Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

181

Foundations of Software Technology and Theoretical Computer Science

Fourth Conference, Bangalore, India, December 1984
Proceedings

Edited by Mathai Joseph and Rudrapatna Shyamasundar

Springer-Verlag
Berlin Heidelberg New York Tokyo

LOGICAL SPECIFICATION AND IMPLEMENTATION

T S E Maibaum*, M R Sadler*, P A S Veloso**

* Dept. of Computing
Imperial College of Science and Technology
180 Queen's Gate, London SW7 2BZ

** Departmento de Informatica
Pontificia Universidade Catolica
Rua Marques de Sao Vicente, 225
22453 Rio de Janeiro, RJ Brazil

Abstract

It has become customary to focus attention on the semantic aspects of specification and implementation, a model theoretic or algebraic viewpoint. We feel, however, that certain concepts are best dealt with at the syntactic level, rather than via a detour through semantics, and that implementation is one of these concepts. We regard logic as the most appropriate medium for talking about specification (whether of abstract data types, programs, databases, specifications - as an interpretation between theories say, rather than something to do with the embedding of models or mapping of algebras. In this paper, we give a syntactic account of implementation and prove the basic results - composability of implementations and how to deal with structured (hierarchical) specifications modularly - for abstract data types.

Introduction

As we see it, the two key concepts in an approach to, or theory of, specification are the notions of specification (our objects if you like) and implementation (the morphisms between objects). At this stage we feel that is not as appropriate to investigate the category-theoretic properties of these notions (giving us the category(s) of specifications and implementations), as to continue to explore particular ways of looking at these notions based on various mathematical formalisms (algebra, set-theory, logic for example) and how these formalisms support more complex ideas like parameterisation and other mechanisms for structuring specifications.

Our claim is that logic, or the logical approach, with an emphasis on syntactic ideas is a particularly fruitful formalism. In this paper we show how the logical approach supports specification of abstract data types and implementations of abstract data types within other abstract data types.

Given an area of computing science an important first step for a theory of specification with respect to that area is an identification of what is (are) the basic unit(s) of specification, see [LZ]. That is, the packages that are used as the atomic building blocks for building more complex, structured specifications. Here we consider abstract data types as our units, or atoms with programming as the obvious area in mind. Similarly any formalism offers, or studies, various structures: in the case of logic, logics or theories say. Now part of what we would like for an approach to specification is a natural match, in some sense, between the formal structures and the units of specification. And this natural match should also extend to a match between on the one hand the kinds of mappings between our formal structures and on the other the ways we naturally put our specifications together to form structured specifications and implement specifications in each other.

The formal structures to study in a logical approach are the theories given by languages, L, over some fixed consequence relation |- determining the logic. The decision to follow a logical approach appears in no way to commit one to any particular |-, other than a requirement that certain meta-theorems are provable about |-, the Craig Interpolation Lemma for example. (See 'Theorem' in the section: Using Implementations).

We choose to use an infinitary logic and this is where criticism is often focused. However we feel that such criticism misses the point. The major focus of attention should be as to whether we have this natural match between units of specification and formal structures, here between abstract data types and theories. We claim this match is obviously natural.

We cannot always expect (for any given approach) things to be so simple, for more complex objects more complex formal structures might be required. For specification of databases, for example, families of modal theories (and even families of families of modal theories) form the more appropriate formal structures, see [KMS].

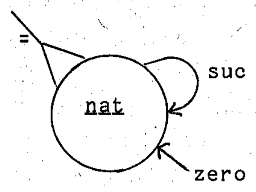
Below we explain the logical approach to specification, implementation (in some detail) and how to use implementations to support the software design process.

Specifications

We begin by reviewing the approach to specification outlined in [MV]. There, structuring of specifications was defined in terms of semantic concepts but here a purely syntactic line using an extended first order logic with infinitary formulae and infinitary rules is taken. We conceive of specifications as theories within this modification of first order logic. The reader is assumed to be familiar with the concepts of first order logic, see, for example [END], [SCH], but we explain the modifications we make as the formal details are presented.

Expressiveness in our approach is determined by the use of many sorted languages. We require an equality symbol for each sort, but regard them as part of the non-logical vocabulary. Thus the equality symbols are not regarded as logical constants as in most approaches based on logic or algebra. In practice, rather than presenting such languages as lists of sorts, operations and so on, we use syntax diagrams to convey the information pictorially. For example:

L_{nat}:



L_{nat} is an appropriate language for the natural numbers. That is, L_{nat} has one sort, nat, one constant, zero, of type (NIL,nat), a further operation, suc, of type (<nat>,nat) and a predicate symbol, =, of type <nat,nat> (where NIL is the empty string, <nat> and <nat,nat> are strings of lengths one and two respectively). For clarity, we usually suppress equality symbols from such diagrams.

The set of terms, Term(L), over a language L is defined in the usual way and we define the names, Name(L), of L to be the variable free terms in Term(L). The formulae and sentences (closed formulae) of L present our first opportunity for extending the more common first order notions. Our formulae are those of the extension to the traditional first-order language given by adding infinitary disjunctions (V). The sets of such formulae and sentences we denote by Form(L), Sent(L) respectively.

Example

for-all x ($V_{n \text{ in } N}$ x = n)

where N is an enumeration of Name(L_{nat}). Thus this formula is equivalent to:

for-all x (x=zero v x=suc(zero) v x=suc(suc(zero)) v ...).

This formula indicates that the only allowed values (up to =) over which a variable x of sort <u>nat</u> can range are the names in Name(L_{nat}). (Thus any structure satisfying this sentence will have no nonstandard values for the natural numbers.)

A **specification**, S, is a pair (L,A) where L is a many sorted language (with infinite disjunctions), A is a consistent subset of Sent(L) and where, for each sort s of L, we have:

i) L contains $=_s$ a predicate symbol of type <s,s> and A contains the usual congruence properties for $=_s$ - reflexivity, symmetry, transitivity and substitutivity. We usually drop the subscript from $=_s$ when it is clear from the context,

ii) A contains for-all x ($V_{n \text{ in } \mathrm{Name}(s)}$ $x =_s n$) where x is a variable of sort s, where Name(s) are the names of sort s in Name(L). (We call such sentences **namability** axioms.)

Again we usually suppress this information in presenting any specification.Note specifications need not contain boolean values and operations - these can be left as logical rather than non-logical concepts.

Examples:

NAT = (L_{nat}, A_{nat}) where A_{nat} would also include:

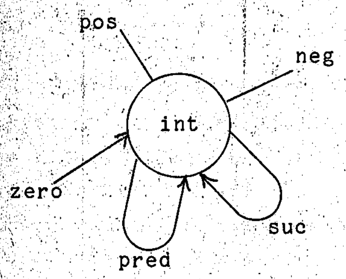
suc(x)=suc(y) -> x=y

-(suc(x)=zero)

and where leading universal quantifiers are suppressed.

INT = (L_{int}, A_{int}) where:

L_{int}:



A_{int}:

suc(x)=suc(y) -> x=y
pred(x)=pred(y) -> x=y
pred (suc(x))=x
suc(pred(x))=x
pos(x) -> pos(suc(x))
neg(x) -> neg(pred(x))
pos(suc(zero)
neg(pred(zero))
-pos(zero)
-neg(zero)

A specification (L,A) defines a theory Con(A), which consists of all provable sentences, or logical consequences, of L from A where the concept of proof is based on usual first order notions together with the following omega-rule (ie, infinitary rule) or some appropriate variant:

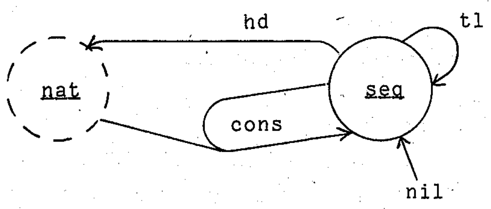
$$\frac{\text{for-all } x\ (\ V_{n \text{ in } N}\ x{=}n),\ Q(n_j)\ \text{for each } n_j \text{ in } J,\ J \text{ a subset of } T}{\text{for-all } x\ (\ V_{n \text{ in } N-J}\ x{=}n \ \text{v}\ Q(x))}$$

That is, if we can prove some property Q for some (possibly infinite) subset J of N, then we can replace the disjuncts involving elements of J by the property Q. Thus, for example, we can derive from the above specification (L_{nat},A_{nat}) the sentence for-all x(x=zero v there-exists y (x=suc(y)). One form of the usual induction formula for the natural numbers. We use A |- Q to denote that Q is a logical consequence of A.

We can use namability axioms to help structure our specifications.

Consider, for example, the extension of NAT obtained by adding to L_{nat} the following:

L_{seq}:



and the axioms:

A_{seq}:
$$\begin{array}{l}
hd(cons(x,l)) =_{nat} x \\
tl(cons(x,l)) =_{seq} l \\
tl(nil) =_{seq} nil \\
-(nil =_{seq} cons(x,l)) \\
cons(x,l) =_{seq} cons(y,l') \rightarrow x =_{nat} y \ \& \ l =_{seq} l' \\
V_n \text{ in } Name(seq) \ l =_{seq} n
\end{array}$$

The (structured) specification, which we denote by (L_{seq},A_{seq})[NAT], has the property that no new natural numbers are introduced by any of the sequence operations as the namability axiom for NAT still ensures that all names for natural numbers must be of the form zero or suc(zero) or ... or equivalent to one of these. Note that this is a much weaker requirement than the sufficient completeness of [GH], [GAN], [WB], etc. Note also that the specification is partial (loose, incomplete, permissive) since no axiom tells us which natural number is the result of hd(nil). Looseness in specifications (the abandonment of a unique isomorphism class of models) has also been introduced by [BG] and [WB]. A different notion of structured specifications was also introduced in [WB] where they were called hierarchical specifications.

The above extension is in fact conservative ([END], [SCH]) - the extended theory contains no new theorems about the language being extended. That is (L_{seq},A_{seq})[NAT] has no theorems in Sent(L_{nat}) that are not provable from A_{nat}.

Semantics for specifications are provided by logical structures satisfying the axioms. Again we should point out that objects related by = symbols need not be identified in models. The namability axioms also ensure that we have only the so-called finitely generated models of [WB].

We mention the following completeness result.

Theorem:
Given a specification S and a sentence Q, then Q is valid in all models of S (ie namable models) iff Q is provable from S.

Implementation

In the corpus of work on specification of data types and programs, theories of implementation have occupied a very important place. It is via such theories that the informal software engineering notion of stepwise refinement can be incorporated into a formalisation of the programming process. Amongst the large amount of material produced on this subject, the most notable are [GTW], [HUP], [EKP], [EHR], [GAN], [EK], [SW], all of which use algebras as semantics for specifications. There is also some work on implementations using logic, as in [NOU].

Clearly, an implementation relates two specifications and the general approach used in the work referenced above is to relate the two specifications via their semantics - by applying various constructions to an algebra (model) satisfying the target specification, one can obtain an algebra satisfying the source specification. We believe that arguing in terms of models is wasteful, both formally and practically, and present a purely syntactic theory of implementation.

The concept of implementation we use is based on the logical notion of interpretation between theories. Thus, our specifications generate corresponding theories and these are related by interpretations. An interpretation shows how one can realise the concepts of one theory in terms of another - this being more or less the process described as implementation. The theories of implementation presented in other approaches do not use interpretations and hence have to resort to non-syntactic reasoning. Moreover, the composition of implementations (which needs to be defined in order to formalise the stepwise refinement process) does not have an adequate definition in any of the algebraic approaches mentioned above. This inadequacy manifests itself both in the shortcomings of the formal properties as well as in not modelling the software engineering practice which seems to work.

Formal inadequacy has been demonstrated by [PV] where it is shown that certain desirable properties are not preserved by composition as well as in [EHR], where composition is not constructive (and so inadequate for practice). Practical inadequacy can be used to illustrate what is wrong with these definitions.

Suppose that a software engineer sets out to write some program using sets. Having proved the correctness of his abstract program using the properties of set operations, he then implements sets in terms of sequences, say. Together with the abstract program, the procedures/functions defining set operations in terms of sequence operations then becomes an 'executable' implementation if the programming language has sequences as a built in data type. If not, then a further suite of procedures/functions is defined to implement sequences and the abstract program together with the set operation implementation and the sequence operation implementation constitute the final implementation. In the theories mentioned above, it is felt necessary to eliminate the equivalent of the sequence procedures (which in the program interface between the abstract program using sets and the language based constructs in terms of which sequences are implemented). This is analogous to defining recursively f in terms of g, g in terms of h, and then trying to compose the two by eliminating any occurrence of g. Often no such finite definition can be obtained by simple substitutions of definitions for symbols - hence the problems indicated in [EHR] and [PV].

When interpretations between theories are composed, no such attempt at eliminating the mediating language is made. Thus these problems are avoided.

An interpretation between theories is a translation between the underlying languages, terms, formulae, etc, which respects the properties expressed in the theory being interpreted. Thus each sort of the source language is mapped to a (tuple of) sort(s) of the target language, non-logical symbols to appropriate non-logical symbols and quantifiers to restricted quantifiers - ie, quantifiers relativised to predicates.

First an example:

We will informally discuss how to interpret INT = (L_{int}, A_{int}) by means of NBOOL = (L_{NB}, A_{NB}) which is an extension of NAT obtained by adding the sort bool, the symbols T, F of type (NIL,bool) and the axiom $\neg T=F$. Note that the namability axiom for bool is simply $b=T \vee b=F$.

Our intention is, of course, to represent the integer n (an abbreviation for n applications of suc to zero) by n of nat and T of bool and -n (an abbreviation for n applications of pred to zero) by n of nat and F of bool. Zero of Int can be represented in two ways (as zero and T or as zero and F). Note that, unlike other theories of data types, we do not have to create a new sort nat x bool and associated pairing and projection functions (of course we are not prevented from doing so). There is also some leeway in implementing the function symbols of L_{int}, either as a single function that returns two values (one from nat and one from bool), or as two functions each returning one value (one function having result nat and the other result bool). We choose the latter option here, but in general choose whichever option seems more appropriate for a given context.

For a term t of Term(L_{int}) we define two components $t_B{}^I$ and $t_B{}^I$ both of Term(L_{NB}) where the subscripts N and B provide us with sort information, nat and bool respectively. Thus our interpretation I consists of the following:

i) We associate with the sort int the pair of sorts nat and bool.

ii) We associate with int a relativisation predicate is_int of type <bool,nat> which we add to L_{NB} and which is defined by the following axiom:

$$\text{is\_int}(x_B, x_N)$$

This axiom says that any pair of boolean and natural number values represents some integer. (In general, we will not be so lucky. Only some values in the target of an interpretation will represent values in the source.)

iii) We associate with each function symbol (including the constants) of L_{int} a pair of function symbols which are added to L_{NB} and whose typing respects the mapping of sorts defined in (i). So we associate with zero of int the zero of nat and T of bool.

(Thus we choose one of the two possibilities mentioned above. The other, the pair zero and F will be equivalent to the pair zero and T.) To suc (pred) of L_{int} we associate sucrepN (predrepN) of type (<bool,nat>,nat), and sucrepB (predrepB) of type (<bool,nat>,bool).

iv) We associate with predicate symbols of L_{int}, predicate symbols which we add to L_{NB} and with typing which respects the mapping defined in (i). Thus to pos (neg) we associate posrep (negrep) of type <bool,nat>. To $=_{int}$ we associate =rep of type (bool,bool,nat,nat). (Note that equality is implemented like any other predicate.)

v) We associate with every variable x of sort int a corresponding pair of variables x_N and x_B of sorts nat and bool, respectively. We also add to A_{NB} axioms defining the new symbols we have added in steps (iii) and (iv). (is_int, added in step (ii), was also defined there via a new axiom added to A_{NB}). Note that =,suc,zero,T and F below are symbols in L_{NB}:

posrep(x_B,x_N) <-> (x_B=T & -x_N=zero)
negrep(x_B,x_N) <-> (x_B=F & -x_N=zero)
=rep (x_B,y_B,x_N,y_N) <-> ((x_N=y_N & x_B=y_B) v (x_N=zero & y_N=zero))

(So two pairs repesenting integers are equivalent - represent the same integer - if the pairs are identical or if the natural number element in each pair is zero.)

x_B=T -> (sucrepN(x_B,x_N)=suc(x_N) & (sucrepB(x_B,x_N)=T)
(x_B=F & -(x_N=zero)) -> (sucrepB(x_B,x_N)=F) &
there-exists y_N(sucrepN(x_B,x_N)=y_N & suc(y_N)=x_N))
(x_B=F & x_N=zero) -> (sucrepB(x_B,x_N)=T & sucrepN(x_B,x_N)=suc(zero)

(These three axioms define sucrepN and sucrepB by cases.)
We add similar axioms for predrepN and predrepB.

Having added the above symbols and axioms to NBOOL, we get a specification ENBOOL. We remark that all values in this new theory are still described by those provided by NBOOL because of the namability axioms for nat and bool. Thus models of ENBOOL can be obtained only by extending those of NBOOL without adding new objects.

(i) - (iv) above define a translation I of terms from INT to ENBOOL

Thus, for example:

$\text{zero}_N{}^I = \text{zero}, \quad \text{zero}_B{}^I = T$
$(\text{suc}(t))_N{}^I = \text{sucrepN}(t_B{}^I, t_N{}^I)$
$(\text{suc}(t))_B{}^I = \text{sucrepB}(t_B{}^I, t_N{}^I)$ etc.

We extend this translation to atomic formulae by:

$\text{pos}(t)^I = \text{posrep}(t_B{}^I, t_N{}^I), \quad \text{neg}(t)^I = \text{negrep}(t_B{}^I, t_N{}^I)$
$(t{=}u)^I = {=}\text{rep}(t_B{}^I, u_B{}^I, t_N{}^I, u_N{}^I)$

For sentences in general, we have to be careful because we want translated sentences to hold only for objects which really represent values in the source of the translation I. (Although we had no 'junk' elements here, in general we do.) We do this by relativising quantifiers - ie, we condition quantifiers to range only over those values which satisfy the relativisation predicates as these are meant to define such representatives. For Q and R formulae we have:

$(-Q)^I = -Q^I, \quad (Q\&R)^I = Q^I\&R^I$

and for Q with free variable x we have:

$(\text{for-all}x Q)^I = \text{for-all}x_B \text{ for-all}x_N \ (\text{is\_int}(x_B, x_N) \text{ -> } Q^I)$
$(\text{there-exists}x Q)^I = \text{there-exists}x_B \text{there-exists}x_N \ (\text{is\_int}(x_B, x_N) \ \& \ Q^I)$

Thus for example,

$(\text{for-all}x \text{ for-all}y \ (\text{suc}(x){=}\text{suc}(y) \text{ -> } x{=}y))^I$
$= \text{for-all}x_B \text{ for-all}x_N \text{ for-all}y_B \text{ for-all}y_N \ (\text{is\_int}(x_B, x_N) \text{ ->}$
$((\text{is\_int}(y_B, y_N) \text{ -> } ($
${=}\text{rep}(\text{sucrepB}(x_B, x_N), \text{sucrepB}(y_B, y_N), \text{sucrepN}(x_B, x_N), \text{sucrepN}(y_B, y_N))$
$\text{-> } {=}\text{rep}(x_B, y_B, x_N, y_N) \ ) \ )).$

To assure ourselves that our translation I is faithful in the sense of preserving the properties of integers as we have defined them, it is sufficient to show that the axioms A_{int} translate under I to theorems of ENBOOL. In particular, the namability axiom for integers

$\text{for-all}x \ (\ V_{z \text{ in } Z} \ x{=}z)$ translates to:
$\text{for-all}x_B \text{for-all}x_N \ (\text{is\_int}(x_B, x_N)) \text{ -> } V_{z \text{ in } Z} {=}\text{rep}(x_B, z_B{}^I, x_N, z_N{}^I))$

That is, every pair of values satisfying the relativisation predicate is_int must be equivalent to the translation of an allowed name for an integer. One usually checks that the translated namability axiom is a consequence of the target theory by some sort of inductive argument based on names in the source. (We remark that the axioms defining equality also translate to relativised equality axioms.)

In general a translation as defined by (i) - (iv) above with the property that axioms of the source are translated to theorems of the target is called an interpretation between theories [END], [SCH].

Given theories (L_i,A_i) for i=1,2,3,4 and interpretations $I_i:(L_i,A_i) \rightarrow (L_{i+1},A_{i+1})$ we have the following simple properties:

a) I_{i+1} o I_i, for i=1,2,3 exists, is defined obviously in terms of mappings defined analogously to (i) - (iv) above, and is an interpretation between theories.
b) Moreover, $I_3o(I_2oI_1)=(I_3oI_2)oI_1$. Thus composition of interpretations is associative.

Based on the above results, extensions of those found in [END], [SCH], we now proceed to give results that connect the two ideas of an extension and an interpretation of a specification.

When we say that we can implement specification S = (L,A) in terms of specification T = (M,B) - for example, INT in terms of NBOOL - we mean that we can extend S conservatively to $T_S[T]$ so that we can define an interpretation between theories $I:S \dashrightarrow T_S[T]$. Denoting conservative extension by >---> we have the following situation:

$$\begin{tikzcd} S \arrow[r, "I"] & T_S[T] \\ & T \arrow[u, tail] \end{tikzcd}$$

We might characterise the **implementation** of S in T by the pair $(I,T_S[T])$.

Using Implementations
In developing specifications we might now consider doing two things. Firstly, we might want to conservatively extend S and automatically

carry our extension over to the implementation of S by T. That is, we would like to be able to complete the diagram:

$$\begin{tikzcd}
S_1[S] \arrow[rr] & & ? \\
S_1 \arrow[u, tail] \arrow[rr, "I"] & & T_S[T] \arrow[u, tail] \\
& & T \arrow[u, tail]
\end{tikzcd}$$

in some automatic fashion.

Secondly, we might wish to use an implementation of T in terms of U to get an implementation of S in terms of U. Graphically, this can be illustrated by the following:

$$\begin{tikzcd}
S \arrow[r, "I_1"] & T_S[T] & \\
& T \arrow[u, tail] \arrow[r, "I_2"] & U_T[U] \\
& & U \arrow[u, tail]
\end{tikzcd}$$

Here $(I_2, U_T[U])$ characterises the implementation of T in U. Again we see that we require the completion of the following 'rectangle',

$$\begin{tikzcd}
T_S[T] \arrow[r, "I_2'"] & ? \\
T \arrow[u, tail] \arrow[r, "I_2"] & U_T[U] \arrow[u, tail]
\end{tikzcd}$$

to get:

$$\begin{tikzcd}
S \arrow[r, "I_1"] & T_S[T] \arrow[r, "I_2'"] & ? \\
& T \arrow[u, tail] \arrow[r, "I_2"] & U_T[U] \arrow[u, tail] \\
& & U \arrow[u, tail]
\end{tikzcd}$$

Where I_2'oI_1 and the target of I_2' characterise the implementation of S in U. We proceed to show that the missing specification in such rectangles can always be automatically constructed by proving a more general result.

We will say that the extension E: S-->S' is **conservative** exactly in the case that S' contains no theorems, stated only in the language of S, other than those which are provable from S.

Theorem

Suppose that S_1, S_2, S_3 are specifications, and suppose

$$\begin{tikzcd} S_3 & \\ S_1 \arrow[u, "E"] \arrow[r, "I"] & S_2 \end{tikzcd}$$

where E is a conservative extension, I an implementation, then there exists S_4 and I' such that:

$$\begin{tikzcd} S_3 \arrow[r, "I'"] & S_4 \\ S_1 \arrow[u, "E"] \arrow[r, "I"] & S_2 \arrow[u, "E'"'] \end{tikzcd}$$

Moreover E':S_2-->S_4 is conservative.

Proof

Briefly, S_4 and I' are obtained by extending S_2 and I as follows: For each new sort or symbol introduced in extending S_1 to S_3, add corresponding sort or symbol to S_2. The translation I now extends to translate symbols introduced into S_3 and so we can translate terms or formulae of S_3 to those of S_4. The extension of S_2 to S_4 is completed by adding to S_2 the translations I' of the axioms introduced to get S_3 from S_1. What remains to be checked is the conservativeness of S_2-->S_4 which since S_2 is consistent gives S_4 as consistent. This is a straightforward application of the Craig Interpolation Lemma [MAK], essentially copying the proof of the Robinson Consistency Theorem from the Craig Interpolation Lemma [CK].

Further:
If $E:S_1 \dashrightarrow S_2$ is not conservative then in general S_4 and I' need not exist. For suppose S_1 has predicate p and constant a and neither p(a) nor -p(a) are in Con(S_1). Suppose further that S_1 ---> S_3 and p(a) Con(S_3) and I:S_1 -> S_2 is such that I(p)=p, I(a)=a and -p(a) is in Con(S_2). Then Con(S_4) would have to contain both p(a) and -p(a) which would mean that S_4 was inconsistent.

As an example, let us implement INT in NAT. We already have an implementation of INT in NBOOL (NAT extended by BOOL) and we will now implement NBOOL in NAT. We outline the necessary details below:

nat and bool of NBOOL are both mapped to nat of NAT. To nat of NBOOL we assign the relativisation predicate is_nat of type <nat> which we then define to be the identity on nat of NAT. To bool we assign the relativisation predicate is_bool of type <nat> which we then define to be:

    is_bool (x) <-> x=zero v x=suc(zero).

So only zero and suc(zero) in NAT are used as boolean values. Thus this implementation produces 'junk'.

We then map zero, suc, $=_{nat}$ of NBOOL identically to zero, suc, $=_{nat}$ of NAT and we map T, F and $=_{bool}$ of NBOOL to the new symbols T', F', and $='_{bool}$ added to NAT respectively, and define them as follows:

    T'=x <-> x=zero   F'=x <-> x=suc(zero)
    x$='_{bool}$ <-> (x=zero & y=zero) v (x=suc(zero) & y=suc(zero))

To check that this mapping is an interpretation we must check that the axioms of NBOOL translate to theorems of this extended NAT which we call ENAT. Clearly, the axioms for natural numbers in NBOOL translate to formulae logically equivalent to the same axioms in ENAT. As for the axioms concerning BOOL, -(T$=_{bool}$ F) becomes -(T'='bool$^{F'}$). (Note that there are no relativisation predicates in the resulting formulae as there are no variables in the original.) This clearly follows from the fact that in NAT, -(zero=suc(zero).) The namability axiom for bool in NBOOL is b$=_{bool}$T v b$=_{bool}$ F). Under interpretation, this becomes: is_bool (x) -> (x='bool$^{T'}$ v x='bool$^{F'}$)). By the definitions of is_bool, $='_{bool}$, T', and F' above this is equivalent to:

    (x=zero v x=suc(zero)) -> (x=zero v x=suc(zero))

which is a tautology.

Thus we have:

$$\begin{tikzcd}
INT \arrow[r, "I_1"] & ENBOOL & \\
& NBOOL \arrow[u, tail] \arrow[r, "I_2"] & ENAT \\
& & NAT \arrow[u, tail]
\end{tikzcd}$$

and by the above theorem we can get:

$$\begin{tikzcd}
ENBOOL \arrow[r, "I_2'"] & BENAT \\
NBOOL \arrow[u, tail] \arrow[r, "I_2"] & ENAT \arrow[u, tail]
\end{tikzcd}$$

with BENAT and $I_2'oI_1$, characterising the implementation of INT in terms of NAT. The various details omitted above, are straightforward.

If we now wanted to extend INT to SINT by defining sequences of integers (as done earlier in this report for NAT) we could use the above result to carry this extension 'along' our implementation to get (unimplemented) sequences of implemented integers, SBENAT:

$$\begin{tikzcd}
SINT \arrow[r, "I_1'"] & SENBOOL \arrow[r] & SBENAT \\
INT \arrow[u, tail] \arrow[r, "I_1"] & ENBOOL \arrow[u, tail] \arrow[r, "I_2'"] & BENAT \arrow[u, tail] \\
& NBOOL \arrow[u, tail] \arrow[r, "I_2"] & ENAT \arrow[u, tail] \\
& & NAT \arrow[u, tail]
\end{tikzcd}$$

Conclusion

We feel that the theory of implementation presented above provides a simple, straightforward technical tool for reasoning about software development. The major point in its favour is the suppression of the technical flaws in earlier theories where implementations were not based on interpretations and composition was inadequately defined. The theory of specification outlined and our theory of implementation provides a wide degree of freedom in choosing implementation strategies. A structured specification can be implemented all at once into a specification which is either the same in structure or quite different from the original. Parts of the structure can be implemented independently.

Further work is in progress on a number of fronts. For example, theories are not always presented in the same formal system, as for example, the realisation of a first order specification in a conventional language like PASCAL which has a Hoare-like, modal logic. We are exploring interpretations between theories in different formal systems. We are also exploring the concept of parameterisation and defining implementations of parameterised specification. The required properties, including the commuting of implementation with parameter passing ([BG], [EK], [GAN], [PV], [SW]), turn out to be straightforwardly derivable again illustrating the suitability of our tools. Finally we are exploring the idea of 'loose implementation' as a last step to get, for example, from a specification of integers to the finite representations possible in any machine. Basically, the idea is to further restrict the relativisation predicates when we consider the translation of axioms. We partition each relativisation predicate into an interior (which respects the axioms of the source specification) and a boundary (which takes care of overflows and other boundary 'errors').

Bibliography

[BG] R M Burstall, J A Goguen. 'The Semantics of CLEAR, a Specification Language'. Proc of Advanced Course on Abstract Software Specifications, Copenhagen, LNCS86, Springer-Verlag 1980

[CK] C C Chang, H J Keisler. Model Theory. North Holland, 1977

[EHR] H-D Ehrich. 'On the Theory of Specification, Implementation and

Parameterisation of Abstract Data Types'.JACM, Vol 29, No 1, 1982

[EK] H Ehrig, H-J Kreowski. 'Parameter Passing Commutes with Implementations of Parameterised Data Types'.Proc of 9th ICALP, LNCS 140, Springer-Verlag 1982

[EKP] H Ehrig, H-J Kreowski, P Padawitz. 'Algebraic Implementation of Abstract Data Types: Concept, Syntax, Semantics and Correctness' Proc 7th ICALP, LNCS 85, Springer-Verlag, 1980

[END] H B Enderton. 'A Mathematical Introduction to Logic'. Academic Press, 1972

[GAN] H Ganzinger. 'Parameterised Specifications: Parameter Passing and Implementation'. Technical Report, Dept EECS, U Calif, Berkeley, 1980. To appear TOPLAS

[GTW] J A Goguen, J W Thatcher, E G Wagner. 'An Initital Algebra Approach to the Specification Correctness, and Implementation of Abstract Data Types'. In R T Yeh (Ed) 'Current Trends in Programming Methodology Vol IV' Prentice Hall, 1978

[GH] J V Guttag, J J Horning. 'The Algebraic Specification of Abstract Data Types'. Acta Informatica, Vol 10, No 1, 1978.

[KMS] S Khosla, T S E Maibaum, M Sadler. 'Database Specification'. Dept. Report, Imperial College, London, 1984

[LZ] B Liskov, S Zilles. 'Specification Techniques for Data Abstraction'.IEEE Trans. Software Eng. Vol SE-1, No 1, 1975

[MV] T S E Maibaum, P A S Veloso. 'A Logical Approach to Abstract Data Types'.Technical Report, Dept of Computing, Imperial College,1981 (To appear in Science of Computer Programming)

[MAK] M Makkai. 'Admissible Sets and Infinitary Logic'.Handbook of Mathematical Logic. North Holland, 1977

[NOU] F Nourani. 'Constructive Extension and Implementation of Abstract Data Types and Algorithms'. PhD thesis, Dept of Computer Science, UCLA, 1979

[PV] A Poigne, J Voss. 'Programs over abstract Data Types - On the Implementations of Abstract Data Types'. Draft Technical Report, University of Dortmund, 1983

[SCH] J R Shoenfield. 'Mathematical Logic'.Addison Wesley, 1967.

[SW] D Sanella, M Wirsing. 'Implementation of Parameterised Specifications'. Proc 9th ICALP, LNCS 140, Springer-Verlag, 1982

[WB] M Wirsing, M Broy. 'An Analysis of Semantic Models for Algebraic Specifications'. International Summer School on Theoretical Foundations of Programming Methodology, Marktoberdorf, Technical Report, Technical University, Munich, 1981