

Def: A theory T is sufficiently complete with respect to a set of operators Σ, sorts S, a subset Σ' of Σ, and variables of sorts X (where $S,X \subseteq sorts(T)$, $\Sigma \subseteq opns(T)$) if for every term t of an S sort containing operators of Σ and variables of X sorts, there exists a term t' with variables of X sorts and operators of Σ' such that $T \vdash t=t'$.

Def: A theory T is conservative with respect to a theory $T' \subseteq T$ if for all equations e containing operators only of T', $T \vdash e \Rightarrow T' \vdash e$.

Sufficient completeness means that T does not contain any new term of an old sort which is not provably equal to an old term (where 'new' and 'old' depend on Σ, S, Σ' and X). Conservativeness means that old terms (from T') are not newly identified in T. Instances of these general notions guarantee that all models of a theory possess a convenient hierarchical structure.

3. A notion of implementation

A formal approach to stepwise refinement of specifications must begin with some notion of the implementation of a specification by another (lower level) specification. Armed with a precise definition of this notion, we can prove the correctness of refinement steps, providing a basis for a methodology for the systematic development of programs which are guaranteed to satisfy their specifications. But first we must be certain that the definition itself is sound and agrees with our intuitive notions built upon programming experience.

Suppose we are given two theories $T=\langle\Sigma,EC\rangle$ and $T'=\langle\Sigma',EC'\rangle$. We want to implement the theory T (the abstract specification) using the sorts and operators provided by T' (the concrete specification). Previous formal approaches (see [GTW 78], [Nou 79], [Hup 80], [EKP 80], [Gan 81], [Ehr 82]) agree that T' implements T if there is some way of deriving sorts and operators like those of T from the sorts and operators of T'. Each approach considers a different way of making the 'bridge' from T' to T. We will require that there be a more or less direct correspondence between the sorts and operators of T and those of T'. Each sort or operator in Σ must be implemented by a sort or operator in Σ' -- this correspondence will be embodied by a signature morphism $\sigma:\Sigma \rightarrow \Sigma'$. Note that two different sorts or operators in Σ may map to the same Σ' sort or operator, and also that there may be some (auxiliary) sorts and operators in Σ' which remain unused. This is a simplification over previous approaches, which generally allow some kind of restricted enrichment of T' to T''

before matching T with T". But the power is the same; we would say that T" implements T and leave the enrichment from T' to T" to the user. As a consequence of a later theorem (see section 5) our results extend to more complex notions.

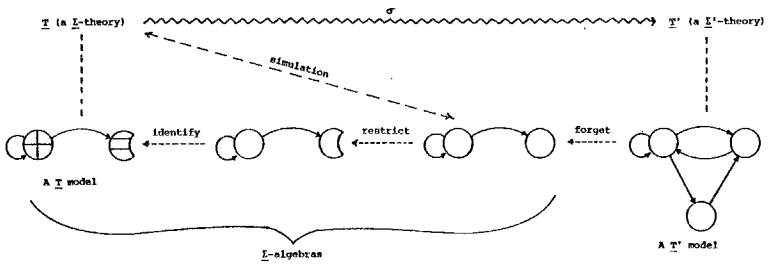
Given a signature morphism $\sigma:\Sigma\rightarrow\Sigma'$, what relationship must hold between T and T' before we can say that T' implements T? One might suspect that $\sigma(EC)\subseteq EC'$ is required -- i.e. that for any model A' of T', $A'|_{\Sigma}^{\sigma} \in$ Models(T) -- but this condition is too strong. Roughly speaking, T' implements T if the Σ-restriction of each model of T' simulates T. A (Σ-restricted) T' model need not be a model of T, but need only have the appearance of a model of T. In particular, values of the T' model must 'represent' the values of the corresponding Σ-terms, but the carriers of the T' model need not match the carriers of any T model. Some flexibility is necessary in order to approximate our intuitive notion of an implementation:

- A subset of the values of a T' sort may be used to represent all the values of a T sort. Example: implementing the natural numbers using the integers -- the negative integers are not needed.

- More than one T' value may be used to represent the same T value. Example: implementing sets by strings -- the order does not matter, so "1.2.3" = "3.2.1" (as sets).

Now T' implements T if (and only if) the Σ-restriction of any model of T' is a model of T after these two considerations have been taken into account. This ensures that corresponding operators will yield the same result (modulo data representation) which is intuitively the decisive criterion for a correct implementation.

Our definition of implementation will proceed in two steps. First we define the simulation of a theory by an algebra, making precise the vague and informal ideas outlined above. The notion of simulation is then used to give a simple definition of implementation. We chose to highlight the notion of simulation because it seems to be a fundamental concept which may be useful outside the present context. The following diagram shows how the definitions given below fit together to give notions of simulation and implementation:



An implementation $T\overset{\sigma}{\rightsquigarrow}T'$

For the definition of simulation we need an auxiliary notion. As mentioned above, a subset of the available 'concrete' values may be used to represent all 'abstract' values. Restricting the carriers of the concrete algebra to the values which are actually used yields an intermediate algebra which plays an important role in the definition of simulation. We do not want to restrict the carrier for every sort, but only for those sorts of Σ which are constrained by hierarchy constraints in T (for unconstrained sorts we do not know which values are unused). This is where we depart from the usual practice of restricting to 'reachable' values (see for example [EKP 80]). We want the subalgebra which has been reduced just enough to satisfy the "no junk" condition for each of these constraints.

Def: If Σ is a signature, A is a Σ-algebra and T is a Σ-theory, then $\text{restrict}_T(A)$ is the largest subalgebra A' of A satisfying the "no junk" condition (section 2) for every hierarchy constraint $\langle i:T'\hookleftarrow T'',\sigma:sig(T'')\rightarrow\Sigma\rangle$ in T.

Note that the subalgebra A' does not always exist. Consider the following example:

```
const T = let Nat = enrich Bool by
                      'data' sorts nat
                             opns 0 : nat
                                  succ : nat -> nat    enden in
          enrich Nat by
            opns neg : nat    enden
```

Let Σ be the signature of T. Suppose A is the Σ-algebra with carrier {-1,0,1,...}, the usual interpretation for the operators 0 and succ, and neg=-1. Now $restrict_T(A)$ does not exist because every subalgebra of A must contain -1 (the value of neg) and hence fails to satisfy the "no junk" condition for the constraint of T.

A Σ-algebra A simulates a Σ-theory T if it satisfies the equations and constraints of T after allowing for unused carrier elements and multiple representations.

Def: If Σ is a signature, A is a Σ-algebra and $T=\langle\Sigma,EC\rangle$ is a Σ-theory, then A simulates T if $restrict_T(A)/\equiv_{EC}$ (call this RI(A)) exists and is a model of T.

[$\equiv_{EC}$ is the Σ-congruence generated by EC -- i.e. the least Σ-congruence on $restrict_T(A)$ containing the relation determined by the equations in EC]

RI stands for restrict-identify, the composite operation which forms the heart of this definition. To determine if a Σ-algebra A simulates a Σ-theory T, we restrict A, removing those elements from the carrier which are not used to represent the value of any Σ-term, for constrained sorts; the result of this satisfies the "no junk" condition for each constraint in T. We then identify multiple concrete representations of the same abstract value by quotienting the result by the Σ-congruence generated by the equations of T, obtaining an algebra which (of course) satisfies those equations and also continues to satisfy the "no junk" condition of the constraints. If this is a model of T (i.e. it satisfies the "no crime" condition for each constraint in T) then A simulates T. Note that any model of T simulates T. It has been shown in [EKP 80] that the order restrict-identify gives greater generality than identify-restrict.

Most work on algebraic specifications concentrates on the specification of abstract data types, following the lead of ADJ [GTW 78] and Guttag et al [GH 78]. As pointed out by ADJ, the initial model (in the category of all models of a theory) best captures the meaning of "abstract" as used in the term "abstract data type", so other models are generally ignored (there is some disagreement on this point -- other authors prefer e.g. final models [Wan 79] -- but in any case some particular model is singled out for special attention). This is not the case in Clear (the ordinary version or our variant); although the 'data' operation may be used to restrict the set of models as discussed in section 2, no particular model is singled out so in general a theory may have many nonisomorphic models (as in the Munich approach). Such a loose theory need not be implemented by a theory with the same broad range of models. A loose theory leaves certain details unspecified and an implementation may choose among the possibilities or not as is convenient. That is:

- A loose theory may be implemented by a 'tighter' theory. Example: implementing the operator choose:set->integer (choose an element from a set) by an operator which chooses the smallest.

This is intuitively necessary because it would be silly to require that a program (the final result of the refinement process) embody all the vagueness of its original specification. This kind of flexibility is already taken into account by the discussion above, and is an important feature of our notion of implementation. Previous notions do not allow for it because they concentrate on implementation of abstract data types and so consider only a single model for any specification.

Now we are finally prepared to define our notion of the implementation of one theory by another. This definition is inspired by the notion of [EKP 80] but it is not the same; they allow a more elaborate 'bridge' but otherwise their notion is more restrictive than ours. Our notion is even closer to the one of Broy et al [BMPW 80]

but there the 'bridge' is less elaborate than ours. It also bears some resemblance to a more programming-oriented notion due to Schoett [Sch 81].

Def: If $T=\langle\Sigma,EC\rangle$ and $T'=\langle\Sigma',EC'\rangle$ are satisfiable theories and $\sigma:\Sigma\to\Sigma'$ is a signature morphism, then T' implements T (via σ), written $T\overset{\sigma}{\rightsquigarrow}T'$, if for any model A' of T', $A'|_\Sigma^\sigma$ simulates T.

Note that any theory morphism $\sigma:T\to T'$ where T' is satisfiable is an implementation $T\overset{\sigma}{\rightsquigarrow}T'$. In particular, if T' is an enrichment of T (e.g. by equations which 'tighten' a loose theory) then $T\rightsquigarrow T'$.

A simple example will show how this definition works (other implementation examples are given in the next section). Consider the theory of the natural numbers modulo 2, specified as follows:

```
const Natmod2 = enrich Bool by
                  'data' sorts natmod2
                        opns 0, 1 : natmod2
                             succ : natmod2 -> natmod2
                             iszero : natmod2 -> bool
                        eqns succ(0) = 1          succ(1) = 0
                             iszero(0) = true     iszero(1) = false    enden
```

Can this be implemented by the following theory?

```
const Fourvalues = enrich Bool by
                     'data' sorts fourvals
                           opns zero, one, zero', extra : fourvals
                                succ : fourvals -> fourvals
                                iszero : fourvals -> bool
                                eq : fourvals, fourvals -> bool
                           eqns succ(zero) = one       succ(one) = zero'
                                succ(zero') = one      succ(extra) = zero
                                iszero(zero) = true    iszero(one) = false
                                iszero(zero') = true   iszero(extra) = false
                                eq(zero,one) = false   eq(zero,zero') = false
                                      . . .                  . . .
                                eq(p,q) = eq(q,p)      eq(p,p) = true    enden
```

The iszero operator of Natmod2 and the eq operator of Fourvalues are needed to avoid trivial models.

All models of Fourvalues have a carrier containing 4 elements, and all models of Natmod2 have a 2-element carrier. Now consider the signature morphism σ:sig(Natmod2)$\to$sig(Fourvalues) given by [natmod2$\mapsto$fourvals, 0$\mapsto$zero, 1$\mapsto$one, succ$\mapsto$succ, iszero$\mapsto$iszero] (and everything in Bool maps to itself). Intuitively, Natmod2$\overset{\sigma}{\rightsquigarrow}$Fourvalues (zero and zero' both represent 0, one represents 1 and extra is unused) but is this an implementation according to the definition? Consider any model of Fourvalues (e.g. the term model -- all models are isomorphic). 'Forgetting' to the signature sig(Natmod2) eliminates the operators zero', extra and eq. Now we check if this algebra (call it A) simulates Natmod2.

- 'Restrict' removes the value of extra from the carrier.

- 'Identify' identifies the values of the terms "succ(1)" (=zero') and "0" (=zero).

The "no crime" condition of Natmod2's constraint requires that the values of true and false remain separate; this condition is satisfied, so A simulates Natmod2 and Natmod2$\overset{\sigma}{\rightsquigarrow}$Fourvalues is an implementation.

Suppose that the equation succ(zero')=one in Fourvalues were replaced by succ(zero')=zero. Forget (producing an algebra B) followed by restrict has the same effect on any model of Fourvalues, but now identify collapses the carrier for sort natmod2 to a single element (the closure of the equations in Natmod2 includes the

equation succ(succ(p))=p, so "succ(succ(0))" (=zero') is identified with "0" (=zero), and "succ(succ(1))" (=zero) is identified with "1" (=one)). Furthermore, the carrier for sort bool collapses; "iszero(succ(succ(1)))" (=true) is identified with "iszero(1)" (=false). The result fails to satisfy the "no crime" condition of the constraint, so B does not simulate Natmod2 and Natmod2$\xrightarrow{\sigma}$Fourvalues is no longer an implementation.

Implementation of parameterised theories

Parameterised theories in Clear are like functions in a programming language; they take zero or more values as arguments and return another value as a result. In Clear these values are theories. Here is an example of a parameterised theory (usually called a theory procedure in Clear):

```
meta Ident = enrich Bool by
               sorts element
               opns eq : element,element -> bool
               eqns eq(a,a) = true
                    eq(a,b) = eq(b,a)
                    eq(a,b) and eq(b,c) --> eq(a,c) = true    enden

proc Set(X:Ident) =
  let Set0 = enrich X by
               'data' sorts set
                      opns ∅ : set
                           singleton : element -> set
                           U : set,set -> set
                           is_in : element,set -> bool
                      eqns ∅ U S = S
                           S U S = S
                           S U T = T U S
                           S U (T U V) = (S U T) U V
                           a is_in ∅ = false
                           a is_in singleton(b) = eq(a,b)
                           a is_in S U T = a is_in S or a is_in T    enden in
    enrich Set0 by
           opns choose : set -> element
           eqns choose(singleton(a) U S) is_in (singleton(a) U S) = true    enden
```

Ident is a metatheory; it describes a class of theories rather than a class of algebras. Ident describes those theories having at least one sort together with an operator which satisfies the laws for an equivalence relation on that sort.

Ident is used to give the 'type' of the parameter for the procedure Set. The idea is that Set can be applied to any theory which matches Ident. Ident is called the metasort or requirement of Set. When Set is supplied with an appropriate actual parameter theory, it gives the theory of sets over the sort which matches element in Ident. For example

```
Set(Nat[element is nat, eq is ==])
```

gives the theory of sets of natural numbers (assuming that Nat includes an equality operator ==). Notice that a theory morphism (called the fitting morphism) must be provided to match Ident with the actual parameter. The result of an application is defined using pushouts as in [Ehr 82] (see [San 81] and [BG 80] for this and other aspects of Clear's semantics) but it is not necessary (for now) to know the details. In this paper we will consider only the single-parameter case; the extension to multiple parameters should pose no problems.

Note that parameterised theories in Clear are different from the parameterised specifications discussed by ADJ [TWW 78]. An ADJ parameterised specification works at the level of algebras, producing an algebra for every model of the parameter. A Clear parameterised theory produces a theory for each parameter theory. The result P(A) may have 'more' models than the theory A (this is the case when Set is applied to Nat, for example). Since ADJ parameterised specifications are a special case of Clear parameterised theories, all results given here hold for them as well.

Since a parameterised theory $\underline{R} \hookleftarrow \underline{P}$ (that is, a procedure with requirement theory $\underline{R}$ and body $\underline{P}$ -- $\underline{R}$ will always be included in $\underline{P}$) is a function taking a theory $\underline{A}$ as an parameter and producing a theory $\underline{P}(\underline{A})$ as a result, an implementation $\underline{R}' \hookleftarrow \underline{P}'$ of $\underline{R} \hookleftarrow \underline{P}$ is a function as well which takes any parameter theory $\underline{A}$ of $\underline{P}$ as argument and produces a theory $\underline{P}'(\underline{A})$ which implements $\underline{P}(\underline{A})$ as result. But this does not specify what relation (if any) must hold between the theories $\underline{R}$ and $\underline{R}'$. Since every actual parameter $\underline{A}$ of $\underline{R} \hookleftarrow \underline{P}$ (which must match $\underline{R}$) should be an actual parameter of $\underline{R}' \hookleftarrow \underline{P}'$, it must match $\underline{R}'$ as well. This requires a theory morphism $\mu:\underline{R}' \to \underline{R}$ (then a fitting morphism $\rho:\underline{R} \to \underline{A}$ gives a fitting morphism $\mu.\rho:\underline{R}' \to \underline{A}$).

Def: If $\underline{R} \hookleftarrow \underline{P}$ and $\underline{R}' \hookleftarrow \underline{P}'$ are parameterised theories, $\mu:\underline{R}' \to \underline{R}$ is a theory morphism and $\sigma:sig(\underline{P}) \to sig(\underline{P}')$ is a signature morphism, then $\underline{R}' \hookleftarrow \underline{P}'$ implements $\underline{R} \hookleftarrow \underline{P}$ (via σ and μ), written $\underline{R} \hookleftarrow \underline{P} \overset{\sigma}{\underset{\mu}{\rightsquigarrow}} \underline{R}' \hookleftarrow \underline{P}'$, if for all theories $\underline{A}$ with fitting morphism $\rho:\underline{R} \to \underline{A}$, $\underline{P}(\underline{A}[\rho]) \xrightarrow{\hat{\sigma}} \underline{P}'(\underline{A}[\mu.\rho])$ where $\hat{\sigma}$ is the extension of σ from $\underline{P}$ to $\underline{P}(\underline{A}[\rho])$ by the identity id (i.e. $\hat{\sigma}\big|_{sig(\underline{P})-sig(\underline{R})} = \sigma$ and $\hat{\sigma}\big|_{sig(\underline{A})} = id$).

Ordinarily $\underline{R}$ and $\underline{R}'$ will be the same theory, or at least the same modulo a change of signature. Otherwise $\underline{R}'$ must be weaker than $\underline{R}$.

Sometimes it is natural to split the implementation of a parameterised theory into two or more cases, implementing it for reasons of efficiency in different ways depending on some additional conditions on the parameters. For example:

- Sets: A set can be represented as a binary sequence if the range of possible values is small; otherwise it must be represented as a sequence (or tree, etc) of values.

- Parsing: Different algorithms can be applied depending on the nature of the grammar (operator precedence, LR, context sensitive, etc).

- Sorting: Distribution sort can be used if the range of values is small; otherwise quicksort.

In each instance the cases must exhaust the domain of possibilities, but they need not be mutually exclusive.

Our present notion of implementation does not treat such cases. We could extend it to give a definition of the implementation of a parameterised theory $\underline{R} \hookleftarrow \underline{P}$ by a collection of parameterised theories $\underline{R}'+\underline{R}_1' \hookleftarrow \underline{P}_1'$, ..., $\underline{R}'+\underline{R}_n' \hookleftarrow \underline{P}_n'$ (where for every theory $\underline{A}$ with a theory morphism $\sigma:\underline{R} \to \underline{A}$ there must exist some $i \geq 1$ such that $\sigma':\underline{R}'+\underline{R}_i' \to \underline{A}$ exists). But we force the case split to the abstract level, rather than entangle it with the already complex transition from abstract to concrete:

$$\underline{R} \hookleftarrow \underline{P} \dashrightarrow \underline{R}+\underline{R}_1 \hookleftarrow \underline{P}_1 = \underline{P}(\underline{R}+\underline{R}_1)$$

$$\vdots$$

$$\underline{R}+\underline{R}_n \hookleftarrow \underline{P}_n = \underline{P}(\underline{R}+\underline{R}_n)$$

This collection of n parameterised theories is equivalent to the original $\underline{R} \hookleftarrow \underline{P}$, in the sense that every theory $\underline{P}(\underline{A}[\sigma])$ with $\sigma:\underline{R} \to \underline{A}$ is the same as the theory $\underline{P}_i(\underline{A}[\sigma'])$ with $\sigma':\underline{R}+\underline{R}_i \to \underline{A}$ for some $i \geq 1$. (A theory of the transformation of Clear specifications is needed to discuss this matter in a more precise fashion; no such theory exists at present.) Now each case may be handled separately, using the normal definition of parameterised implementation:

$$\underline{R}+\underline{R}_1 \hookleftarrow \underline{P}_1 \rightsquigarrow \underline{R}'+\underline{R}_1' \hookleftarrow \underline{P}_1'$$

$$\vdots$$

$$\underline{R}+\underline{R}_n \hookleftarrow \underline{P}_n \rightsquigarrow \underline{R}'+\underline{R}_n' \hookleftarrow \underline{P}_n'$$

4. Examples

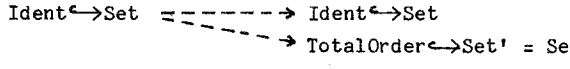
Sets (as defined in the last section) can be implemented using sequences. We must define sequences as well as operators on sequences corresponding to all the operators in Set. We begin by defining everything except the choose operator:

```
meta Triv = theory sorts element    endth

proc Sequence(X:Triv) =
  enrich X + Bool by
    'data' sorts sequence
           opns empty : sequence
                unit : element -> sequence
                . : sequence,sequence -> sequence
                head : sequence -> element
                tail : sequence -> sequence
           eqns empty.s = s
                s.empty = s
                s.(t.v) = (s.t).v
                head(unit(a).s) = a
                tail(unit(a).s) = s    enden

proc SequenceOpns(X:Ident) =
  enrich Sequence(X) by
    opns is_in : element,sequence -> bool
         add : element,sequence -> sequence
         U : sequence,sequence -> sequence
    eqns a is_in empty = false
         a is_in unit(b) = eq(a,b)
         a is_in s.t = a is_in s or a is_in t
         add(a,s) = s  if a is_in s
         add(a,s) = unit(a).s  if not(a is_in s)
         empty U s = s
         unit(a).t U s = add(a,t U s)    enden
```

The head and tail operators of Sequence and their defining equations are needed to avoid trivial models; they serve no other function in the specification.

Before dealing with the choose operator, we split Set into two cases:

```
meta TotalOrder = enrich Ident by
                    opns ≤ : element,element -> bool
                    eqns a≤a = true
                         a≤b and b≤a --> eq(a,b) = true
                         a≤b and b≤c --> a≤c = true
                         a≤b or b≤a = true    enden
```



These two cases may be handled separately. The choose operator can select the minimum element when the element type is totally ordered; otherwise we can leave the precise choice unspecified as before.

```
proc SequenceAsSet(X:Ident) =
  enrich SequenceOpns(X) by
    opns choose : sequence -> element
    eqns choose(unit(a).t) is_in (unit(a).t) = true    enden

proc SequenceAsSet'(X:TotalOrder) =
  enrich SequenceOpns(X) by
    opns choose : sequence -> element
    eqns choose(unit(a)) = a
         choose(unit(a).unit(b).s) = choose(unit(a).s) if a≤b
                                   else choose(unit(b).s)    enden
```

Now Ident↪Set $\xrightarrow{\sigma}_{\mu}$ Ident↪SequenceAsSet and TotalOrder↪Set' $\xrightarrow{\sigma}_{\mu'}$ TotalOrder↪SequenceAsSet', where σ = [element↦element, eq↦eq, set↦sequence, ∅↦empty, singleton↦unit, U↦U, is_in↦is_in, choose↦choose] (and everything in the signature of Bool maps to itself), and μ and μ' are the identity morphisms on Ident and TotalOrder respectively. Note that choose is not specified for empty sets and sequences -- although the same notion of implementation should work for error theories and algebras, we prefer to avoid the issue of errors for now. Also note that an incorrect implementation results if choose in SequenceAsSet is changed to select the first element; Set contains an equation choose(singleton(x) U singleton(y))=choose(singleton(y) U singleton(x)), so the identify step would collapse the parameter sort (and consequently bool).

This example illustrates all of the features of our notion of implementation. Not all sequences are needed to represent sets -- sequences with repeated elements are not used. Each set is represented by many sequences, since the sequence representation of a set keeps track of the order in which elements were inserted. Set is split into two theories before implementation, and finally SequenceAsSet' is 'tighter' than Set' because the choose operator (select an element) is implemented by an operator which chooses the minimum element.

A nonparameterised example is obtained by applying Set or Set' and SequenceAsSet or SequenceAsSet' to an argument, for example:

Set(Nat[element is nat, eq is ==])$\xrightarrow{\sigma}$SequenceAsSet(Nat[element is nat, eq is ==])

where σ is the same as σ above except that element↦element is replaced by nat↦nat.

Two additional examples:

- Lists can be implemented using arrays of (value,index) pairs, where the index points to the next value in the list (and where some distinguished index value denotes nil). There are many representations for the same list (the relative positions of cells in the array are irrelevant, for example) and circular structures are not needed to represent the value of any list.

- The specification of matrix inversion in the Introduction can be implemented by a specification of matrix inversion using the Gauss-Seidel method. Conversely, this specification can be implemented by the specification in the Introduction (enriched by some auxiliary functions).

The matrix inversion example shows that the expectation that A⟶B should imply that B is 'lower level' than A is not always justified. This is because the definition of implementation is concerned with classes of models rather than with the equations used to describe those classes. In this case both theories will have the same class of models except that the Gauss-Seidel method will probably require auxiliary operators.

5. Horizontal and vertical composition

Large specifications are needed to solve large problems. But a large monolithic specification of a compiler (for example) would be impossible to understand because of the sheer numbers of interacting operators and equations. The value of a specification depends on the ease with which it was written and can be understood; a large number of pages full of equations are not of much use to anybody.

The Clear [BG 77] and CIP-L [Bau 81] specification languages were invented to combat just this problem. Clear and CIP-L are languages for writing structured specifications; that is, they provide facilities for combining small theories in various ways to make large theories. A large specification can thus be built from small easy-to-understand bits. Following [GB 80] this shall be called horizontal structure.

Likewise, the implementation of a large specification is not done all at once; it is good programming practice to implement and test pieces of the specification separately and then construct a final system from the finished components. If the theories which make up a Clear or CIP-L specification are implemented separately, it should be possible to put together (horizontally compose) the implementations in the same way that the theories themselves are put together, yielding an implementation of the entire specification.

Although the problem of developing a program from a specification is simplified by dividing it into smaller units, the step from specification of a component to its implementation as a program is still often uncomfortably large. A way to conquer this is to break the development of a program into a series of consecutive refinement steps. That is, the specification is refined to a lower level specification, which is in turn refined to a still lower level specification, and so on until a program is obtained. Again following [GB 80], this is called the vertical structure (of the development process). If a specification A is implemented by another specification B, and B is implemented by C, then these implementations should vertically compose to give an implementation of A by C. Goguen and Burstall [GB 80] propose a system called CAT for the structured development of programs from specifications by composing implementations in both the horizontal and vertical dimensions.

The vertical composition of two implementations is not always an implementation. For example, consider the following theories:

```
const T = enrich Bool by
            opns extra : bool   enden

const T' = enrich Bool by
             opns extra : bool
             eqns extra = true   enden

const T" = theory 'data' sorts three
                          opns tt, ff, extra : threevals   endth
```

Now $T \rightsquigarrow T'$ and $T' \rightsquigarrow T"$ but $T \not\rightsquigarrow T"$ (consider the model of T" where $tt \neq ff \neq extra$). The theories must satisfy an extra condition.

Def: A theory $\underline{T}$ is reachably complete with respect to a parameterised theory $\underline{R} \hookleftarrow \underline{P}$ with $\underline{P} \subseteq \underline{T}$ if $\underline{T}$ is sufficiently complete with respect to opns($\underline{P}$), constrained-sorts($\underline{P}$), constrained-opns($\underline{P}$), and variables of sorts($\underline{R}$) $\cup$ unconstrained-sorts($\underline{P}$). A theory $\underline{T}$ is reachably complete with respect to a nonparameterised theory $\underline{A}$ if it is reachably complete with respect to $\emptyset \hookleftarrow \underline{A}$.

In the example above T" is not reachably complete with respect to T because extra is not provably equal to either tt or ff.

Vertical composition theorem

1. [Reflexivity] $\underline{T} \overset{id}{\rightsquigarrow} \underline{T}$.

2. [Transitivity] If $\underline{T} \overset{\sigma}{\rightsquigarrow} \underline{T}'$ and $\underline{T}' \overset{\sigma'}{\rightsquigarrow} \underline{T}"$ and $\underline{T}"$ is reachably complete with respect to $\sigma.\sigma'(\underline{T})$, then $\underline{T} \overset{\sigma.\sigma'}{\rightsquigarrow} \underline{T}"$.

Corollary

1. [Reflexivity of parameterised implementations] $\underline{R} \hookleftarrow \underline{P} \overset{id}{\underset{id}{\rightsquigarrow}} \underline{R} \hookleftarrow \underline{P}$.

2. [Transitivity of parameterised implementations] If $\underline{R} \hookleftarrow \underline{P} \overset{\sigma}{\underset{\mu}{\rightsquigarrow}} \underline{R}' \hookleftarrow \underline{P}'$ and $\underline{R}' \hookleftarrow \underline{P}' \overset{\sigma'}{\underset{\mu'}{\rightsquigarrow}} \underline{R}" \hookleftarrow \underline{P}"$ and $\underline{P}"$ is reachably complete with respect to $\sigma.\sigma'(\underline{R}) \hookleftarrow \sigma.\sigma'(\underline{P})$, then $\underline{R} \hookleftarrow \underline{P} \overset{\sigma.\sigma'}{\underset{\mu'.\mu}{\rightsquigarrow}} \underline{R}" \hookleftarrow \underline{P}"$.

In the absence of constraints (as in the initial algebra [GTW 78] and final algebra [Wan 79] approaches), reachable completeness is guaranteed so this extra condition is unnecessary.

To prove that implementations of large theories can be built by arbitrary horizontal composition of small theories, it is necessary to prove that each of Clear's theory-building operations (combine, enrich, derive and apply) preserves implementations. We will concentrate here on the application of parameterised theories and the enrich operation. Extension of these results to the remaining operations should not be difficult.

For the apply operation our object is to prove the following property of implementations:

Horizontal Composition Property: $\underline{R} \hookleftarrow \underline{P} \rightsquigarrow \underline{R}' \hookleftarrow \underline{P}'$ and $\underline{A} \rightsquigarrow \underline{A}'$ implies $\underline{P}(\underline{A}) \rightsquigarrow \underline{P}'(\underline{A}')$.

But this is not true in general; in fact, $\underline{P}'(\underline{A}')$ is not even always defined. Again, some extra conditions must be satisfied for the desired property to hold.

Def: Let $\underline{R} \hookleftarrow \underline{P}$ be a parameterised theory.

- $\underline{R} \hookleftarrow \underline{P}$ is called structurally complete if $\underline{P}$ is sufficiently complete with respect to opns($\underline{P}$), sorts($\underline{R}$)$\cup$constrained-sorts($\underline{P}$), opns($\underline{R}$)$\cup$constrained-opns($\underline{P}$), and variables of sorts($\underline{R}$)$\cup$unconstrained-sorts($\underline{P}$). A nonparameterised theory $\underline{A}$ is called structurally complete if $\emptyset \hookleftarrow \underline{A}$ is structurally complete.

- $\underline{R} \hookleftarrow \underline{P}$ is called parameter consistent if $\underline{P}$ is conservative with respect to $\underline{R}$.

If $\underline{R}' \hookleftarrow \underline{P}'$ is structurally complete, parameter consistent and reachably complete, and $\underline{A}'$ is structurally complete and a valid actual parameter of $\underline{R}' \hookleftarrow \underline{P}'$, then the horizontal composition property holds.

Horizontal composition theorem: If $\underline{R} \hookleftarrow \underline{P}$ and $\underline{R}' \hookleftarrow \underline{P}'$ are parameterised theories with $\underline{R}' \hookleftarrow \underline{P}'$ structurally complete and parameter consistent, $\underline{P}'$ is reachably complete with respect to $\sigma(\underline{R}) \hookleftarrow \sigma(\underline{P})$, $\underline{R} \hookleftarrow \underline{P} \overset{\sigma}{\rightsquigarrow} \underline{R}' \hookleftarrow \underline{P}'$ and $\underline{A} \overset{\sigma'}{\rightsquigarrow} \underline{A}'$ are implementations with $\underline{A}'$ structurally complete, and $\rho:\underline{R} \to \underline{A}$ and $\rho':\underline{R}' \to \underline{A}'$ are theory morphisms where $\rho' = \mu.\rho.\sigma'$, then $\underline{P}(\underline{A}[\rho]) \overset{\tilde{\sigma}\,\tilde{\sigma}'}{\rightsquigarrow} \underline{P}'(\underline{A}'[\rho'])$, where $\tilde{\sigma}'|_{sig(\underline{P}(\underline{A}[\rho]))-sig(\underline{A})} = id$ and $\tilde{\sigma}'|_{sig(\underline{A})} = \sigma'$.

Corollary (Horizontal composition for enrich): If $\underline{A} \overset{\sigma}{\rightsquigarrow} \underline{A}'$ is an implementation, $\underline{B}$ = enrich $\underline{A}$ by <stuff> and $\underline{B}'$ = enrich $\underline{A}'$ by σ<stuff>, $\underline{A}' \hookleftarrow \underline{B}'$ is structurally complete and parameter consistent, $\underline{B}'$ is reachably complete with respect to $\sigma(\underline{A}) \hookleftarrow \sigma(\underline{B})$ and $\underline{A}'$ is structurally complete, then $\underline{B} \overset{\tilde{\sigma}}{\rightsquigarrow} \underline{B}'$, where $\tilde{\sigma}|_{sig(\underline{B})-sig(\underline{A})} = id$ and $\tilde{\sigma}|_{sig(\underline{A})} = \sigma$.

A consequence of this corollary is that our vertical and horizontal composition theorems extend to more elaborate notions of implementation such as the one discussed in [EKP 80]. Again, reachable completeness is guaranteed in the absence of constraints.

The vertical and horizontal composition theorems give us freedom to build the implementation of a large specification from many small implementation steps. The correctness of all the small steps guarantees the correctness of the entire implementation, which in turn guarantees the correctness of the low-level 'program' with respect to the high-level specification. This provides a formal foundation for a methodology of programming by stepwise refinement. CAT's 'double law' [GB 80] is an easy consequence of the vertical and horizontal composition theorems. This means that the order in which parts of an implementation are carried out makes no difference, and that our notion of implementation is appropriate for use in CAT.

Our notions of simulation and implementation extend without modification to ordinary Clear (with data constraints rather than hierarchy constraints); all of the results in this paper then remain valid except for the horizontal composition theorem and its corollary. These results hold only under an additional condition.

Def: A data theory T is hierarchical submodel consistent if for every model M of T and every hierarchical submodel M^- of M (i.e. every submodel of M satisfying the constraints of T when viewed as hierarchy constraints), M^- satisfies the data constraints of T.

Horizontal composition theorem (with data): In Clear with data, if $R \hookrightarrow P$ and $R' \hookrightarrow P'$ are parameterised theories with $R' \hookrightarrow P'$ structurally complete and parameter consistent, P' is hierarchical submodel consistent and reachably complete with respect to $\sigma(R) \hookrightarrow \sigma(P)$, $R \hookrightarrow P \overset{\sigma}{\underset{\mu}{\rightsquigarrow}} R' \hookrightarrow P'$ and $A \overset{\sigma'}{\rightsquigarrow} A'$ are implementations with A' structurally complete, and $\rho: R \to A$ and $\rho': R' \to A'$ are theory morphisms where $\rho' = \mu.\rho.\sigma'$, then $P(A[\rho]) \overset{\sigma.\sigma'}{\rightsquigarrow} P'(A'[\rho'])$.

The horizontal composition theorem for enrich extends analogously.

This result is encouraging because ordinary Clear is easier to use than our 'hierarchical' variant. However, the extra condition on the horizontal composition theorem is rather strong and it may be that it is too restrictive to be of practical use.

Acknowledgements

We are grateful to the work of Ehrig, Kreowski and Padawitz [EKP 80] for a start in the right direction. Thanks: from DS to Rod Burstall for guidance, from MW to Manfred Broy and Jacek Leszczylowski for interesting discussions, to Burstall and Goguen for Clear, to Bernhard Möller for finding a mistake, and to Oliver Schoett for helpful criticism. This work was supported by the University of Edinburgh, by the Science and Engineering Research Council, and by the Sonderforschungsbereich 49, Programmiertechnik, München.

REFERENCES

Note: LNCS n = Springer Lecture Notes in Computer Science, Volume n

[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.

[BDPPW 79] Broy, M., Dosch, W., Partsch, H., Pepper, P. and Wirsing, M. Existential quantifiers in abstract data types. Proc. 6th ICALP, Graz, Austria. LNCS 71, pp. 73-87.

[BMPW 80] Broy, M., Möller, B., Pepper, P. and Wirsing, M. A model-independent approach to implementations of abstract data types. Proc. of the Symp. on Algorithmic Logic and the Programming Language LOGLAN, Poznan, Poland. LNCS (to appear).

[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.

[BMS 80] Burstall, R.M., MacQueen, D.B. and Sannella, D.T. HOPE: an experimental applicative language. Proc. 1980 LISP Conference, Stanford, California, pp. 136-143; also Report CSR-62-80, Dept. of Computer Science, Univ. of Edinburgh.

[Dij 72] Dijkstra, E.W. Notes on structured programming. Notes on Structured Programming (Dahl O.-J., Dijkstra, E.W. and Hoare, C.A.R.), Academic Press, pp. 1-82.

[Ehr 81] Ehrich, H.-D. On realization and implementation. Proc. 10th MFCS, Strbske Pleso, Czechoslovakia. LNCS 118.

[Ehr 82] Ehrich, H.-D. On the theory of specification, implementation, and parameterization of abstract data types. JACM 29, 1 pp. 206-227.

[EK 82] Ehrig, H. and Kreowski, H.-J. Parameter passing commutes with implementation of parameterized data types. Proc. 9th ICALP, Aarhus, Denmark (this volume).

[EKP 80] Ehrig, H., Kreowski, H.-J. and Padawitz, P. Algebraic implementation of abstract data types: concept, syntax, semantics and correctness. Proc. 7th ICALP, Noordwijkerhout, Netherlands. LNCS 85, pp. 142-156.

[Gan 81] Ganzinger, H. Parameterized specifications: parameter passing and implementation. TOPLAS (to appear).

[GB 80] Goguen, J.A. and Burstall, R.M. CAT, a system for the structured elaboration of correct programs from structured specifications. Computer Science Dept., SRI International.

[GTW 78] Goguen, J.A., Thatcher, J.W. and Wagner, E.G. An initial algebra approach to the specification, correctness, and implementation of abstract data types. Current Trends in Programming Methodology, Vol. 4: Data Structuring (R.T. Yeh, ed.), Prentice-Hall, pp. 80-149.

[Grä 79] Grätzer, G. Universal Algebra (2nd edition), Springer.

[GH 78] Guttag, J.V. and Horning, J.J. The algebraic specification of abstract data types. Acta Informatica 10 pp. 27-52.

[Hup 80] Hupbach, U.L. Abstract implementation of abstract data types. Proc. 9th MFCS, Rydzyna, Poland. LNCS 88, pp. 291-304.

[Hup 81] Hupbach, U.L. Abstract implementation and parameter substitution. Proc. 3rd Hungarian Computer Science Conference, Budapest.

[KR 71] Kaphengst, H. and Reichel, H. Algebraische Algorithmentheorie. VEB Robotron, Zentrum für Forschung und Technik, Dresden.

[MS 82] MacQueen, D.B. and Sannella, D.T. Completeness of proof systems for equational specifications. In preparation.

[Nou 79] Nourani, F. Constructive extension and implementation of abstract data types and algorithms. Ph.D. thesis, Dept. of Computer Science, UCLA.

[Rei 80] Reichel, H. Initially-restricting algebraic theories. Proc. 9th MFCS, Rydzyna, Poland. LNCS 88, pp. 504-514.

[San 81] Sannella, D.T. A new semantics for Clear. Report CSR-79-81, Dept. of Computer Science, Univ. of Edinburgh.

[Sch 81] Schoett, O. Ein Modulkonzept in der Theorie Abstrakter Datentypen. Report IFI-HH-B-81/81, Fachbereich Informatik, Universität Hamburg.

[TWW 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.

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

[WB 81] Wirsing, M. and Broy, M. An analysis of semantic models for algebraic specifications. International Summer School on Theoretical Foundations of Programming Methodology, Marktoberdorf.

[Wir 71] Wirth, N. Program development by stepwise refinement. CACM 14, 4 pp. 221-227.