SPECIFICATIONS, MODELS, AND IMPLEMENTATIONS OF DATA ABSTRACTIONS

by

Mitchell Wand

Computer Science Department

Indiana University

Bloomington, Indiana 47405

TECHNICAL REPORT NO. 88

SPECIFICATIONS, MODELS, AND IMPLEMENTATIONS OF DATA ABSTRACTIONS

MITCHELL WAND

MARCH 1980, (REVISED, FEBRUARY, 1981)

IN MEMORY OF CALVIN ELGOT

To appear in: Theoretical Computer Science

This material is based upon work supported by the National Science Foundation under Grants MCS75-06678A01 and MCS79-04183.

SPECIFICATIONS, MODELS, AND IMPLEMENTATIONS OF DATA ABSTRACTIONS

Mitchell Wand

Abstract:

We consider the specification and verification of modules in hierarchically structured programs, as proposed by Parnas and Hoare. We argue that a specification for such a module is a set of sentences in some logical language in which the names to be exported by the module appear as nonlogical symbols. We further argue that an implementation of one module in terms of another module is a translation of the nonlogical symbols of the first specification into the language of the second. Equality must also be interpreted. We propose necessary conditions which any such notion of "correct implementation" ought to satisfy. These criteria provide a basis for judging the logical adequacy of any proposed specification language and definition of implementation. We then study DLP, a specification language obtained by adding uninterpreted procedure symbols to Pratt's first order dynamic logic. We present a definition of "implementation" for DLP, and we show it satisfies these conditions. The main theorem, called the Implementation Theorem, extends the Interpretation Theorem from first-order logic to DLP. The proof of this theorem is complicated by the necessity of dealing with modalities, parameters to procedures, interpretations of equality, and interpretations of sorts as tuples.

0. Introduction

The purpose of this paper is to present some mathematical results in support of a thesis concerning the nature of specifications, models, and implementations of so-called "abstract data types" and other modules in hierarchically organized programs.

In Section 1, we propose necessary conditions which any such notion of "correct implementation" ought to satisfy. This methodology is largely independent of one's choice of specification language. In Section 2, we present the syntax and semantics of a many-sorted first-order Dynamic Logic [14, 29] enriched by uninterpreted procedure symbols. We refer to this logic as DLP. Section 3 is devoted to the issue of free and bound variables. Our notion of implementation in DLP is divided into two stages: (a) substitution of phrases for nonlogical symbols, and (b) introduction of sorts for record types. Section 4 deals with the former stage, and Section 5 with the latter. Section 6 summarizes these results by presenting our main theorem, the Implementation Theorem for DLP. The theorem extends the Interpretation Theorem for conventional first-order Logic [33, p. 62] both in its treatment of the modal operators in dynamic logic, and by allowing equality to be interpreted as an arbitrary equivalence relation.

1. Methodology

Our concern is the logical treatment of data abstraction. Data abstraction is a phenomenon which may be observed when a programmer explaining his program says either of the following:

(1) "that's not just an array and a counter; it's really a stack."

(2) "that's not really a stack; it's just an array and a counter."

While most languages support the second view though procedural abstractions, the first view is somewhat more subtle. In the first view, the purpose of a data abstraction is to build a machine with a certain behavior (in this case a stack) and then to forget how it was built. Parnas called this "information-hiding." To control the hiding of information, Parnas proposed the use of specifications, which were to be independent objects which would serve as interfaces between modules [28]. This idea, wedded to language mechanisms based on SIMULA [3], remains the most important idea in modern work on data abstractions [21, 38].

Consider a specification of a module. A programmer who writes a program which uses the module interacts with the implementation of the module only through the specification. That is, the program which uses that module must work correctly with any implementation of the module that satisfies the specifications.

A specification is therefore a formal statement of those properties of a module on which a potential user may rely. From such a statement, the user must be able to prove the correctness of his program. The statement may be true of an alleged implementation or false of it. These requirements suggest that a specification should be a formula or set of formulas in some <u>logic</u>.

Our next observation is that a specification can describe a module only in terms of the <u>names</u> of the procedures which the implementation of the module will make available to the user.* For this we advance two arguments. First, the specification is supposed to exist before the module is implemented; therefore, the specification can include the names but not the procedures which they denote, since the procedures do not yet exist at the time the specification is written. Second, the specification is supposed to isolate the user from the actual procedures. The actual procedure corresponding to the name P is presumed to vary among different implementations of the module. The specification

*The module may, of course, export functions, predicates, constants as well as procedures. To simplify the argument, we use the term "procedure" in this argument to mean any object exported by a module.

tells the user that if he calls on the implemented module to perform the procedure named P, then certain observable effects will follow, regardless of which implementation of the module is actually employed. The specification acts as a constraint on the behavior of the procedures whose names appear in the specification.*

In logic, one often deals with formulas containing symbols whose meaning is unknown but constrained by the formulas. Such symbols are called <u>undefined terms</u> or <u>non-logical symbols</u> [1, p. 57; 33, p. 14; 36, p. 114*f*]. In geometry, for example, the terms "point" and "line" are among the undefined terms. The Euclidean axioms give an initial set of constraints on the possible meanings of the terms "point" and "line." By formal proofs, one can derive from

---

*This discussion is not meant to eliminate specifications of the form "the procedure named P behaves like the following mathematical function ...." Such specifications are produced by modeling techniques which are "representational" rather than "implicit" [22]. This specification is still a statement about the behavior of the procedure whose <u>name</u> is P. Often, such specifications are difficult to interpret because the meaning of "behaves like" has not been sufficiently delineated.

these axioms additional constraints (theorems) on the meanings of these terms.

Similarly, a specification gives an initial set of constraints on the possible meanings of the procedure names which appear in the specification. Thus we can state the first part of our thesis as follows:

> Thesis A. A specification is a set of sentences in some logical language. The names of the functions, predicates, and procedures which the specification is intended to specify appear as nonlogical symbols in these sentences.

As an illustration, consider a specification for a bounded stack of integers of size 100. For the specification language, we choose DLP, which is Dynamic Logic extended by procedure symbols. This is the logic to be developed in this paper; the formulas should be clear to anyone familiar with Dynamic Logic [14, 29]. Our arguments are, however, largely independent of one's choice of specification language; we hope this independence will be apparent.

We will have two sorts of variables, called int and stk. Some reasonable portions of the specification might be:

$(\forall s{:}stk)([init(s)](length(s) = 0))$ (1)

$(\forall s{:}stk)(\forall s_0{:}stk)(\forall n{:}int)(length(s_0) < 100 \supset$
$[push(s;n,s_0);pop(s;s)](s=s_0))$ (2)

$(\forall s{:}stk)(\forall t{:}stk)(length(s)=0 \supset [pop(t;s)]false)$ (3)

$(\forall s{:}stk)(\forall t{:}stk)(length(s) > 0 \supset \langle pop(t;s)\rangle true)$ (4)

We call the set of sentences comprising the specification T_s, the <u>theory</u> of bounded stacks.

Here we are specifying:

length	a function from stacks to integers
init(s)	a procedure which sets s to be the empty stack
push(s;n,s')	a procedure which sets s to be the stack obtained by pushing the integer n onto stack s'
pop(s;s')	a procedure which sets to be the stack obtained by popping one element off s'

The first axiom states that the empty stack should have length 0. The second says that pushing any integer onto a stack s_0 and then popping the stack should leave the stack unchanged. (In these calls a semicolon separates output (result) parameters from input (value) parameters. The third axiom states that an attempt to pop the empty stack should always fail to terminate. The fourth axiom states that an attempt to pop a non-empty stack should always succeed.

(For a finer analysis, see [15]). The last two axioms demonstrate the power of DL to discuss error conditions. Our reasons for choosing DL as the basis of a specification language will be discussed further in Section 7.

A <u>structure</u> A for T_s will consist of two non-empty sets (one for integers and one for stacks), and functions, predicates, and procedures corresponding to the nonlogical symbols appearing in T_s. Such a structure is a <u>model</u> of T_s iff every formula of T_s is true in the structure. For example, we can let the sets be

A_{int}	$= \omega$	(the nonnegative integers)
A_{stk}	$= \omega^*$	(all finite strings of nonnegative integers)

and the single function be

$$\text{length}^A(x) = |x|.$$

We associate with each procedure an input-output relation on states:

$$[\![\text{init}(s)]\!]^A = \{(\rho,\rho')\,|\,\rho'(s) = \Lambda \ \&\ (\forall v)(v\neq s \supset \rho(v) = \rho'(v))\}$$

$$[\![\text{push}(s;n,s')]\!]^A = \{(\rho,\rho')\,|\,(\rho(s')=n_1\ldots n_k \supset \rho'(s)=nn_1\ldots n_k) \ \&(\forall v)(v\neq s \supset \rho(v) = \rho'(v))\}$$

$$[\![\text{pop}(s;s')]\!]^A = \{(\rho,\rho')\,|\,(\exists k)(k \geq 1 \ \&\ \rho(s') = n_1\ldots n_k \ \&\ \rho'(s) = n_2\ldots n_k \ \&\ (\forall v)(v\neq s \supset \rho(v) = \rho'(v)))\}$$

This structure A is a model of formulas (1) - (4) and would probably be a model of all of T_s had we written it out. We write $A|=T_s$ when A is a model of T_s.

If T is any theory (set of sentences), we write T|=G if the formula G is true in any model of T. (The details for DLP will be discussed in Section 2). If T' is also a theory, we write T|=T' if T|=G for each $G \in T'$.

A reasonable implementation of T_s might represent a stack s as a record

```
s.arr:  array [1..100] of integer;
s.k  :  0..100
```

Here s.k is to represent the length of the stack. We might implement some of the procedures as follows:

```
[init(s)] = [s.arr := (an array of 100 0's); s.k := 0]
[pop(s;s')] = [(s'.k > 0)?; s.arr := s'.arr; s.k:=(s'.k)-1]
[push(s;n,s')] = [(s'.k) < 100)?;
                    s.arr := update(s'.arr, s'.k+1, n);
                    s.k:= (s'.k) + 1]
length(s) = s.k  .
```

We claim that this is a reasonable implementation of T_s. (It looks even more reasonable for the usual case where the calls on push and pop are always of the forms push(s;n,s) or pop(s;s)). But it is not a model of T_s, because it makes formula (2) false. Let s_o be the stack created by <u>init</u> and let $n = 2$. After executing $[push(s;2,s_o);pop(s,s)]$, the value of s is

$<(2\ 0\ 0\ 0\ ..),\ 0>$

whereas s_o is

$<(0\ 0\ 0\ 0...),\ 0>$

So $s \neq s_o$.

Lehmann and Smyth [20] claim this as a defect of algebraic specification methods, but it evidently can arise with any specification languages, because there may be more than one representation of a stack [16]. We can remedy the situation if we define an equivalence relation $\approx$ on stack representations by:

$$<s.arr,\ s.k> \approx <s'.arr,\ s'.k> \text{ iff}$$
$$s.k = s'.k\ \&\ (\forall i)(1 \leq i \leq s.k \supset s.arr[i] = s'.arr[i])$$

then formula (2) is true in the implementation with equality replaced by this equivalence.

But in what sense is formula (2), or some variant of it, "true in the implementation?" The implementation might have been expressed in terms of some other "abstract data types" or modules whose implementation is unknown. In our example, the arrays in the implementation of stacks might be regarded as an abstract data type. Thus an implementation involves two theories (specifications): the implemented theory (in this case T_s, the theory of stacks) and the <u>implementing</u> theory (in this case T_{arr}, the theory of arrays).* (See Figure 1.1).

*There may, of course, be more than one implementing theory [34].

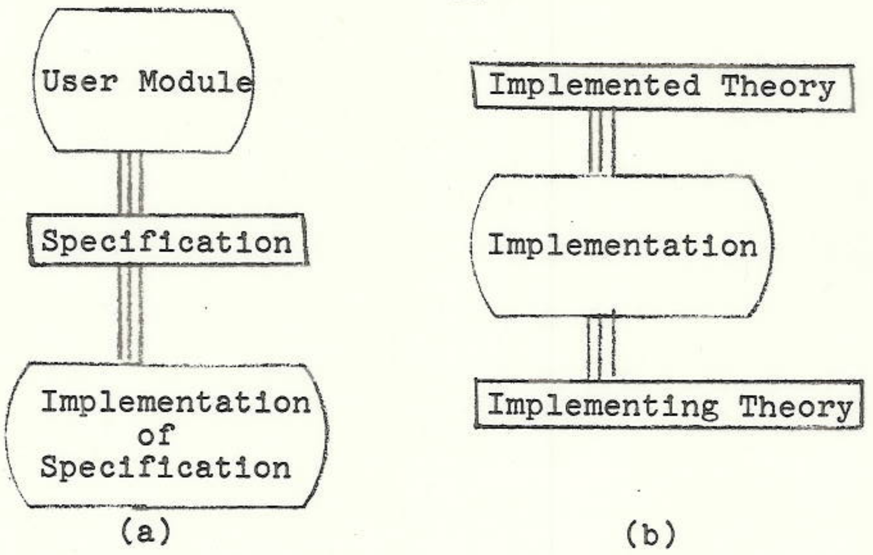

Figure 1.1 (a) A specification (theory as an interface between modules
(b) An implementation as an interface between two theories.

In this terminology, our implementation provided a translation from the language of the implemented theory to the language of the implementing theory. Each procedure symbol was translated into a program (a modality) in the implementing theory. In addition, the equality symbol was translated into the equivalence relation $\simeq$. To prove the correctness of our implementation, we must prove our translated formulas. Since the translated formulas involve calls on an abstract data type, (arrays, in this case) the best we can do is to derive them from the specifications for arrays. If $T_s{}^I$ denotes the translated formulas, we need to show:

$$T_{arr} \models T_s{}^I.$$

We may now state the second portion of our thesis:

<u>Thesis B</u>. An implementation I of a theory T_1 is a <u>translation</u> of the nonlogical symbols of T_1 and of equality into the language of the implementing theory T_2.

When is an implementation correct? The preceding development suggests the hypothesis that translated formulas be consequences of T_2: symbolically, $T_2 \models T_1{}^I$. This is the definition adopted by [32] and [12, <u>inter alia</u>]. This condition seems necessary, but is it sufficient? The two quotations which began this section give us some clues.

The first quotation suggests a synthetic view: given an array and a counter, we should be able to perceive a stack. Mathematically, given a model M of T_2, we should be able to construct a model M' of T_1. We would like some connection between the models, of course.* One such connection would be a surjective map $J:M \to M'$ (Hoare's abstraction function [16]). Another connection might insist that M' and M "behave similarly." For example, if G is any formula with no free variables, then

$$M' \models G \quad \text{iff} \quad M \models G^I.$$

This criterion is useful if T_1 is incomplete. For example, if T_1 did not specify what to do with an error condition, we could determine what M' did by examining the behavior of M. (This is a common real-world reason for opening up a black box).

*This behavior is familiar in algebra: a morphism between algebraic theories $T_1 \to T_2$ (a syntactic map) induces a forgetful functor (a semantic construction) from the category of T_2-algebras to the category of T_1-algebras.

The second quotation is analytic: given a stack, one should be able to think about the underlying array. If we reason about the implemented theory T_1, we should be able to draw conclusions about the implementation. For example, if we deduce that at the end of some program α, the stack does not underflow, then we should be able to predict that at the end of the implemented program, the counter does not underflow. Mathematically, if $T_1 \models G$, then $T_2 \models G^I$. (The converse is not a realistic expectation, since T_1 is usually incomplete: an implementation typically is required to make decisions about issues left unspecified in the original specification).

Joining these we get the following goal:

Theorem (The Implementation Theorem). Let I be a correct implementation of T_1 in T_2. Then

(i) (synthetic version) if M is any model of T_2, then there is a model M' of T_1 such that for any closed formula G in the language of T_1, $M' \models G$ iff $M \models G^I$.

(ii) (analytic version) for any formula G in the language of T_1, if $T_1 \models G$, then $T_2 \models G^I$.

This is a theorem which should hold for any reasonable notion of specification language and correct implementation.

We believe that the use of specifications as a tool for information hiding and of implementation as translation is a naturally occuring phenomenon. Consider a specification for a GCD module. We implement the specification by writing a GCD program in PASCAL, which is translated by the PASCAL compiler into P-code, which is translated into machine code, which is translated by the digital architecture into actions of registers and busses... .

Each such translation is typically called an "implementation" of the preceding level. At every level the implementation forgets what is involved both above and below the translation. A subtle but important shift in paradigm has occurred: *Instead of regarding a specification as an interface between two modules, we now regard a module as an interface between two specifications.** (Figure 1.1)

In the remainder of this paper, we prove an implementation theorem for a particular specification language which we call DLP.

---

*For comparison, note two other shifts in paradigm which have been exceedingly influential: In 1972, Dijkstra urged us to stop writing programs to run on our machines and start building machines to execute our programs [4, pp. 48-49]. Similarly, we have stopped regarding a loop invariant as a description of the loop's action; instead, we believe the purpose of the loop body is to maintain an invariant. [e.g. 5]

2. The Syntax and Semantics of DLP

2.1 Syntax

We are concerned with two sets of strings, called formulas and programs, which are built from two classes of symbols, the logical symbols, which are fixed, and the non-logical symbols, which may vary between different specifications. A particular choice of non-logical symbols is called a language [33, p. 14]. The non-logical symbols are classified into five disjoint classes:

(a) sort symbols: σ, σ_1, τ,... (usually, but not necessarily, a finite set)

(b) function symbols: f, g, h...; each function symbol has a signature* $<\sigma_1,\ldots,\sigma_n> \to \sigma$ where $n \geq 0$ and $\sigma_1,\ldots,\sigma_n,\sigma$ are sort symbols. (A constant symbol of sort σ is a function symbol of signature $<> \to \sigma$).

(c) predicate symbols: p, q, r...; each predicate symbol has a signature $<\sigma_1,\ldots,\sigma_n>$ where $n \geq 0$ and $\sigma_1,\ldots,\sigma_n$ are sort symbols. For each sort symbol σ there is a distinguished predicate symbol $=_\sigma$ of signature $<\sigma,\sigma>$.

(d) individual variable symbols: x,y,z,u,v,...; each individual variable symbol has a sort σ, where σ is a sort symbol. We assume there are infinitely many individual variable symbols of each sort.

*We use the word "signature" to avoid the already overloaded word "type." Generally, sorts are atomic (single symbols) and signatures are composite (strings).

(e) <u>procedure</u> <u>symbols</u>: A, B, C...; each procedure symbol has a signature "$<\sigma_1,\ldots,\sigma_n>:=<\tau_1,\ldots,\tau_m>$," where $n, m \geq 0$ and $\sigma_1,\ldots,\sigma_n,\tau_1,\ldots,\tau_m$ are sort symbols. For each sort symbol σ, there are distinguished procedure symbols assign_σ, with signature $<\sigma>:=<\sigma>$, and forall_σ, with signature $<\sigma>:=<>$.

We write $f:<\sigma_1,\ldots,\sigma_n> \to \sigma$ if f has signature $<\sigma_1,\ldots,\sigma_n> \to \sigma$; intuitively, the function denoted by f is to take n arguments of sorts $\sigma_1,\ldots,\sigma_n$, and is to return an answer of sort σ. We use similar notation for the other nonlogical symbols.

The set of <u>terms</u> is built up from the individual variable symbols by use of the function symbols in the usual way, subject to the constraint that the sorts must agree.

If $t_1,\ldots,t_n$ are terms of sorts $\sigma_1,\ldots,\sigma_n$, and p is a predicate symbol of sort $<\sigma_1,\ldots,\sigma_n>$, then $pt_1\ldots t_n$ is an <u>atomic</u> <u>formula</u>.

If $v_1,\ldots,v_n$ are individual variable symbols of sorts $\sigma_1,\ldots,\sigma_n$, and $t_1,\ldots,t_m$ are terms of sorts $\tau_1,\ldots,\tau_m$ and A is a procedure symbol of signature

$<\sigma_1,\ldots,\sigma_n>:=<\tau_1,\ldots,\tau_m>$, then $A(v_1,\ldots,v_n;t_1,\ldots,t_m)$ is an <u>atomic</u> <u>program</u>. Intuitively, the first n parameters are the parameters which are assigned by A (the "output" parameters); hence the requirement that the actual parameters be variables rather than terms. The remaining m parameters are the input parameters; these may be any terms of the correct sorts.

The atomic programs $\text{assign}_\sigma(v;t)$ and $\text{forall}_\sigma(v)$ are usually written as $v{:=}t$ and $\forall v{:}\sigma$ respectively. (For the sense in which a universal quantifier may be viewed as a program, see [29]).

The sets of formulas and atomic programs may now be defined by a simultaneous induction, which we express in a BNF-like format. Here G and H range over formulas and α and β range over programs.

Definition:

Formula ::= Atomic Formula | $G\&H$ | $G \vee H$ | $\sim G$ | $G \supset H$ | $[\alpha]G$

Program ::= Atomic Program | $\alpha;\beta$ | $\alpha \cup \beta$ | α^* | $G?$

The grammar is ambiguous; conflicts may be resolved by parentheses or by the following precedence table:

$\supset \ \vee \ \& \ \sim \ [\alpha]$

$\cup \ ; \ ^*$

evaluate last	evaluate first

Repeated operators are assumed to associate to the left. $[\forall v{:}\sigma]G$ is typically written $(\forall v{:}\sigma)G$. $\langle\alpha\rangle G$ abbreviates $\sim[\alpha]\sim G$. $(\exists x{:}\sigma)G$ abbreviates $\sim(\forall x{:}\sigma)\sim G$.

Formulas (1) - (4) of Section 1 are typical formulas of DLP.

2.2 Semantics

The semantics of DLP is developed in two stages:

(i) a <u>structure</u> is defined as an assignment of meanings to a set of non-logical symbols.

(ii) these meanings are extended to give formulas and programs meanings in the structure. In particular, we get the notion of truth in a structure. This is the standard way in which Tarskian semantics proceeds [9, pp. 81-82]. Without further commentary, we proceed with the construction.

<u>Definition</u> A <u>structure</u> A is given by the following data:

(a) for each sort symbol σ, a nonempty set U_σ. We denote by U the union of the sets U_σ as σ ranges over the sort symbols. U_σ is called the <u>carrier</u> of sort σ.

(b) for each function symbol $f\colon <\sigma_1,\ldots,\sigma_n> \to \sigma$, a function: $f^A\colon U_{\sigma_1} \times\ldots\times U_{\sigma_n} \to U_\sigma$

(c) for each predicate symbol $p{:}<\sigma_1,\ldots,\sigma_n>$, a predicate p^A on $U_{\sigma_1}\times\ldots\times U_{\sigma_n}$, such that the predicate $=_\sigma^A$ (corresponding to the distinguished predicate symbol $=_\sigma$) is the equality predicate on $U_\sigma \times U_\sigma$ (*)

(d) for each procedure symbol $A{:}<\sigma_1,\ldots,\sigma_n>{:=}$ $<\tau_1,\ldots,\tau_m>$, a predicate p_A^A on $U_{\sigma_1}\times\ldots\times U_{\sigma_n}\times U_{\tau_1}\times\ldots\times U_{\tau_m}$, such that the predicate corresponding to the assignment

---

*given by $(x,y) \mapsto \begin{cases} \text{TRUE} & \text{if } x = y \\ \text{FALSE} & \text{otherwise} \end{cases}$

symbol assign_σ is the equality predicate on $U_\sigma \times U_\sigma$, and the predicate corresponding to the symbol forall_σ is the predicate on U_σ which is always true.

The way in which a predicate p_A^A, corresponding to a procedure symbol, gives meaning to a procedure call (an atomic program) will be explained below.

Before proceeding with the definition of truth in a structure, we need the notion of a <u>state</u>:

<u>Definition</u>: A <u>state</u> ρ is a function from the set of individual variable symbols to U which is <u>sort-preserving</u> in the sense that if v is an individual variable symbol of sort σ, then $\rho(v) \in U_\sigma$.

We now define the semantics of a term in a structure. We usually write the evaluation map $\models$ in infix notation.

<u>Definition</u>: $\models$ is the map States x Terms $\to$ U defined as follows:

(i) if x is an individual variable symbol of sort σ, then $(\rho \models x) = \rho(x)$.

(ii) if $t_1,\ldots,t_n$ are terms of sorts $\sigma_1,\ldots,\sigma_n$, and f is a function symbol of signature $\langle\sigma_1,\ldots,\sigma_n\rangle \to \sigma$, then

$\rho \models ft_1\ldots t_n = f^A(\rho \models t_1,\ldots,\rho \models t_n)$.

This is just the usual notion of the meaning of a term in a first-order structure [33].

We next define a predicate $\models$ on States x Formulas (truth in a state of a structure) and a function $[\![\ldots]\!]$:

Programs $\rightarrow 2^{\text{States} \times \text{States}}$ (the input-output relation of a program).* Since formulas and programs are defined by mutual recursion, these two constructs are likewise defined by mutual recursion. We write $\models$ in infix.

<u>Definition</u>: $\models$ and $[\![\ldots]\!]$ are defined as follows:

(1) if $pt_1 \ldots t_n$ is an atomic formula, then

$$\rho \models pt_1 \ldots t_n \text{ iff } p^{A}(\rho \models t_1, \ldots, \rho \models t_n)$$

(2) if $A(v_1, \ldots, v_n; t_1, \ldots, t_m)$ is an atomic program then

$[\![A(v_1, \ldots, v_n; t_1, \ldots, t_m)]\!] =$

$$\{(\rho,\rho') | p_A{}^{A}(\rho'(v_1), \ldots, \rho'(v_n), \rho \models t_1, \ldots, \rho \models t_m)$$
$$\& (\forall w)(w \notin \{v_1, \ldots v_n\} \supset \rho(w) = \rho'(w))\}$$

(3) $\rho \models G\&H$ iff $\rho \models G$ and $\rho \models H$

(4) $\rho \models G \vee H$ iff $\rho \models G$ or $\rho \models H$

(5) $\rho \models \sim G$ iff not $\rho \models G$

(6) $\rho \models G \supset H$ iff $\rho \models \sim G$ or $\rho \models H$

(7) $\rho \models [\alpha]G$ iff $(\forall \rho')((\rho,\rho') \in [\![\alpha]\!] \supset \rho' \models G)$

(8) $[\![\alpha;\beta]\!] = \{(\rho,\rho'') | (\exists \rho')((\rho,\rho') \in [\![\alpha]\!] \text{ and } (\rho',\rho'') \in [\![\beta]\!])\}$

(9) $[\![\alpha \cup \beta]\!] = [\![\alpha]\!] \cup [\![\beta]\!]$

(10) $[\![\alpha^*]\!]$ = the reflexive, transitive closure of $[\![\alpha]\!]$

(11) $[\![G?]\!] = \{(\rho,\rho) | \rho \models G\}$

Of these clauses, only clause (2) merits comment. A procedure A establishes the relation $p_A{}^{A}$ between its outputs

*This open-faced bracket comes from [29]; though similar notation is often used for syntactic arguments in denotational semantics [35], that should cause no confusion here.

and its inputs.* In the atomic program $A(v_1,\dots,v_n;t_1,\dots,t_m)$, beginning execution in state ρ, the evaluated actual input parameters are $\rho|=t_1,\dots,\rho|=t_m$, and, if the output state is ρ', the outputs are $\rho'(v_1),\dots,\rho'(v_n)$ (we could equally well have written $\rho'|=v_1$,etc.) Thus the relation to be established is

$$p_A{}^{\mathcal{A}}\ (\rho'(v_1),\dots,\rho'(v_n);\rho|=t_1,\dots,\rho|=t_m)$$

We often use a semicolon to distinguish input and output values, as we have done here. Furthermore, the atomic program $A(v_1,\dots,v_n;t_1,\dots,t_m)$ may alter no variables other than $v_1,\dots,v_n$. This restriction is enforced by the second conjunct in clause (2). We write $\mathrm{Equiv}(\rho,\rho',\{v_1,\dots,v_n\})$ for this conjunct.

<u>Proposition</u> <u>2.2.1</u> $[\![\mathrm{assign}_\sigma(v,t)]\!] = \{(\rho,\rho')|\rho'(v) = (\rho|=t)$ $\&\ (\forall w)(w \neq v \supset \rho(w) = \rho'(w))\}$. $\Box$

<u>Proposition</u> <u>2.2.2</u> $[\![\mathrm{forall}_\sigma(v)]\!] = \{(\rho,\rho')|(\forall w)(w \neq v \supset$ $\rho(w) = \rho'(w))\}$. $\Box$

<u>Definition</u>: G is <u>true</u> in structure $\mathcal{A}$ iff $(\forall\rho)(\rho|=G)$. We write $\mathcal{A} \models G$. G is <u>valid</u> iff it is true in every structure for the language of G. If G is valid, we write $\models G$.

*We place the output parameters first out of deference to the conventional notation for assignment [19].

A theory is a set of formulas. We write $\rho \models T$ iff $\rho \models G$ for every $G \in T$. We say T logically implies H (we write $T \models H$) iff for every structure A, if $A \models T$ then $A \models H$.

Note that a structure for DLP is a model of First Order Dynamic Logic; the validity problem for DLP is likewise Π_1^1-complete.

3. Free and Bound Variables

In this section, we shall consider the question of free and bound variables in DLP. This question is normally entwined with the problem of substitution and renaming of variables. Here, however, we are concerned with a more restricted application: on what variables does a program or formula depend, and which variables does a program set? (We have also developed an adequate treatment of substitution, which is unfortunately beyond the scope of this paper).

Definition. For each term, program, or formula, we define its free variables and bound variables as follows:

phrase	*f*v	bv
term t	all variables in t	$\emptyset$
$A(v_1,\ldots,v_n;t_1,\ldots,t_m)$	$fv(t_1)\cup\ldots\cup fv(t_m)$	$\{v_1,\ldots,v_n\}$
$\alpha;\beta$	$fv(\alpha)\cup(fv(\beta)-bv(\alpha))$	$bv(\alpha)\cup bv(\beta)$
$\alpha\cup\beta$	$fv(\alpha)\cup fv(\beta)$	$bv(\alpha)\cap bv(\beta)$
α^*	$fv(\alpha)$	$\emptyset$
G?	$fv(G)$	$\emptyset$
$pt_1\ldots t_n$	$fv(t_1)\cup\ldots\cup fv(t_n)$	$\emptyset$
$G \otimes H$ ($\otimes$ any boolean operator)	$fv(G)\cup fv(H)$	$\emptyset$
$[\alpha]G$	$fv(\alpha)\cup(fv(G)-bv(\alpha))$	$\emptyset$

A variable, according to this definition, is bound iff it is guaranteed to be set, regardless of which alternative is chosen. Note that in $[(x:=y)^*](x^2 \geq 0)$, both x and y are free, since (intuitively) if zero iterations of (x:=y) are executed, x is not set. This is assured in the definition by the clause $bv(\alpha^*) = \emptyset$. A variable may be both free and bound in a program e.g. y in (x:=y; y:=0).

Our definition of free and bound will be adequate for our purposes if we can prove that whenever ρ_1 and ρ_2 agree on $fv(G)$, then $\rho_1 \models G$ iff $\rho_2 \models G$. For the behavior of programs, however, we shall need some finer information. If ρ_1 and ρ_2 agree on $fv(\alpha)$, and $(\rho_1,\rho_1') \in [\![\alpha]\!]$, then we should be able to make "the same choices" starting at ρ_2, and we come out at a ρ_2' such that $(\rho_2,\rho_2') \in [\![\alpha]\!]$ and ρ_2' agrees with ρ_1' on $bv(\alpha)$. We proceed by induction on the depth of modalities which appear in tests.

<u>Definition</u>: A program is <u>non-branching</u> iff it is of the form $\alpha_1;\ldots;\alpha_n$, where the α_i are atomic programs or tests.

<u>Definition</u>: A program is of <u>class</u> 0 if its tests contain no modalities (programs); it is of <u>class</u> k + 1 iff its tests contain only programs of class k or less. A formula is of <u>class</u> k iff it contains only programs of class k.

Note that every program is of some finite class.

<u>Lemma 3.1</u>. If α is non-branching, $v \notin bv(\alpha)$, and $(\rho,\rho') \in [\![\alpha]\!]$, then $\rho(v) = \rho'(v)$. $\square$

<u>Lemma 3.2</u>. If α is non-branching and of class 0, and $(\rho_1,\rho_1') \in [\![\alpha]\!]$, and ρ_1 and ρ_2 agree on $fv(\alpha)$, then there

is a state ρ_2' such that $(\rho_2,\rho_2') \in [\![\alpha]\!]$, ρ_2' agrees with ρ_1' on $bv(\alpha)$, and ρ_2' agrees with ρ_2 on all other variables.

<u>Proof:</u> We need to consider atomic procedures, tests, and sequences. For atomic programs, note that the conditions determine ρ_2' uniquely; a straightforward calculation, using the definition of the semantics of an atomic program α, shows that if $(\rho_1,\rho_1') \in [\![\alpha]\!]$, then $(\rho_2,\rho_2') \in [\![\alpha]\!]$ also. Tests are also simple, since they contain no modalities.

It remains to consider sequences $\alpha;\beta$. Let $(\rho_1,\rho_1') \in [\![\alpha;\beta]\!]$. Then there is a state ρ_1'' such that $(\rho_1,\rho_1'') \in [\![\alpha]\!]$ and $(\rho_1'',\rho_1') \in [\![\beta]\!]$. By the induction hypothesis for α, there is a state ρ_2'' such that $(\rho_2,\rho_2'') \in [\![\alpha]\!]$ and ρ_2'' agrees with ρ_1'' on $bv(\alpha)$ and with ρ_2 everywhere else. We claim that ρ_2'' agrees with ρ_1'' on $fv(\beta)$. If $v \in fv(\beta) \cap bv(\alpha)$, then $\rho_2''(v) = \rho_1''(v)$. If $v \in fv(\beta) - bv(\alpha)$, then $v \in fv(\alpha;\beta)$, so $\rho_2''(v) = \rho_2(v) = \rho_1(v) = \rho_1''(v)$ as before. So ρ_2'' agrees with ρ_1'' on $fv(\beta)$. By the induction hypothesis for β, there is a ρ_2' such that $(\rho_2'',\rho_2') \in [\![\beta]\!]$ and ρ_2' agrees with ρ_1' on $bv(\beta)$ and with ρ_2'' elsewhere. In particular, if $v \in bv(\alpha) - bv(\beta)$, then $\rho_2'(v) = \rho_1''(v) = \rho_1'(v)$. So ρ_2' agrees with ρ_1' on $bv(\alpha) \cup bv(\beta)$. This completes the proof. □

<u>Theorem 3.1</u> (i) if ρ_1 and ρ_2 agree on $fv(G)$, then $\rho_1 \models G$ iff $\rho_2 \models G$.

(ii) if ρ_1 and ρ_2 agree on $fv(\alpha)$, and $(\rho_1, \rho_1') \in [\![\alpha]\!]$, then there is a state ρ_2' such that $(\rho_2, \rho_2') \in [\![\alpha]\!]$ and ρ_2' agrees with ρ_1' on $bv(\alpha)$.

<u>Proof</u>: By induction on the class k of α and G. First, note that any program α of class k can be decomposed into a countably infinite set of non-branching programs α_i, all of class k, such that

$$[\![\alpha]\!] = \bigcup [\![\alpha_i]\!]$$
$$fv(\alpha) = \bigcup fv(\alpha_i)$$
$$\text{and} \quad bv(\alpha) = \bigcap bv(\alpha_i),$$

by converting α^* to $\bigcup_n \alpha^n$ and distributing. The α_i are all of class k since tests are not affected.

Now, assume the theorem holds for all $m < k$. We first consider (ii). Decompose α into $\bigcup \alpha_i$. The proof of (ii) is then the same as in Lemma 3.2, except that the case for tests is replaced by an appeal to induction hypothesis (i), and we must note that $fv(\alpha_i) \subseteq fv(\alpha)$ and $bv(\alpha) \subseteq bv(\alpha_i)$.

Having established (ii) at k, we consider formulas of class k, and proceed by structural induction. As usual, the interesting case is $[\alpha]G$. Assume ρ_1 and ρ_2 agree on $fv([\alpha]G)$. Decompose α into $\bigcup \alpha_i$. Now $\rho_1 \models [\alpha]G$ iff for all i, $\rho_1 \models [\alpha_i]G$. So $\rho_1 \models \sim[\alpha]G$ iff there exists an i and a ρ_1' such that $(\rho_1, \rho_1') \in [\![\alpha_i]\!]$ and $\rho_1' \models \sim G$. Since α_i is of class k, result (ii) applies, so there is a ρ_2'

such that $(\rho_2, \rho_2') \in [\![\alpha_1]\!]$ and ρ_2' agrees with ρ_1' on $bv(\alpha_1)$. We claim that ρ_2' and ρ_1' agree on all of $fv(G)$. If $v \in fv(G) - bv(\alpha)$, then $v \in fv(G) - bv(\alpha_1)$ (since $bv(\alpha_1) \subseteq bv(\alpha)$), so

$$\begin{aligned} \rho_2'(v) &= \rho_2(v) && \text{(since } \alpha \text{ is non-branching)} \\ &= \rho_1(v) && \text{(since } fv(G) - bv(\alpha) \subseteq fv([\alpha]G)) \\ &= \rho_1'(v) && \text{(since } \alpha \text{ is non-branching)} \end{aligned}$$

So, by the structural induction hypothesis, $\rho_2' \models \sim G$, and $\rho_2 \models \sim [\alpha]G$. This completes the proof. □

4. The Interpretation Theorem

As discussed in Section 1, an implementation of a theory T_1 provides a translation of the non-logical symbols (undefined terms) of T_1 into the language of the implementing theory T_2. This process is complicated by the need to interpret sorts in T_1 as tuples of sorts in T_2. Therefore, we first expand T_2 to T_2' by adding a new sort for each tuple of sorts which is required. We then provide a translation of the symbols in T_1 into the language of T_2'. The total translation is a composite: (a) from T_1 to T_2', and then (b) from T_2' to T_2. (See Figure 4.1) In this section we shall deal with part (a) of the translation. To simplify the notation, we shall talk about an "interpretation from T_1 to T_2" rather than "from T_1 to T_2'" That is, the T_2 of this section will be the T_2' of the final section.

$$\begin{tikzcd}
\text{nonlogical symbols} \arrow[d] & T_1 \arrow[d, "(a)"] & \\
\text{phrases} & T_2' \arrow[d, "(b)"] & \text{sorts} \arrow[d] \\
& T_2 & \text{tuples of sorts}
\end{tikzcd}$$

Figure 4.1 The Translation Process

We assume that theory T_1 is expressed using a language L_1 (that is, a particular set of non-logical symbols, as in Section 2.1), and that theory T_2 is expressed in a language L_2. We begin by defining an interpretation from L_1 to L_2 as a map associating certain strings in L_2 with each symbol in L_1. This map is then extended to programs and formulas. We then define the notion of an interpretation from T_1 to T_2.

We show that, given an interpretation I from T_1 to T_2 and a structure A_2 for L_2, how to construct a structure A_1 for L_1 such that for any closed formula G, $A_1 \models G$ iff $A_2 \models G^I$. We use this result to show that if $T_1 \models G$, then $T_2 \models G^I$. These are the two theorems we need for this part of the translation.

We begin with the definition of a pre-interpretation from L_1 to L_2. Each sort σ of L_1 is interpreted as a sort σ^I of L_2 (which may eventually, in turn, be interpreted as a tuple of sorts). But in an L_2-structure, not every value of sort σ^I may be a representation of a value of sort σ. We therefore introduce a formula is-σ (of one free variable) to decide whether a value is a representation or not. This formula is sometimes called the concrete invariant [38].

At various times, we shall need to apply is-σ to different values. If the lone free variable of is-σ is

z_1, we write is-$\sigma(t)$ for $[z_1{:=}t]$is-σ. This has the same effect as substituting t for z_1 in is-σ, but (as mentioned in Section 3) we have not defined a substitution operator for the language. We shall refer to the combination of the formula is-σ and the variable z as "a formula is-$\sigma[z_1]$ of signature $<\sigma^I>$."

Similarly, to each procedure symbol A of L_1, of signature $<\sigma_1,\ldots,\sigma_n> := <\tau_1,\ldots,\tau_m>$, an interpretation assigns a program A^I of L_2 and variables $y_1,\ldots y_n, z_1,\ldots,z_m$ of L_2. The intention is that the atomic program $A(v_1,\ldots,v_n;t_1,\ldots,t_m)$ will be translated as

$$z_1{:=}t_1{}^I;\ldots;z_m{:=}t_m{}^I;A^I;v_1{}^I{:=}y_1;\ldots;v_n{}^I{:=}y_n$$

Intuitively, the y's and z's may be thought of as output and input registers to which A^I refers; the input parameters are passed by value and the output parameters are passed by result. Again, we package this information by saying "a program $A^I[y_1,\ldots,y_n;z_1,\ldots,z_m]$."

<u>Definition</u>: An <u>interpretation</u> I of L_1 in L_2 is an assignment of phrases of L_2 to each symbol of L_1 as follows:

(a) to each sort symbol σ of L_1, a sort symbol σ^I of L_2 and a formula is-$\sigma[z]$ (with signature $<\sigma^I>$) of L_2 (the <u>invariant</u> of σ in I).

(b) to each function symbol f (with signature $<\sigma_1,\ldots,\sigma_n> \rightarrow \tau$) of L_1, a function symbol f^I (with signature $<\sigma_1{}^I,\ldots,\sigma_n{}^I> \to \tau^I$) of L_2.

(c) to each predicate symbol p (with signature $<\sigma_1,\ldots,\sigma_n>$) of L_1, a formula $p^I[z_1,\ldots,z_n]$ (with signature $<\sigma_1{}^I,\ldots,\sigma_n{}^I>$) of L_2.

(d) to each individual variable symbol v (with signature σ) of L_1, an individual variable symbol v^I (with signature σ^I) of L_2.

(e) for each procedure symbol A of L_1, of signature $<\sigma_1,\ldots,\sigma_n>: = <\tau_1,\ldots,\tau_m>$, a program $A^I[y_1,\ldots,y_n;z_1,\ldots,z_m]$ of L_2, of signature $<\sigma_1{}^I,\ldots,\sigma_n{}^I>: = <\tau_1{}^I,\ldots,\tau_m{}^I>$, such that

(i) $(\text{assign}_\sigma)^I$ is $y_1: = z_1$

(ii) $(\text{forall}_\sigma)^I$ is $\text{forall}_\tau(y_1)$; is-$\sigma(y_1)$ (where $\tau = \sigma^I$).

(iii) no variable of the form v^I may appear in A^I, $fv(A^I) = \{z_1,\ldots,z_m\}$, and $bv(A^I) = \{y_1,\ldots,y_n\}$.

The parameters z_i and y_i must not be of the form v^I; they may be different for different symbols of L_1 or they may be the same.

Note that the interpretation of a distinguished predicate symbol $=_\sigma$ need <u>not</u> be equality in σ^I.

<u>Example 4.1</u> Consider the implementation of bounded stacks by arrays in Section 1. L_1, the language of stacks, consists of:

```
sort symbols:  int, stk

function symbols:  length:  <stk> → int

procedure symbols:  init:  <stk>:= <>
                    pop:   <stk>:= <stk>
                    push:  <stk>:= <int,stk>
```

plus variables, forall, assign, and the usual symbols of arithmetic operating on sort int.

L_1 is interpreted in a language L_2 of arrays, integers, and array-integer records:

```
sort symbols:  int, arr, rec

function symbols:  pair:  <arr, int> → rec
                   pr1:   <rec> → arr
                   pr2:   <rec> → int

procedure symbols:  initarray:  <arr>:= <int>
                    fetch    :  <int>:= <arr,int>
                    update   :  <arr>:= <arr,int,int>
```

plus variables, forall, assign, and the symbols of arithmetic as before.

The interpretation I of L_1 in L_2 is specified as follows:

Sorts: int $\mapsto$int, stk$\mapsto$rec; is-int=true; is-stk = (pr2(s) $\geq$ 0)

function symbols: length $\mapsto$pr2
individual variable symbols: s$\mapsto$s', n$\mapsto$n', etc.
predicate symbols: $=_{stk}\mapsto$ pr2(s_1) = pr2(s_2) &
[forall$_{int}$(i); (1≤i)?; (i≤pr2(s_1))?;
fetch (n_1;s_1,i); fetch (n_2,s_2,i)](n_1=n_2
(input variables s_1,s_2)

```
procedure symbols:  init ↦ [initarray(x;100); assign(s,pair(x,0))]
                         (output variable s)
                    pop ↦ [pr2(s_o) > 0?; assign(s_1;pair(pr_1(s_o),
                                                          pr2(s_o)-1))]
                         (output variable s_1, input variable s_o)
                    push ↦ [pr2(s_o) < 100?;
                           assign(x;pr1(s_o));
                           update(x;pr2(s_o)+1,n_o);
                           assign(s_1;pair(a,pr2(s_o)+1))]
                         (output variable s_1, input variables n_o,s_o)
```

The symbols of arithmetic are mapped to themselves.(*) This implementation differs from the one in Section 1 primarily in that in L_2 initarray, fetch, and update have been changed from function symbols to procedure symbols; we have done so merely to illustrate that possibility.

(End of example)

We extend an interpretation to terms in the obvious way; we write t^I for the interpretation of t. We interpret an atomic formula $pt_1 \ldots t_n$ as

$$[z_1:=t_1{}^I;\ldots;z_n:=t_n{}^I]p^I$$

and an atomic program $A(v_1,\ldots,v_n;t_1,\ldots,t_m)$ as

$$z_1:=t_1{}^I;\ldots;z_m:=t_m{}^I;A^I;v_1{}^I:=y_1;\ldots;v_n{}^I:=y_n.$$

We sometimes denote this program by $A^I(v_1,\ldots,v_n;t_1,\ldots,t_m)$.

*Strictly speaking, the arithmetic predicate symbols should be mapped to formulas, but we shall ignore this.

We cannot merely extend I to formulas and programs as the obvious homomorphism of strings, because a formula has an implicit universal quantification over its free variables. In the interpreted formula, this quantification must be restricted to those values of the free variables in L_2 which are legal, i.e., which satisfy the invariant of their sort. For example, if one interprets the theory of real numbers in the theory of complex numbers, the interpretation of the true formula $x^2 \geq 0$ is not just $x^2 \geq 0$ (which is false for the complex numbers), but

$$\text{is-real}(x) \supset (x^2 \geq 0).$$

Given an interpretation I, let the maps $G \mapsto G\dagger$ and $\alpha \mapsto \alpha\dagger$ be obtained by extending I to a homomorphism on strings.

If $x_1,\ldots,x_n$ are the free variables of G and $x_1,\ldots,x_n$ have sorts $\sigma_1,\ldots,\sigma_n$, then we define G^I to be

$$\text{is-}\sigma_1(x_1{}^I) \;\&\; \ldots \;\&\; \text{is-}\sigma_n(x_n{}^I) \supset G\dagger$$

we often abbreviate the hypothesis of this implication by U_G, and call it the preamble of G. Similarly, if $x_1,\ldots,x_n$ are the free variables of program α, we call

$$\text{is-}\sigma_1(x_1{}^I) \;\&\; \ldots \;\&\; \text{is-}\sigma_n(x_n{}^I)$$

the preamble of α, and abbreviate it by U_α. We do likewise for terms t.

Example 4.2. The formula

$$\text{length}(s_o) < 100 \supset [\text{push}(s;n,s_o);\text{pop}(s;s)](s=s_o)$$

is interpreted as

```
[s:=s'_o](pr2(s) ≥ 0) ⊃          -- s'_o is the image of the free
                                     variable s_o
    (pr2(s'_o) < 100 ⊃

        [n_o:=n';s_o:=s';        -- assign actuals to value formals
                                     for push
         pr2(s_o) < 100?;        -- code for push

         assign(x;pr1(s_o));

         update(x;pr2(s_o) + 1,n_o);

         assign(s_1;pair(a,pr2(s_o) + 1));

         s':=s_1;                -- assign result formal to actual

         s_o:=s';                -- now do the same for call on pop

         pr2(s_o) < 100?;

         assign(s_1;pair(pr1(s_o),pr2(s_o)-1));

         s':=s_1]

        [s_1:=s';s_2:=s'_o]      -- now load formals for equality
                                     formula
        (pr2(s_1) = pr2(s_2)  -- equality formula

         & [forall_int(i); (1 ≤ i)?;(i ≤ pr2(s_1))?;   -- choose an
                                                          index i
            fetch (n_1;pr1(s_1),i); fetch(n_2;(pr2(s_2),i)]
                                   -- access both arrays
          (n_1=n_2)))              -- should get same answer
```

Here the first line is the preamble; although n was free in the original formula, we have not included it in the preamble since the associated invariant is always true. This formula expresses the fact the "bodies" of push and pop given by the interpretation behave in the proper way.

<u>Definition</u>. If T_1 is a theory in language L_1, and T_2 is a theory in language L_2, then an <u>interpretation of</u> T_1 <u>in</u> T_2 is an interpretation I of L_1 in L_2 such that the following formulas are logical consequences of T_2:

I0. $\exists x(\text{is-}\sigma(x))$ for each sort σ of L_1

I1. $\text{is-}\sigma_1(x_1) \,\&..\&\, \text{is-}\sigma_n(x_n) \supset \text{is-}\sigma(f^I x_1 \ldots x_n)$
for each function symbol $f:\langle\sigma_1,\ldots,\sigma_n\rangle \to \sigma$ in L_1

I2. $\text{is-}\tau_1(z_1) \,\&...\&\, \text{is-}\tau_m(z_m) \supset [A^I]\ \text{is-}\sigma_i(y_i)$
for each procedure symbol A of L_1 with signature $\langle\sigma_1,\ldots,\sigma_n\rangle := \langle\tau_1,\ldots,\tau_m\rangle$, and interpretation $A^I[y_1,\ldots,y_n;z_1,\ldots z_m]$, and $1 \le i \le n$.

I3a. $(x=x)^I$

I3b. $(x_1=y_1 \,\&...\&\, x_n=y_n \supset (fx_1\ldots x_n = fy_1\ldots y_n))^I$

I3c. $(x_1=y_1 \,\&...\&\, x_n=y_n \supset (px_1\ldots x_n \supset py_1\ldots y_n))^I$

I4. G^I for each axiom G of T_1.

Before proceeding, we should explain the significance of this definition. As we shall see in Section 6, an implementation of T in T' consists of an interpretation of T in an extension T'' of T. Therefore, to verify the correctness of an alleged implementation of T in T', one must deduce I0-I4 from T''. Put another way, I0-I4 are the conditions for correctness of an implementation.

I0-I3 are "frame" conditions. I0 states that the domain of interpretation of each sort is non-empty. I1 states that functions preserve their sort invariants, that is, if the input data satisfies the invariants of the input sort, then the output of the interpreted function satisfies the invariant of its output sort. Similarly, I2 says that interpreted atomic programs preserve their sort invariants. The formulas I3a,b,c say that the interpretation of equality is a reflexive relation which is respected by all the function and predicate symbols of L_1.

This leaves I4 as the sole "interesting" property required of an interpretation. It says, as suggested in the Introduction, that the translations of the axioms of T_1 are logical consequences of T_2.

Lemma 4.1 (Thinning Lemma) If $x_1,\ldots,x_k$ include (perhaps properly) the free variables of G, and $T_2 \models \text{is-}\sigma_1(x_1{}^I) \& \ldots \& \text{ is-}\sigma_k(x_k{}^I) \supset G^\dagger$, then $T_2 \models G^I$.

Proof. As in the predicate calculus case using I0 [33]. □

This lemma allows us to remove superfluous variables from preambles.

Lemma 4.2. If I is an interpretation of T_1 in T_2, and t is a term of sort σ in L_1 then $T_2 \models U_t \supset \text{is-}\sigma(t^I)$.

Proof. By induction on t, using condition I1. □

Lemma 4.3 If I is an interpretation of T_1 in T_2, and $\alpha = A(v_1,\dots,v_n;t_1,\dots,t_m)$ is an atomic program in L_1, with v_i of sort σ_i, then

$$T_2 \models U_\alpha \supset [A^I(v,t)]\text{is-}\sigma_i(v_i{}^I)$$

Proof. By Lemma 4.2 and I2. □

Lemma 4.4 If I is an interpretation of T_1 in T_2, then for any α and G, $T_2 \models U_{[\alpha]G} \supset [\alpha^I]U_G$.

Proof. By induction on α.

Let α be atomic. We will show that for α atomic, if $x \in fv(G)$, then $T_2 \models U_{[\alpha]G} \supset [\alpha^I]\ \text{is-}\sigma(x^I)$. Now

$fv([\alpha]G) = fv(\alpha) \cup (fv(G) - bv(\alpha))$, so $fv(G) \subseteq (fv([\alpha]G) - bv(\alpha)) \cup bv(\alpha)$. If $x \in fv([\alpha]G) - bv(\alpha)$, then x^I is not assigned in α^I, so

$$\models U_{[\alpha]G} \supset \text{is-}\sigma(x^I) \qquad (x \in fv([\alpha]G))$$

and $\quad T_2 \models \text{is-}\sigma(x^I) \supset [\alpha^I]\ \text{is-}\sigma(x^I) \quad$ (x^I not assigned in α^I).

If $x \in bv(\alpha)$, then

$$\models U_{[\alpha]G} \supset U_\alpha \qquad (fv(\alpha) \subseteq fv([\alpha]G))$$

and $\quad T_2 \models U_\alpha \supset [\alpha^I]\ \text{is-}\sigma(x^I) \qquad$ (Lemma 4.3)

Conjoining these results for each free variable in G, $T_2 \models U_{[\alpha]G} \supset [\alpha^I]U_G$.

If $\alpha = \beta;\gamma$, we must show $T_2 \models U_{[\beta][\gamma]G} \supset [\beta^I][\gamma^I]U_G$.

We proceed as follows:

1. $T_2 \models U_{[\beta][\gamma]G} \supset [\beta^I]U_{[\gamma]G}$ (Induc. Hyp. - β)

2. $T_2 \models U_{[\gamma]G} \supset [\gamma^I]U_G$ (Induc. Hyp. - γ)

3. $T_2 \models [\beta^I]U_{[\gamma]G} \supset [\beta^I][\gamma^I]U_G$ (PDL, 2)

4. $T_2 \models U_{[\beta][\gamma]G} \supset [\beta^I][\gamma^I]U_G$ (Taut., 1,3)

If $\alpha = \beta \cup \gamma$, then $U_{[\beta \cup \gamma]G} \equiv U_{[\beta]G} \,\&\, U_{[\gamma]G}$. The rest of the calculation is trivial.

If $\alpha = [H?]$ then the result is immediate.

This leaves the case $\alpha = \beta^*$. Since $bv(\beta^*) = \phi$, $fv([\beta^*]G) = fv(\beta) \cup fv(G) = fv(G) \cup fv([\beta]G)$. Hence $U_{[\beta^*]G} \equiv U_G \,\&\, U_{[\beta]G}$. We will show that $T_2 \models U_G \,\&\, U_{[\beta]G} \supset [\beta^I](U_G \,\&\, U_{[\beta]G})$, from which the needed result follows easily. If $x \in bv(\beta)$, then $T_2 \models U_{[\beta]G} \supset [\beta^I]\ \text{is-}\sigma(x^I)$, by the same argument as in the base case, except that we appeal to the induction hypothesis instead of Lemma 4.3. If $x \notin bv(\beta)$, then x^I is not assigned in β^I, so $\models \text{is-}\sigma(x^I) \supset [\beta^I]\ \text{is-}\sigma(x^I)$. Conjoining these results, we deduce $T_2 \models U_G \,\&\, U_{[\beta]G} \supset [\beta^I](U_G \,\&\, U_{[\beta]G})$. □

Lemma 4.5 Let I be an interpretation of T_1 in T_2, let $\mathcal{A}_2$ be any L_2-structure, and σ be any sort of L_1. Then the interpretation of $=_\sigma$ induces an equivalence relation on that subset of U_σ where is-σ is true.

Proof. Trivial from I3. □

We shall need one more bit of notation.

Definition. If ρ is a state, v a variable of sort σ and t a term of sort σ, then $\rho[t/v]$ is the state

$$\lambda w.\ \text{if}\ \ w = v\ \ \text{then}\ \ (\rho\models t)\ \ \text{else}\ \ \rho(w).$$

We may now state the first main result of this section, which gives the "model construction" result for interpretations.

Theorem 4.1 Let I be an interpretation of T_1 in T_2, and let A_2 be an L_2-structure. Then there is an L_1-structure A_1 and a map J from states of A_2 to states of A_1 such that

(i) for any formula G of L_1 and state ρ of A_2 such that $\rho\models U_G$,

$$J\rho\models G \ \text{ iff } \ \rho\models G^{\dagger}$$

and

(ii) for any program α of L_1 and states ρ,ρ' of A_2 such that $\rho\models U_G$,

$$(J\rho,J\rho') \in [\![\alpha]\!] \ \text{ iff } (\exists\rho'')(J\rho'' = J\rho' \ \&\ (\rho',\rho'') \in [\![\alpha^I]\!]).$$

Proof: We will use superscripts (1) and (2) in place of A_1 and A_2. Thus, $U_\sigma^{(2)}$ denotes the carrier of sort σ^I in A_2. Let $\approx_\sigma$ denote the equivalence relation induced by $=_\sigma$ on the is-σ subset of $U_\sigma^{(2)}$. Denote that subset by $V_\sigma^{(2)}$. Following the definition of an L_1-structure in Section 2.2, we build A_1 as follows:

(a) for each sort σ of L_1, let $U_\sigma^{(1)} = V_\sigma^{(2)}/\approx$. This is nonempty by IO.

(b) for each function symbol $f{:}\langle\sigma_1,\ldots,\sigma_n\rangle \to \sigma$ of L_1, let $f^{(1)}{:}\ U_{\sigma_1}^{(1)} \times\ldots\times U_{\sigma_n}^{(1)} \to U_\sigma^{(1)}{:}([a_1],\ldots,[a_n]) \to [f^{(2)}a_1\ldots a_n]$.

This is independent of choice of representatives by I3b. (Here the square brackets denote equivalence classes).

(c) for each predicate symbol $p:<\sigma_1,\ldots,\sigma_n>$, let the predicate $p^{(1)}$ on $U_{\sigma_1}^{(1)} \times\ldots\times U_{\sigma_n}^{(1)}$ given by

$$p^{(1)}([a_1],\ldots,[a_n])$$

iff

$$(\forall\rho)(\rho z_1=a_1 \;\&\; \ldots \;\&\; \rho z_n=a_n \supset p^I[z_1,\ldots,z_n])$$

This is independent of choice of representatives by I3c.

(d) for each procedure symbol A of L_1, of signature $<\sigma_1,\ldots,\sigma_n>:=<\tau_1,\ldots,\tau_m>$, let the predicate $A^{(1)}$ on $U_{\sigma_1}^{(1)} \times\ldots\times U_{\sigma_n}^{(1)} \times U_{\tau_1}^{(1)} \times\ldots\times U_{\tau_m}^{(1)}$ be given by

$A^{(1)}([a_1],\ldots,[a_n],[b_1],\ldots,[b_m])$ iff

$(\forall\rho_1)(\rho_1 z_1=a_1\&\ldots\&\rho z_n=a_n \supset (\exists\rho_2)(\rho_2 y_1=b_1\&\ldots\&\rho_2 y_m=b_m\&(\rho_1,\rho_2)\in[\![A^I]\!]))$.

Recall $A^{(1)}$ is a predicate, but A^I is a program in L_2.

(End of construction)

To show that $A^{(1)}$ is an L_1-structure, we must show that assign$^{(1)}$ is the equality predicate and that forall$^{(1)}$ is the true predicate. Both of these conditions may be verified straightforwardly.

We next define the required map J from states of A_2 to states of A_1. For each sort σ of L_1, let e_σ be an arbitrarily chosen element of $U_\sigma^{(1)}$. Define $J_\sigma:U_\sigma^{(2)} \to U_\sigma^{(1)}$ by $J_\sigma a = [a]$ if is-$\sigma[a]$ and $J_\sigma a = e_\sigma$ otherwise. Then we may define $J\rho = \lambda v(J_\sigma(\rho v^I))$, where σ is the sort of v.

Note that J is surjective and that if t is any term, and $\rho \models U_t$, then $(J\rho \models t) = (\rho \models t^I)$.

We next verify the required conditions (i) and (ii) by structural induction on formulas and programs. Atomic formulas and boolean combinations are easy.

We next consider the case of $[\alpha]G$. Assume $\rho \models U_{[\alpha]G}$. Since $fv(\alpha) \subseteq fv([\alpha]G)$, we have $\rho \models U_\alpha$ as well. Then

$J\rho \models [\alpha]G$ by straightforward manipulation of the definitions, using the surjectivity of J and the induction hypotheses for α and G. Lemma 4.4 is needed to apply the induction hypothesis for G.

We next turn to programs. Again, without loss of generality, let $A(v,t)$ be an atomic program in L_1. Assume $\rho \models U_\alpha$ or equivalently, $\rho \models U_t$. We must show

$$(J\rho, J\rho') \in [\![A(v,t)]\!] \text{ iff } (\exists\rho'')(J\rho''=J\rho' \;\&\; (\rho,\rho'') \in [\![A^I(v,t)]\!])$$

In the left-to-right direction, we calculate:

1. $(J\rho, J\rho') \in [\![A(v,t)]\!]$ (Assumption)
2. $A^{(1)}(J\rho' v, J\rho \models t) \;\&\; \text{Equiv}\,(J\rho, J\rho', v)$ (Defn of $[\![\;]\!]$)
3. $(\forall\rho_1)(J\rho_1 z_1 = (J\rho \models t) \supset (\exists\rho_2)(J\rho_2 y_1 = J\rho' v \;\&\; (\rho_1,\rho_2) \in [\![A^I]\!]))$
   (Defn of $A^{(1)}$, surjectivity of J)

Let $\rho_1 = \rho[t^I/z_1]$. Then $\rho_1 z_1 = (\rho \models t^I) = (J\rho \models t)$. So from (3) we deduce

4. $(\exists\rho_2)(J\rho_2 z_2 = J\rho' v \;\&\; (\rho_1,\rho_2) \in [\![A^I]\!])$
5. $J\rho_2 = J\rho_1$ (no variable of the form x^I is assigned in A^I)

Let $\rho'' = \rho_2[y_1/v^I]$. We claim ρ'' is the required state. By the construction of ρ_1 and ρ'',

$$(\rho,\rho'') \in [\![z_1 := t^I; A^I; v^I := y_1]\!].$$

We claim $J\rho'' = J\rho'$. If w is a variable other than v, then

$$\begin{aligned} J\rho''w &= J\rho_2 w && \text{(Defn of } \rho''\text{)} \\ &= J\rho_1 w && (5) \\ &= J\rho w && \text{(Defn of } \rho_1\text{)} \\ &= J\rho' w && (2) \end{aligned}$$

At the variable v,

$$\begin{aligned} J\rho''v &= J\rho_2 z_2 && \text{(Defn of } \rho''\text{)} \\ &= J\rho' v && (4) \end{aligned}$$

So $J\rho'' = J\rho'$, as required.

In the right-to-left direction, we assume

$$(\exists\rho'')(J\rho'' = J\rho' \ \& \ (\rho,\rho'') \in [\![z_1 := t^I; A^I; v^I := y_1]\!])$$

and we need to conclude $\mathrm{Equiv}(J\rho, J\rho', v)$ and

$$(\forall\rho_1)(J\rho_1 z_1 = (J\rho\models t) \supset (\exists\rho_2)(J\rho_2 y_1 = J\rho' v \ \& \ (\rho_1,\rho_2) \in [\![A^I]\!]).$$

Since no variable of the form x^I gets assigned in A^I, the Equiv term is easy to prove. Continuing, let ρ_1 be an arbitrary $\mathcal{A}_2$-state such that $J\rho_1 z_1 = (J\rho\models t)$. Then by Theorem 3.1, there is a ρ_2 such that $(\rho_1,\rho_2) \in [\![A^I]\!]$ and ρ_2 agrees with ρ' on the bound variables of A^I. This completes the base step.

The cases $\alpha;\beta$, $\alpha \cup \beta$, and $G?$ follow by manipulation of the definitions; Lemma 4.4 is used in the $\alpha;\beta$ case to apply the induction hypothesis for β. For α^*, we rely on the fact that $(\rho,\rho') \in [\![\alpha^*]\!]$ iff $(\exists n)((\rho,\rho') \in [\![\alpha^n]\!])$ and proceed by induction on n. □

<u>Corollary 4.1</u>. If G is a closed formula, then $A_1 \models G$ iff $A_2 \models G^I$.

<u>Proof</u>: Immediate from part (i) of the theorem. □

<u>Corollary 4.2</u>. Let I be an interpretation of T_1 in T_2, and let A_2 be a model of T_2. Then the structure A_1 of the theorem is a model of T_1.

<u>Proof</u>: Let G be a formula of T_1. Now, any state of A_1 is of the form $J\rho$, where ρ is an A_2-state such that every variable v^I passes its is-σ test. Then Theorem 4.1 says that

$$(\rho \models U_G \ \& \ \rho \models G^\dagger) \supset (J\rho \models G)$$

Since T_2 logically implies G^I and $\rho \models U_G$, we have $\rho \models G^\dagger$, so $J\rho \models G$. □

This theorem allows us to take an L_2-structure and view it as an L_1-structure, giving the "upward-going" result discussed in the introduction. We shall sketch an example in Section 6. We now proceed to the "downward-going" result:

<u>Theorem 4.2</u> Let I be an interpretation of T_1 in T_2. If T_1 logically implies G, then T_2 logically implies G^I.

<u>Proof</u>. Let A_2 be any model of T_2. We will show that for any state ρ of A_2, $\rho \models G^I$. Assume $\rho \models G^I$ is false. Then $\rho \models U_G$ and $\rho \models \sim G^\dagger$.

Build A_1 as in Theorem 4.1. By the relation of the theorem, $J\rho \models \sim G$. But since A_1 is a model of T_1 and T_1 logically implies G, $J\rho \models G$. Therefore $\rho \models G^I$ must have been true. □

5. Adding New Sorts

The translations considered in Section 4 allowed considerable freedom in interpreting predicate symbols as formulas and procedure symbols as programs. Sort symbols, however, must be interpreted as sorts. This is not adequate for applications, since the implementing theory T_2 would not be expected to have a sort for "pair of array and integer." We therefore interpret T_1 not in T_2, but in T_2', an extension of T_2 obtained by adding product sorts as needed.

For typographical convenience, we consider a theory T in language L and extend it to a theory T' in language L'. Again, given an L'-structure, we show that we can build a related L-structure, as we did in Theorem 4.1, and we give a translation from formula G of L' to formulas G' of L such that $T' \models G$ iff $T \models G'$.

Let σ_1 and σ_2 be sort symbols of L. Add to L a new sort symbol σ, a countably infinite set of variables of sort σ, and function symbols $pr1{:}\sigma \rightarrow \sigma_1$, $pr2{:}\sigma \rightarrow \sigma_2$, and $pair{:}\ \langle\sigma_1,\sigma_2\rangle \rightarrow \sigma$. For each variable x of sort σ, designate two variables x^L and x^R of sorts σ_1 and σ_2. Let L' be the language obtained by adding these new symbols and *deleting* all variables of the form x^L and x^R. (We assume that the variables x^L and x^R are chosen so as to leave infinitely many variables of sorts σ_1 and σ_2.

Let T' be the theory obtained by adding to T the axioms:

P1. pair(pr1(x,pr2(x)) = x

P2a. pr1(pair(x,y)) = x

P2b. pr2(pair(x,y)) = y

Again, we proceed by defining a translation from formulas of L' to formulas of L. This translation will be the identity except on phrases in which variables of sort σ occur.

<u>Definition</u>: If t is a term of L' of sort other than σ, then we define the term t' of L as follows:

1. if t is pr1(x), then $t' = x^L$
2. if t is pr1(pair(t_1,t_2)), then $t' = t_1'$
3. if t is pr2(x), then $t' = x^R$
4. if t is pr2(pair(t_1,t_2)), then $t' = t_2'$
5. if t is a variable, then t' = t
6. if $t = ft_1 \ldots t_n$ ($f \notin$ {pr1,pr2,pair}), then

$t' = ft'_1 \ldots t'_n$.

Note that, since <u>pair</u> is the only function of sort σ, these cases are exhaustive.

We now extend this translation to programs and formulas. Given α or G in L', we obtain α' or G' in L by replacing every occurrence of:

$t_1 = t_2$ (t_1,t_2 of sort σ) by

$$(\text{pr1}(t_1))' = (\text{pr1}(t_2))' \ \& \ (\text{pr2}(t_1))' = (\text{pr2}(t_2))'$$

$(\forall x)$ (x of sort σ) by $\forall x^L; \forall x^R$

$x{:=}t$ (x of sort σ) by

$$z_1{:=}(\mathrm{pr1}(t))'; \ z_2{:=}(\mathrm{pr2}(t))'; \ x^L{:=}z_1; x^R{:=}z_2$$

where z_1 and z_2 are variables which appear nowhere else in G.

Example 5.1. The formula generated in Example 4.2 would be translated as:

$$[z_1{:=}\ s_o'^{L};\ z_2{:=}s_1'^{R};\ s^L{:=}\ z_1;\ s^R{:=}\ z_2](s^R \geq 0) \supset$$
$$(s_o'^{R} < 100 \supset$$
$$[n_o{:=}\ n';\ z_3{:=}\ s'^{R};\ z_4{:=}\ s'^{R};\ s_o^L{:=}\ z_3; s_o^R{:=}z_4;$$
$$s_o'^{R} < 100?;$$
etc.

(end of example)

Theorem 5.1 Any L-structure $\mathcal{A}$ can be extended to an L^L-structure $\mathcal{A}'$ with a bijective map J from states of $\mathcal{A}$ to states of $\mathcal{A}'$ such that

(i) the carrier of σ in $\mathcal{A}'$ is $U_{\sigma_1} \times U_{\sigma_2}$

(ii) for any formula G of L' and state ρ of $\mathcal{A}$,

$$\rho \models G' \ \text{ iff } \ J\rho \models G$$

Proof Let $\mathcal{A}'$ be $\mathcal{A}$ augmented by adding $U_{\sigma_1} \times U_{\sigma_2}$ as the carrier for sort σ, with pr1, pr2, and pair being the evident Cartesian functions. Let $J\rho$ be the same as ρ on all variables of sort other than σ, and for each variable x of sort σ, let $(J\rho)(x) = \langle\rho(x^L), \rho(x^R)\rangle$. J is clearly bijective, since the variables x^L and x^R have been deleted in L'.

If t is any term of L' of sort other then $(\rho|=t') = (J\rho|=t)$, by the inductive definition of $(-)'$. We may now prove (ii) by a routine structural induction; for α^*, we recall that $[\![\alpha^*]\!] = \bigcup [\![\alpha^n]\!]$ and proceed by induction on n. □

__Corollary 5.1__. A is a model of T iff A' is a model of T'. □

__Corollary 5.2__. $T |= G'$ iff $T' |= G$. □

__Definition__. We say a theory T' is an __extension by definitions__ of T iff T' is obtained from T by repeatedly adding new sorts. If T' is an extension by definitions of T, and G is a formula in the language of T', we use G' to denote the formula obtained from G by performing in turn the translations corresponding to each addition of a predicate or a sort.

__Corollary 5.3__. If T' is an extension by definitions of T, and G is a formula in the language of T', then $T' |= G$ iff $T |= G'$.

__Proof__. By induction on the number of new sorts, using Theorem 5.1. □

6. Implementations

We may now synthesize the results of the last two sections to restate Part B of our thesis:

Definition: An implementation of a theory T_1 in a theory T_2 is an interpretation I of T_1 in an extension by definitions T_2' of T_2.

We call T_1 the implemented theory, T_2 the implementation theory, and T_2' the interface theory. T_1 is implementable in T_2 if there is an implementation of T_1 in T_2.

Therefore, to prove the correctness of an alleged implementation of T_1 in T_2, one need only prove the formulas I0-I4, given in Section 4, in T_2'.

We may now state the main theorem, which says that our definition of implementation meets the requirements set forth in Section 1.

Theorem 6.1 (The Implementation Theorem). Let $(I, (-)')$ be an implementation of T_1 in T_2.

(i) (synthetic version) If A is any L_2-structure, then there is an L_1-structure A' such that for any closed formula G of L_1, $A' \models G$ iff $A \models (G^I)'$.

(ii) (analytic version) For any formula G of L_1, if $T_1 \models G$, then $T_2 \models (G^I)'$.

Proof (i) By Corollaries 4.1 and 5.1, and (ii) by Theorem 4.2 and Corollary 5.2. □

Example 6.1. Let us start with the "standard model" of arrays and construct a model of stacks. By Theorem 5.1, we first augment the model by adding a sort <rec> with carries $U_{arr} \times U_{int}$, and with the standard pairing and projection functions. By Theorem 4.1, we now construct a structure whose carrier for the sort stk is the quotient of $\{(x,n) | x \in U_{arr} \ \& \ n \geq 0\}$ by the equivalence relation induced by $=_{stk}$; this quotient set is isomorphic to ω^*. Construction (d) of the proof of Theorem 4.1 now tells how push, pop, etc. work in this model. (End of example).

7. Conclusions and Open Problems

This work had its roots in the paper by Elgot and Snyder [8] on the notion of equality of lists, which led us to Tarski [36]. Our reading of Tarski's book crystallized our thinking about the problem of implementation, which we had addressed in a less than satisfying way in [37].

Given this background, a few words are in order about our choice of Dynamic Logic as a specification language. We chose DL for a number of reasons. It subsumes the most commonly used specification language, that of partial correctness assertions P{S}Q. It also has a well-understood theoretical development, and relies on the standard "assignment" model of programs.

Our decision to abandon, at least temporarily, the use of algebra as a specification language was a conscious one. Workers in algebraic semantics have been keenly aware of the pitfalls of implementations [6, 7, 11, 12, 24]. Nevertheless, as we demonstrated in Section 1, the distinction between a model and an implementation arises in any specification language, not just in algebra. We were therefore led to abjure any approach which leaned too heavily on algebraic machinery.

Communication is also a factor. The crucial distinction between specification and modelling is particularly difficult to communicate in an algebraic framework. Indeed, the confusion between these two notions seems to be the cause of considerable debate inside the algebraic community.

In any case, it is not our intention to promote DL or DLP as the One True Specification Language. We intend instead that this work be taken as a paradigm: one test of a reasonable specification language is that a reasonable version of the Implementation Theorem should hold for it.

Our notion of implementation subsumes that of Hoare [16] and Robinson & Levitt [31]. Both of these works restrict the specification language (that of T_1) to formulas of the form $P \supset [A]Q$, where A is a single procedure call. For essentially the same amount of work in establishing the conditions for the Interpretation Theorem, we get a far richer specification language. Furthermore, we allow equality to be interpreted as an arbitrary equivalence relation, whereas preceding work used fixed interpretations. Hoare [16] interprets $x = y$ as $\mathcal{A}(x) = \mathcal{A}(y)$ where $\mathcal{A}$ is the so-called "abstraction function." Often, however, $\mathcal{A}$ lacks a suitable range; this problem is alleviated in Robinson & Levitt [31], where $x = y$ is interpreted as

$$f(x) = f(y) \quad \text{for all V-functions } f.$$

In order to get a sufficiently fine interpretation of equality, it was then necessary to introduce "hidden V-functions." Our notion of interpretation includes both of these as special cases. (See, however, [18]).

Formula (3) in Section 1 is a typical example of a specification which is difficult to treat adequately in either Hoare's or Parnas, Robinson, and Levitt's framework.

Ehrig, Kreowski, and Padawitz [7] have studied a paradigm similar to ours in the context of algebraic specifications. Let T_1 and T_2 be algebraic specifications, i.e. the axioms are equalities. They implement T_1 in T_2 in two stages, just as we do. First, T_2 is extended to T_2' by adding additional sorts and function symbols corresponding to those in T_1, and additional equations constraining them. The symbols of T_1 may then be interpreted as the corresponding symbols of T_2'. This gives a functor from T_2-algebras to algebras in the language of T_1 (in their terminology, "synthesis" followed by "forgetting"). Our "concrete invariant" (is-σ) is replaced by restriction to the prime subalgebra (their "reachability"). Equality is interpreted as the smallest equivalence relation which contains the axioms of T_1 and which is preserved by the operations of T_1 (comparable to our I3). Thus all the equalities deducible from T_1 are automatically true in the interpretation. The implementation is correct iff an equality in the language of T_1 is true in the interpretation <u>only if</u> it is deducible from T_1 (Theorem 5.5(4) of [7]).

In other related work, Nakajima et. al. [26] have considered the importance of the interpretation theorem, with a quite different specification language. Correll [2] considered how function symbols could be interpreted as looping programs. Guttag, Horowitz, and Musser [13] and Gaudel [10] have discussed the notion of an interpretation of equality in the context of algebraic semantics. The "mapping function expressions" of Robinson & Levitt [31] are a kind of interpretation.

We conclude with some open questions and problems.

1. Provide a reasonable axiomatics for DLP, and give a proof-theoretic version of the Analytic Implementation Theorem. (We have developed such an axiomatics). While no axiomatic system for DLP can be complete, the capacity of proving such a theorem is an interesting test of the adequacy of a deductive system for a specification language.

2. We say that T_1 is implementable in T_2 iff there is an implementation of T_1 in T_2. Show that implementability is transitive.

3. Give a version of the Synthetic Implementation Theorem phrased in terms of implementations instead of models. This is closely related to the previous problem.

4. Extend our notion of interpretation to allow functions to be interpreted as procedures. One technical problem is that the usual way of introducing functions by definitions [33, pp. 59-60] involves equality, which is not preserved under interpretation. The techniques of Musser [25] should be relevant here. (Extending the notion of interpretation to allow functions to be interpreted as terms is trivial).

5. Extend our notion of interpretation to allow (a) array assignments and (b) value-result parameters.

6. Apply these techniques to some interesting implementations. Salwicki [32] has done some similar examples.

7. Extend this family of results to some other specification languages.

8. Extend this family of results to include other kinds of modalities [27, 30].

9. Many of our theorems take the form of adjunctions or Galois connections [23]. Are these theorems adjunctions in any useful sense? If so, how can this fact be exploited in algebraic logic?

Acknowledgements

We wish to thank C. C. Elgot for his encouragement over the years and for [8], and G. J. Sussman, whose comment that some versions of semantics suffered from "premature binding" taught me the elusive nature of The One True Specification Language. Neither of them, however, bears any responsibility for the direction I travelled with their remarks. Thanks also to A. R. Meyer for his remarks on an early version of this paper.

1. A. Church, Introduction to Mathematical Logic, Volume I Princeton University Press, Princeton, N. J., 1956.

2. C. H. Correll, Proving Programs Correct Through Refinement, Acta Informatica 9 (1978), 121-132.

3. O. J. Dahl and C. A. R. Hoare, "Hierarchical Program Structures" in Dahl, O. J., Dijkstra, E. W., and Hoare, C. A. R., Structured Programming, Academic Press, London, 1972, pp. 175-220.

4. E. W. Dijkstra, "Notices on Structured Programming" in Dahl, O. J., Dijkstra, E. W., and Hoare, C. A. R., Structured Programming, Academic Press, London, 1972.

5. E. W. Dijkstra, A Discipline of Programming, Prentice-Hall, Englewood Cliffs, N. J., 1976.

6. H. D. Ehrich, Extensions and Implementations of Abstract Data Type Specifications, Universitat Dortmund, Report 55/78.

7. H. Ehrig, H. J. Kreowski, and P. Padawitz, Algebraic Implementation of Abstract Data Types: Concept, Syntax, Semantics and Correctness, in Proc. 7th Colloq. on Automata, Languages, and Programming (1980).

8. C. C. Elgot and L. Snyder, "On the Many Facets of Lists" Theoret. Comp. Sci. 5 (1977), 275-306.

9. H. B. Enderton, A Mathematical Introduction to Logic, Academic Press, New York and London, 1972.

10. M. C. Gaudel, Specifications Incomplètes Mais Suffisantes de la Representation des Types Abstraits, IRIA Research Report No. 320 (August, 1978).

11. J. Goguen, and F. Nourani, Some Algebraic Techniques for Proving Correctness of Data Type Implementations, extended abstract (1978).

12. J. A. Goguen, J. W. Thatcher, and E. G. Wagner, An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types, Current Trends in Programming Methodology, IV: Data Structuring (R. Yeh, ed.) Prentice-Hall, New Jersey, (1978), pp. 80-149.

13. J. V. Guttag, E. Horowitz, and D. R. Musser, Abstract Data Types and Software Validation, Comm. ACM 21 (1978), 1048-1064.

14. D. Harel, A. R. Meyer, and V. R. Pratt, Computability and Completeness in Logics of Programs, Conf. Rec. 9th Ann. ACM Symp. on Theory of Computing (1977), 261-268.

15. D. Harel and V. R. Pratt, Nondeterminism in Logics of Programs, Proc. 5th ACM Conf. on Principles of Programming Languages (1978), 203-213.

16. C. A. R. Hoare, Proving Correctness of Data Representations, Acta Informatica 1 (1972), 271-281.

17. D. R. Hofstadter, Godel, Escher, Bach: An Eternal Golden Braid, Basic Books, New York, 1979.

18. S. Kamin, Final Data Type Specifications: A New Data Type Specification Method, Conf. Rec. 7th ACM Symp. on Principles of Programming, Languages (1980), 131-138.

19. D. E. Knuth and L. T. Pardo, The Early Development of Programming Languages, Stanford University Computer Science Department Technical Report STAN-CS-76-562 (August, 1976).

20. D. J. Lehmann and M. B. Smyth, Data Types, University of Warwick, Theory of Computation Report No. 19, (May, 1977).

21. B. Liskov, A Snyder, R. Atkinson, and C. Schaffert, "Abstraction Mechanisms in CLU" Comm. ACM 20 (1977), 564-576.

22. B. Liskov and S. Zilles, Specification Techniques for Data Abstractions, IEEE Trans. on Software Eng., SE-1 (1975), 7-19.

23. Saunders MacLane, Categories for the Working Mathematician, Springer-Verlag, New York, 1971.

24. M. E. Majster, Limits of the Algebraic Specification of Data Types, SIGPLAN Notices 12, 10 (October, 1977), 37-42.

25. D. R. Musser, A Proof Rule for Functions, University of Southern California Information Sciences Institute, Marina del Rey, CA., Technical Report ISI/RR-77-62 (October, 1977).

26. R. Nakajima, M. Honda and H. Nakahara, Hierarchical Program Specification and Verification - a Many-Sorted Logical Approach, Acta Informatica 14 (1980), 135-155.

27. R. Parikh, A Decidability Result for a Second Order Process Logic, Proc. 19th Ann. Symp. on Foundations of Computer Science (1978) IEEE, pp. 177-183.

28. D. L. Parnas, "A Technique for Module Specification with Examples" Comm. ACM 15 (1972), 330-336.

29. V. R. Pratt, Semantical Considerations on Floyd-Hoare Logic, Proc. 17th IEEE Symp. on Foundations of Comp. Sci. (1976), 109-121.

30. V. R. Pratt, Six Lectures on Dynamic Logic, Massachusetts Institute of Technology, Laboratory for Computer Science MIT/LCS/TM-117 (December, 1978).

31. L. Robinson, and K. N. Levitt, Proof Techniques for Hierarchically Structured Programs, Comm. ACM 20 (1977), 271-283.

32. A. Salwicki, "On Algorithmic Theory of Stacks" Mathematical Foundations of Computer Science 1978 (J. Winkowski, ed.) Lecture Notes in Computer Science, vol. 64, Springer, Berlin, 1978, pp. 452-461.

33. J. R. Shoenfield, Mathematical Logic, Addison-Wesley, Reading, Massachusetts, 1967.

34. J. M. Spitzen, K. N. Levitt, and R. Robinson, An Example of Hierarchical Design and Proof, Comm. ACM 21 (1978), 1064-1075.

35. J. E. Stoy, Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory, MIT Press, Cambridge, Massachusetts, 1977.

36. A. Tarski, Introduction to Logic and to the Methodology of Deductive Sciences, Oxford University Press, New York, 2nd edition, 1946.

37. M. Wand, Final Algebra Semantics and Data Type Extensions, J. Comp. & Sys. Sci. 19 (1979), 27-44.

38. W. A. Wulf, R. L. London, and M. Shaw, "An Introduction to the Construction and Verification of Alphard Programs," IEEE Trans. on Software Eng. SE-2 (1976), 253-265.