On Proving the Correctness of Data Type Implementations

By

P.A. Subrahmanyam

Department of Computer Science
University of Utah
Salt Lake City, Utah

UUCS - 80-101

On Proving the Correctness of Data Type Implementations

P.A.Subrahmanyam

Department of Computer Science

University of Utah

Salt Lake City, Utah 84112

September 1979

ABSTRACT

In order to prove the correctness (or consistency) of an implementation of a data type with respect to the data type's specifications, the minimal amount of information that needs to be provided consists of: (i) a specification of the type being implemented; (ii) a specification of the representation type; and (iii) a specification of an implementation. This paper develops a method for proving the correctness of data type implementations that requires only this minimal amount of information to be specified in order for a proof to be attempted; this is in contrast to several of the existing methods which need additional information augmenting (i)-(iii) to be specified in order to be applicable. The ensuing generality of the proposed method makes it more amenable to automation. Examples of applications of the proof method are presented, all of which have been automated.

This work was supported in part by an IBM Fellowship

Table of Contents

Appendices

List of Figures

ON PROVING THE CORRECTNESS OF DATA TYPE IMPLEMENTATIONS

1. Introduction

Programming involves representing the abstractions of the objects and operations relevant to a given problem domain using "primitive" objects and operations that are presumed to be already available; ultimately, such primitives are those provided by the available hardware. Various programming methodologies advocate ways of achieving "good" organizations of layers of such representations, in attempting to provide effective means of coping with the complexity of programs. The importance of data abstractions in achieving elegant organizations was cogently argued for by Hoare in [1], and their use has, by now, been amply demonstrated.

Hoare also proposed a method for proving the correctness of implementations of data abstractions in [7]. Due to a proliferation of languages incorporating variations of the notion of abstract data types (for example, [8] and [14]), techniques for proving the correctness of implementations of abstract types have since gained in importance [15]. Two of the most widely used techniques are those due to Hoare [7], and Guttag et al [5]. In this paper, we present a new proof method that is more general than the existing methods; the nature of this generality makes our method more amenable to automation. In particular, the method proposed has the important advantage of normally requiring only the minimal amount of information that is necessary in order to enable a proof of the correctness (or consistency) of an implementation of a data type with respect to its specifications. This is in contrast to most of the existing proof methods, including those of [7] and [5], wherein it is usually necessary to augment the specifications of (i) the data type being implemented, (ii) the representation type, and (iii) the implementation, with additional information in order to carry out the proofs. We relegate details of further comparisons to section 5.

1.1. Summary of the Paper

We briefly review some basic definitions relating to abstract data types in Section 2. We adopt the view that the inherent structure of an abstract data type is characterized by its "externally observable behavior" -- such behavior is reflected by functions that return elements of "known" types (i.e. types other than the one being defined). A notion of equivalence of instances of a type under extraction is developed to make precise this externally observable behavior. An implementation of one data type (the Type of Interest TOI) in terms of another (the Target Type TT) is defined as a map between the functions and the objects of the two types that preserves the observable behavior of the TOI. We show (Theorem 7) that this definition coincides with the more conventional definition of an implementation as a surjective homomorphism from the equivalence classes of the representation (target) type to the equivalence classes of the Type of Interest. However, it is this difference in perspective that affords insight into the added generality of our proof method.

Section 3 outlines the theoretical basis underlying the proof method. We first observe that a straightforward induction proof based directly on the developments in Section 2 is not feasible in practice; an alternative proof strategy is then developed and shown to be correct. In Section 4 we illustrate an application of the proof method; we have chosen to first illustrate the proof of a implementation of a Stack in order to highlight some of the important differences between the present method and previously proposed proof strategies (these are elaborated in section 5.) Other examples attempted include proofs of implementations of a Queue, a SymbolTable, and a TextEditor. All of these proofs have been automated.

2. Preliminary Definitions

Definition 1: An abstract data type can be regarded as a many sorted algebra, consisting of a set X of sorts, a set F of function symbols, and a set of equations relating terms generated by F and containing free variables. Each f in F has an associated arity that is an element $(x_1 x_2 .. x_n, x_{n+1})$ of $X^* \times X$. We also write $f:(x_1, x_2, ... x_n) \rightarrow x_{n+1}$ (for an example, see figure 2-1 on page 3).

Definition 2: Let $V = \{V_1, \ldots V_i, \ldots\}$, where V_i is a set of variables of sort x_i. The word algebra $W[F,V]$ generated by F and V consists of the union of the sets $W_x^{(n)}[F,V]$, $n = 0,1,2\ldots$ defined as follows:

1. all variables of sort x are in $W_x^{(0)}[F,V]$
2. all constants of sort x, (that is $f\colon () \rightarrow x$) are in $W_x^{(0)}[F,V]$
3. if $f\colon x_1, x_2, \ldots, x_k \rightarrow x$, then $f(t_1, \ldots, t_k)$ is in $W_x^{(n)}[F,V]$ if for each i, t_i is in $W_{x_i}^{(n-1)}[F,V]$, and at least one t_i is not in $W_{x_i}^{(n-2)}[F,V]$.

Figure 2-2 illustrates the word algebra generated by functions defined on a Stack.

```
Type Stack(Item)

Syntax
        NEWSTACK: () -> Stack
        PUSH: (Stack, Item) -> Stack
        POP: (Stack) -> Stack
        TOP: (Stack) -> Item U {UNDEFINED}
        ISEMPTY: (Stack) -> Boolean

Semantics

        for all s in Stack, x in Item,

        POP(NEWSTACK) = NEWSTACK
        POP(PUSH(s,x)) = s

        TOP(NEWSTACK) = UNDEFINED
        TOP(PUSH(s,x)) = x

        ISEMPTY(NEWSTACK) = true
        ISEMPTY(PUSH(s,x)) = false

End Stack
```

Figure 2-1: STACK DEFINITION[1]

---

[1] For the purposes of this paper, we ignore the technicalities arising out of the presence of parameterized types and functions returning "error" values (see [13, 4]). However, the reader's intuition will not lead him astray in his comprehension of this paper.

The data type Stack can be viewed as consisting of

- the set of sorts X, X = {Stack, Item, Boolean}, the sorts themselves being Stack, Item, Boolean;
- the set of function symbols F^{Stack} = {NEWSTACK, PUSH, POP, TOP, ISEMPTY, TOP}, with associated arities as shown in figure 2-1, $F^{Boolean}$ = {FALSE, TRUE}, etc.;
- the set of terms in the word algebra generated by this set of functions consists of

$W_{Stack}[F^{Stack}, \{x,y,...\}]$
    = {NEWSTACK,
       PUSH(NEWSTACK,x),
       PUSH(NEWSTACK,y),
       ...,
       PUSH(PUSH(NEWSTACK,x),x),
       PUSH(PUSH(NEWSTACK,y),y),
       PUSH(PUSH(NEWSTACK,x),y),
       PUSH(PUSH(NEWSTACK,y),x),
       ...,
       POP(NEWSTACK),
       POP(PUSH(NEWSTACK,x)),
       ... } etc.;

$W_{Item}[F^{Stack}, \{x,y,...\}]$
    = {TOP(NEWSTACK),
       TOP(PUSH(NEWSTACK,x)),
       ...,
       ...,
       TOP(POP(NEWSTACK)),
       ...} etc.;

- the equations are those shown in figure 2-1.

Figure 2-2: Word algebra generated by F^{Stack}

2.1. Some Notational abbreviations

F^T denotes the set of functions defined on the data type T; V_T denotes the (countable) set of variables of type T. To improve readability, we often abbreviate $W_T[F \cup F^G, V]$ to $W_T[F]$ (that is, the functions F^G defined on the "known" or "global" types G are omitted). When $F = F^T$, i.e., F is the entire set of functions defined on type T, we further abbreviate $W_T[F^T]$ to W_T.

2.2. Equivalence under extraction operations

The functions F^T defined on an abstract data type T can be categorized into Base constructors (BC^T), which spawn new instances of the type (e.g. NEWSTACK), Constructors (C^T), which form new instances of the type from existing ones (e.g. PUSH, POP), and extraction functions or extractors (E^T), which return members of other "known" types (e.g. TOP, ISEMPTY).

We adopt the viewpoint that any object representing an instance of a type is completely characterized by its "externally observable" properties; such properties are just those that are obtained as results of applications of extraction functions defined on the type. This is made precise in the notion of extraction equivalence of instances of the type [12, 10].

Informally, two terms t_1 and t_2 are said to be extraction equivalent if every sequence of function applications that terminates with the application of an extraction function yields the same (or "equivalent") results on the two terms. As an example, two instances of the type Stack (say, s_1 and s_2) are extraction equivalent iff the applications TOP(s_1) and TOP(s_2), TOP(POP(s_1)) and TOP(POP(s_2)), ..., TOP(PUSH(s_1,x_1)) and TOP(PUSH(s_2,x_1)), ..., ISEMPTY(s_1) and ISEMPTY(s_2), ISEMPTY(POP(s_1)) and ISEMPTY(POP(s_2)), ..., ISEMPTY(PUSH(s_1,x_1)) and ISEMPTY(PUSH(s_2,x_1)), ..., yield the same results pairwise.

We now formalize the notion of extraction equivalence. For any term t, we denote by $t[v|t']$ the term obtained from t by replacing each occurrence of v in t by the term t′. (For this to be well defined, it is necessary that the sorts of t′ and v be the same.) We denote by $t[v \text{ in } V_T|t']$ the term obtained by substituting t′ for all occurrences, in t, of variables that are contained in V_T. Let t_g be a term in the word algebra $W_g[F,V]$ where g in G is different from T; further, let t_g contain (one or more) occurrences of variables of sort T. Let t′ and t″ be obtained by substituting t_1 and t_2 respectively for all occurrences of variables of sort T in t_g. Thus $t' = t_g[v \text{ in } V_T|t_1]$ and $t'' = t_g[v \text{ in } V_T|t_2]$. (Note that the terms t′ and t″ obtained by this process represent all possible pairs of terms obtained by applying sequences of functions ending in an extraction function to t_1 and t_2 cf. the example in the previous paragraph.)

> Definition 3: t_1 and t_2 are said to be extraction equivalent in T if and only if t′ and t″ are (extraction) equivalent in g. Thus, $t_1 =_T t_2$ if and only if
>
> either (i) $t_1 = t_2$,
>
> or (ii) (for all g in G)(for all t_g in $W_g[F,V]$) $(t_g[v \text{ in } V_T|t_1] =_g t_g[v \text{ in } V_T|t_2])$,
>
> where G is the union of all "known types" that are returned by extraction functions defined on T. To avoid ambiguity, the = sign has been labeled to apply over the type domain of its arguments.

Two important observations immediately follow as a result of this definition:

1. When G is the empty set, extraction equivalence becomes identical to syntactic equivalence.
2. Syntactic equivalence implies extraction equivalence. Thus, $t_1 = t_2 \Rightarrow t_1 =_T t_2$.

2.3. Defining an implementation

Informally, an implementation of one data type, the type of interest TOI, in terms of another, the target type TT, is a map from the functions and the objects of TOI to those of TT which preserves the "observable behavior" of the type of interest. That is, whenever extraction functions are applied to objects of TOI, yielding instances of known types, the corresponding computation in the implementation domain should yield identical results. This is the import of the Definition 6 below.

On the other hand, the conventional characterization of a "correct" implementation embodies the requirements that (i) every instance of TOI is represented by some instance(s) of the representation type, and that (ii) the implementations of the functions defined on TOI "work properly." Formally, the existence of a surjective map from the equivalence classes in the representation type TT to the equivalence classes in the type of interest TOI ensures that every instance of TOI is represented by at least one instance of TT. Further, if this map is a homomorphism, it ensures that the functions "work properly" (see [13]). The existing proof methodologies are based primarily on this definition (see Section 5). In contrast, the proof method that we will outline in section 3 is based on the definition of correct implementation as developed in Definition 6. We show in Theorem 7 that the above notions of a correct implementation are formally equivalent. However, as mentioned in Section 1, the generality of the proof method delineated herein stems from the difference in our perspective.

We can define an implementation map with greater precision in terms of a (restricted) derivor [13]; this is done in Definition 4 below. However, we first need to introduce the notion of a term being viewed as a derived operator: informally, a term "POP(PUSH(s,x))" can be viewed as an operator (say POP-PUSH) with arity POP-PUSH: Stack, Item -> Stack, that maps the arguments (s,x) to the Stack "POP(PUSH(s,x))." POP-PUSH is called a derived operation ("derived" from the term "POP(PUSH(s,x))," where s and x are variables). When we explicitly want to indicate the function derived from a term t, we shall denote it d-(t).

Definition 4: A derivor d consists of the following pair of maps

(a) a map d_a from ({TOI} U G) to ({TT} U G); we shall be concerned only with the case where d_a maps TOI to TT and is the identity operator on all of the global sorts g in G. That is,

$$d_a(\mathrm{TOI}) = \mathrm{TT}, \text{ and}$$

$$(\text{for all } g \text{ in } G)\ [d_a(g) = g]$$

(This merely embodies the fact that we compute with TT-objects in place of TOI-objects and that everything else is unchanged.)

(b) a map θ from F^{TOI} to W_{TT} that preserves arity: if $f:x_1\ldots x_n\to x$ (f in F^{TOI}), then d-(θ[f]), (a term in W_{TT}) when viewed as a "derived operator" must have arity

$$d\text{-}(\theta[f]) : d_a(x_1)\ldots d_a(x_n) \to d_a(x).$$

By virtue of the simplification in (a), this arity is simply $x_1,\ldots x_n\to x$ with any occurrences of TOI being replaced by TT.

Henceforth, we simply write $\theta(f)$ for d-$\theta(f)$. The map θ which is of interest to us acts as the "identity" for functions f in F^G. Thus, the non-trivial part of θ is the one that transforms the functions defined on the type of interest to terms in the target type. This map will henceforth be referred to as the implementation map (or simply the implementation θ), and in essence, defines an implementation of the type TOI in terms of the type TT.

Definition 5: The d-derived algebra dTT defined by a derivor d is an algebra with functions {d-θ(f) | f in F^{TOI}} that is, the function corresponding to f is the term θ(f) viewed as a derived function. The equations of dTT are identical to those of TT.

<u>Example</u> If we consider the implementation of a Stack in terms of an Indexed Array (see Figure 2-3), the maps comprising the derivor are: d_a(Stack) = Indexed Array, d_a(Item) = Item, d_a(Boolean) = Boolean. The type Indexed Array is a tuple consisting of an Array and an integer; the map θ is detailed in figure 2-3.

It is straightforward to extend the domain of θ from F^{TOI} to $W_X[F^{TOI} \cup F^G, V]$, X in {TOI} U G: variables of sort TOI are mapped to variables of sort TT, while variables (and functions) of all other sorts remain unchanged. Then, if $t = f(t_1\ldots t_n)$, we define

$$\theta(t) = \theta(f_i^{TOI})(\theta(t_1,\ldots,\theta(t_n)).$$

Definition 6: A map θ defines a correct implementation of TOI in terms of TT if

$$(\text{for all } g \text{ in } G)(\text{for all } t_g \text{ in } W_g[F^{TOI}) \; [\theta(t_g) =_g t_g].$$

Theorem 7 shows that this interpretation of an implementation coincides with one defining a surjective homomorphism from the extraction equivalence classes of dTT to the extraction equivalence classes of TOI.

Theorem 7:

An implementation map θ such that

$$(\text{for all } g \text{ in } G)(\text{for all } t_g \text{ in } W_g[F^{TOI}])[\theta(t_g) = t_g] \quad \text{(I)}$$

implies the existence of a surjective homomorphism

$$\theta' : W_{dTT}/ E^{TT} \to W_{TOI} / E^{TOI}$$

where W_{dTT}/E^{TT} (respectively W_{TOI}/E^{TOI},) denotes the extraction equivalence classes induced by the functions E^{TT} (respectively E^{TOI}).

Proof: See Appendix I.

2.4. Kernel Functions

The first phase of constructing the formal specifications for a problem involves specifying an appropriate syntax that embodies the visible "syntactic interface" requirements of the problem, i.e. enumerating a set of functions F^T associated with appropriate arities. The second phase of the specification process involves specifying the semantics of the functions in F^T. In this later phase, it is convenient to first tentatively identify a minimal set of base constructors and constructors that serve to generate all representative instances of the type, such as {NEWSTACK, PUSH} for a Stack; we will refer to such a set of functions as a kernel set and denote it K^T. If the semantics of the remaining functions can be completely specified by defining their action only on the instances of the type generated by the postulated kernel set, then the initial identification of K^T fulfills the formal requirements of a set of kernel functions [11].[2]

More formally, a set of kernel functions K^T is characterized by the fact that every term in $W_T[F^T]$ is equivalent (under the set of defining equations) to at least one term in $W_T[K^T]$. Invariably, such a set K^T is identical to a syntactic version of a kernel set, defined to be the union of the functions that appear in the arguments on the left hand sides of the defining equations of the non-kernel functions; an algorithm to identify such a set can be found

---

[2] Of course, this phase of constructing formal specifications may undergo several iterations before a final set of specifications is settled upon, since the initial (and intermediate) specifications may provide an "unsatisfactory" interface for the user.

The map θ defining an implementation of a Stack using an Indexed Array is defined below. Let θ(s) = <a,i>.

θ(NEWSTACK) = <NEWARRAY, ZERO>
θ(PUSH(s,x)) = <ASSIGN(a,SUCC(i),x), SUCC(i)>
θ(POP(s)) = <a, PRED(i)>
θ(TOP(s)) = DATA(a,i)
θ(ISEMPTY(s)) = [i = ZERO]

i is an Integer Index, SUCC(i) is the Successor of the integer i (=i+1), PRED(i) is the Predecessor of the integer i (with the semantics $i \dot{-} 1$ for monus). Appendix III details the definitions of the types Array and Integer.

Figure 2-3: THE IMPLEMENTATION OF A STACK USING AN INDEXED ARRAY

---

in [12].[3] In other words, the equations that define the semantics of non-kernel functions refer explicitly only to terms generated by syntactic kernel functions; henceforth, we shall use K^T to denote the syntactic kernel set obtained from a given specification of the type T. We now proceed to elaborate on the relevance of this observation to the proof method.

3. On proving the correctness of an implementation

Recall from Definition 6 that a proof of the correctness of an implementation specified by a map θ involves showing that the following holds

$$\text{(for all } g \text{ in } G)\text{(for all } t_g \text{ in } W_g[F^{TOI}])\ t_g =_g \theta(t_g) \qquad \text{---(P)}$$

Now, every such term t_g is either of the form $e(v_1,\ldots,v_n)$ (for some extraction function e in E^{TOI}, where e: $X_1, \ldots, X_n \rightarrow X$, and v_i in V_{X_i},) e.g. TOP(s), or is obtained by instantiating the variables in $e(v_1,\ldots,v_n)$ e.g. TOP(NEWSTACK), TOP(POP(s′)), etc. Thus, if we consider the set of (uninstantiated) terms S of the form $e(v_1,\ldots,v_n)$ and prove that $e(v_1,\ldots,v_n) = \theta(e(v_1,\ldots,v_n))$ for every such term in S, then we shall have proved that θ defines a correct implementation. However, it may not be possible to carry through all of the required proofs directly, because of the lack of the appropriate forms of the defining equations. For example, there is no defining equation of the form TOP(s) = ..., that is normally specified for a stack.

As a consequence, in order to use the defining equations of TOI and TT in proving equivalences, it may be required to instantiate the variables in $e(v_1,\ldots,v_n)$ with some specific terms. For example, if the variable s in TOP(s) is instantiated to either NEWSTACK or PUSH(s′,x), it becomes possible to use the defining equations of TOP. It is, however, imperative to guarantee

---

[3] The notion of a syntactic kernel set is introduced only to circumvent the pathological undecidabilities that can arise in computing a "semantic" version of the kernel set.

that the generality of the overall proof procedure is not compromised by any such (set of) specialization(s). The most obvious way to ensure this generality is to use induction on the syntactic structure of the terms in the word algebra generated by F^T. For example, this would require considering the terms TOP(NEWSTACK), TOP(PUSH(s,x)), TOP(POP(s)), etc.

Unfortunately, even the specializations ensuing from such a set of instantiated terms may not be adequate to enable a completion of the required proofs. This will be the case if the type is not freely generated by the constructors, i.e., if the set of non-kernel constructors ($F^{TOI}-K^{TOI}-E^{TOI}$) is non-empty. Thus, in the case of the type Stack, POP is a non-kernel constructor, and there is no explicit equation of the form TOP(POP(s)) = ...

Nonetheless, it is possible to develop a proof procedure that uses induction only on the terms generated by a set of kernel functions, by recognizing (proving) the extraction equivalence of certain terms in the derived algebra. Proofs of extraction equivalence of terms in the derived algebra must in turn rely primarily on an induction on the structure of terms in W_{dTT}, but this often turns out to be feasible in practice. The resulting proof procedure is quite general; what is of greater relevance, however, is that it is more amenable to automation. Concluding this prologue, we now outline the proof procedure in greater detail.

We denote by $=_{dTT}$ extraction equivalence in the derived algebra dTT.

Theorem 8: Let R denote the set of defining equations of TOI. For each defining equation $t_1 = t_2$ in R, where t_1, t_2 are not in W_{TOI}, if

$$t_1 = t_2 \Rightarrow \theta(t_1) =_{dTT} \theta(t_2) \qquad \text{--(A)}$$

and if

$$(\text{for all } g \text{ in } G)\ (\text{for all } t_g \text{ in } W_g[K^{TOI} \cup E^{TOI}])\ t_g =_g \theta(t_g) \qquad \text{--(B)}$$

then θ defines a correct implementation.

Proof: See Appendix II.

It is crucial to note that the equation (B) above considers only $W_g[K^{TOI} \cup E^{TOI}$ and not $W_g[F^{TOI}]$.]

In order to prove $t_1 =_{dTT} t_2$, it is necessary to prove that

$$\text{(for all } g \text{ in } G) \text{ (for all } t_g \text{ in } W_g[F^{dTT}])$$
$$t_g[v \text{ in } V_{dTT}|t_1] =_g t_g[v \text{ in } V_{dTT}|t_2].$$

This proof may again be based upon induction on the structure of the terms in the word algebra W_{dTT}, and consists of the following steps:

Base case Prove

$$\text{(for all } g \text{ in } G) \text{ (for all } t_g \text{ in } W_g^{(0)}[F^{dTT}]$$
$$t_g[v \text{ in } V_{dTT}|t_1] = t_g[v \text{ in } V_{dTT}|t_2]$$

Assume (as the induction hypothesis)

$$\text{(for all } g \text{ in } G) \text{ (for all } t_g \text{ in } W_g^{(n)}[F^{dTT}]$$
$$t_g[v \text{ in } V_{dTT}|t_1] = t_g[v \text{ in } V_{dTT}|t_2]$$

Induction step Prove

$$\text{(for all } g \text{ in } G) \text{ (for all } t_g \text{ in } W_g^{(n+1)}[F^{dTT})$$
$$t_g[v \text{ in } V_{dTT}|t_1] = t_g[v \text{ in } V_{dTT}|t_2]$$

The proof of part (B) of Theorem 8 is again obtained by an induction on the terms of $W_{TOI}[K^{TOI} \text{ U } E^{TOI})$.

We now illustrate the proof method based on Theorem 8 by proving the correctness of the Stack implementation given in figure 2-3.

4. Illustrations of the Proof Method

4.1. Proof of an Implementation of a Stack

To prove the given implementation θ correct (see figure 2-3), it is necessary to prove that

$$\text{TOP}(s) = \theta(\text{TOP}(s)) \text{ for all } s \text{ in } W_{\text{Stack}} \qquad \text{---(S1)}$$

and

$$\text{ISEMPTY}(s) = \theta(\text{ISEMPTY}(s)) \text{ for all } s \text{ in } W_{\text{Stack}}. \qquad \text{---(S2)}$$

We will discuss only the proof of (S1) here. The proof of (S2) is almost identical.

Proof of (S1) The most natural form of a proof of (S1) relies on induction on the structure of the terms in $W_{Stack}[F^{Stack}]$, but involves the following proof:

$$(\text{for all } s \text{ in } W^{(n)}_{Stack})\ TOP(POP(s)) = \theta(TOP(POP(s)) \qquad \text{--(T-POP)}$$

Note however, that the defining equations for TOP apply only to terms of the form NEWSTACK or PUSH(s,x). Thus, (T-POP) cannot be proved directly. In general, equations that involve non-kernel functions cannot be proved directly by using the defining equations. Consequently, any syntactic equivalences that are implied by the defining equations for non-kernel functions (on Stack) must be proven to carry over as extraction equivalences in the (derived) implementation algebra. That is, we need to show that

$$POP(NEWSTACK) = NEWSTACK$$
$$\Rightarrow \theta(POP(NEWSTACK)) =_{dAI} \theta(NEWSTACK) \qquad \text{--(A1)}$$

and

$$POP(PUSH(s,x)) = s$$
$$\Rightarrow \theta(POP(PUSH(s,x))) =_{dAI} \theta(s). \qquad \text{--(A2)}$$

In such a case, by virtue of Theorem 2, it is sufficient to show that

$$TOP(s) = \theta(TOP(s)) \text{ for all } s \text{ in } W_{Stack}[K^{stack}],$$

where the kernel set for Stack is {NEWSTACK,PUSH}. This in turn can be proved by induction on the structure of terms in $W_{Stack}[K^{Stack}]$, and consists of the following steps:

Base Case Prove

$$TOP(NEWSTACK) = \theta(TOP(NEWSTACK)) \qquad \text{--(B1)}$$

Assume as the induction hypothesis that

$$(\text{for all } s \text{ in } W^{(n)}_{Stack}[K^{Stack}])\ TOP(s) = \theta(TOP(s))$$

Induction Step Prove

$$\text{for all } s \text{ in } W^{(n+1)}_{Stack}[K^{Stack}])TOP(PUSH(s,x) = \theta(TOP(PUSH(s,x))) \qquad \text{--(B2)}$$

We now detail some of these proofs.

Proof of (A1)

(LHS) $= \theta(POP(NEWSTACK))$
$= \theta(POP)(\theta(NEWSTACK))$
$= \theta(POP)\ (<NEWARRAY,ZERO>)$
$= <NEWARRAY,\ PRED(ZERO)>$
$= <NEWARRAY,\ ZERO>$ by the defining equation of PRED.

RHS $= \theta(NEWSTACK)$
$= <NEWARRAY,\ ZERO>$

$= \text{LHS}$

Since syntactic equivalence implies extraction equivalence, the proof of (A1) is complete.

Proof of (A2) By the definition of θ, we have,

$$\begin{aligned}
\text{LHS} &= \theta(\text{POP})\ (\theta(\text{PUSH}(s,x))) \\
&= \theta(\text{POP})\ (\theta(\text{PUSH})\ (\langle a,i\rangle,x)) \\
&= \theta(\text{POP})\ (\langle \text{ASSIGN}(a,\text{SUCC}(i),x),\text{SUCC}(i)\rangle) \\
&= \langle \text{ASSIGN}(a,\text{SUCC}(i),x),\ \text{PRED}(\text{SUCC}(I))\rangle \\
&= \langle \text{ASSIGN}(a,\text{SUCC}(i),x),i\rangle \quad (\text{by using } \text{PRED}(\text{SUCC}(i)) = i) \\
\text{RHS} &= \theta(s) = \langle a,i\rangle
\end{aligned}$$

Thus, we need to prove that the terms $\langle \text{ASSIGN}(a,\text{SUCC}(i),x),i\rangle$ and $\langle a,i\rangle$ are extraction equivalent in the derived target type algebra. These terms are not syntactically equivalent. Consequently, to prove the extraction equivalence of these two terms, we again need to resort to the basic definition and use induction on the structure of the terms in the derived algebra W_{dAI}, where we denote by dAI the derived Array-Index algebra. Observe that

$$W^{(0)}_{dAI}[\theta(F^{Stack})] = \langle \text{NEWARRAY},\text{ZERO}\rangle$$

$$\begin{aligned}
W^{(n+1)}_{dAI}[\theta(F^{Stack})] =\ &\{\langle \text{ASSIGN}(a,\text{SUCC}(i),x),\text{SUCC}(i)\rangle, \\
&\ \langle a,\text{PRED}(i)\rangle \mid \langle a,i\rangle \text{ in } W^{(n)}_{dAI}[\theta(F^{Stack})]\}
\end{aligned}$$

and that $\theta(E^{Stack}) = \{\theta(\text{TOP}),\ \theta(\text{ISEMPTY})\}$

A proof of (A2) by induction therefore consists of the following steps:

Base case

$$\begin{aligned}
&\theta(\text{TOP})\ (\langle \text{NEWARRAY},\text{ZERO}\rangle))\ [\langle a,i\rangle \mid \langle \text{ASSIGN}(a,\text{SUCC}(i),x1),i\rangle] \\
&\qquad = \theta(\text{TOP})\ (\langle \text{NEWARRAY},\text{ZERO}\rangle)[\langle a,i\rangle \mid \langle a,i\rangle] \qquad \text{--(A2-1)}
\end{aligned}$$

Induction hypothesis Assume

(for all $\langle a,i\rangle$ in $W^{(n)}_{dAI}$

$$\begin{aligned}
&\theta(\text{TOP})\ (\langle a,i\rangle)\ [\langle a,i\rangle \mid \langle \text{ASSIGN}(a,\text{SUCC}(i),x1),i\rangle] \\
&\qquad = \theta(\text{TOP})\ (\langle a,i\rangle)\ [\langle a,i\rangle \mid \langle a,i\rangle].
\end{aligned}$$

Induction step Prove
(for all $\langle a,i\rangle$ in $W^{(n+1)}_{dAI}$)

$$\theta(\text{TOP})\ (\langle \text{ASSIGN}(a,s(i),x),s(i)\rangle)\ [\langle a,i\rangle \mid \langle \text{ASSIGN}(a,\text{SUCC}(i),x1),i\rangle]$$

= θ(TOP) (<ASSIGN(a,s(i),x),SUCC(i)>|<a,i>) --(A2-2)

(for all <a,i> in $W_{dAI}^{(n+1)}$)

θ(TOP) (<a,PRED(i)>)[<a,i>|<ASSIGN(a,SUCC(i),x1),i>]
= θ(TOP) (<a,PRED(i)>) [<a,i>|<a,i>] --(A2-3)

In addition, proofs with θ(ISEMPTY) substituted for θ(TOP) must also be carried out. We illustrate only the proofs for θ(TOP), since the proof for θ(ISEMPTY) is similar. The proof of (A2-1) is trivial, since both the LHS and RHS are identical.

Proof of (A2-2)
LHS = θ(TOP) (<ASSIGN(ASSIGN(a,SUCC(i),x),SUCC(i)>)
= DATA(ASSIGN(ASSIGN(a,SUCC(i),x1),SUCC(i),x),SUCC(i))
= x (by the defining equations of DATA)

RHS = θ(TOP) (<ASSIGN(a,SUCC(i),x),SUCC(i)>)
= DATA(ASSIGN(a,SUCC(i),x),SUCC(i))
= x (by the defining equations of DATA)
= LHS

Proof of (A2-3)
LHS = θ(TOP) (<a,PRED(i)>) [<a,i>|<ASSIGN(a,SUCC(i),x1),i>]
= θ(TOP) (<ASSIGN(a,SUCC(i),x1), PRED(i)>)
= DATA(ASSIGN(a,SUCC(i),x1),PRED(i))
= DATA(A,PRED(i))

RHS = DATA(a,PRED(i)) = LHS

In conjunction with the proofs for θ(ISEMPTY), this completes the proof of (A2), and therefore of part (A).

Proof of (B1)
LHS = TOP(NEWSTACK) = UNDEFINED.
RHS = θ(TOP(NEWSTACK))
= θ(TOP) (θ(NEWSTACK))
= θ(TOP) (<NEWARRAY, ZERO>)
= DATA(NEWARRAY, ZERO)
= UNDEFINED
= LHS

Proof of (B2)
Let θ(s) = <a,i>
LHS = TOP(PUSH(s,x))
= x (by the defining equations for TOP)
RHS = θ(TOP(PUSH(s,x)))
= θ(TOP) (θ(PUSH(s,x)))
= θ(TOP) (θ(PUSH) (<a,i>,x))
= θ(TOP) (<ASSIGN(a,SUCC(i),x), SUCC(i)>)

= DATA(ASSIGN(a,SUCC(i),x), SUCC(i))
= x (by the defining equations for DATA).
= LHS

By Theorem 2, the above proofs of Part (A) and (B) together imply that θ defines a correct implementation of Stack.

5. Some comparisons with other proof methods

The conventional notion of a proof of the correctness of an implementation map θ involves proving the existence of a surjective homomorphism θ′ from W_{dTT}/E^{TT} onto W_{TOI}/E^{TOI}. Most of the proof methods that have been employed thus far are based primarily on this definition of correctness, and follow essentially either one of following two procedures:

(i) an "abstraction function" A: $W_{dTT} \rightarrow W_{TOI}$ is specified, which serves as a postulated map θ′. The correctness proof then involves showing that A does indeed define a surjective homomorphism. This method is basically due to Hoare [7]. The rep function used in the ALPHARD verification methodology serves a similar purpose [15].

(ii) an equality relation $=_{eq}$ (called an "equality interpretation" in [5]) is specified on the terms in W_{dTT}. The existence of the required homomorphic map θ′ is then proved by making use of this equality interpretation. This method is a slight generalization of (i), since an abstraction function can be used to impose an equality interpretation on dTT, whereas the converse is not true. Specifically, the equality interpretation induced by an abstraction function A is:

$$A(tt1) = A(tt2) \Rightarrow tt1 =_{eq} tt2.$$

Strictly speaking, however, in order to prove the correctness of an implementation of a type of interest TOI in terms of a target type TT, it should only be necessary to provide the following information:

1. a specification of the type being implemented TOI;
2. a specification of the representation type TT;

3. a specification of the implementation map θ.

It therefore detracts from the generality of a proof method if it is required to augment the specifications (1)-(3) above with some additional information in order to carry through a correctness proof. The existing methods, of which we have given some examples above, suffer from this drawback. In both of the above proof methods, it is necessary to supply some extra information--in the form of an abstraction function in (i), or an equality interpretation in (ii). This is also true of a recent proposal of Flon and Misra [2].

In contrast, the method we have outlined in this paper does not require any additional information augmenting the specifications (1)-(3). To make a specific comparison, if the proof techniques of [GHM78] are used, the proof of an implementation of a Stack identical to the one discussed in section 4.1 needs the following equality interpretation to be specified:

$$\begin{aligned}&\theta'(\langle a,i\rangle) =_{eq} \theta'(\langle a1,i1\rangle) = \\ &\quad \text{if } i=i1 \text{ and (for all } k\text{) } [1 \le k \le i = \mathrm{DATA}(a,i) = \mathrm{DATA}(a1,i)]\end{aligned}$$

As we indicated in section 1, the added generality of our proof procedure is quite important, since it facilitates automation. (For example, all of the proofs presented in this paper have been automated using the simplifier that forms part of the Stanford Verifier [9].) Of course, it is possible that in the course of a particular proof, some specific step cannot be carried through automatically, just as it is possible that in the course of attempting a correctness proof of a program using, say, Floyd-Hoare proof methods (cf. [3], [6],) it may prove to be difficult (or infeasible) in practice to demonstrate the invariance of certain assertions. However, our initial empirical explorations with an automated system have certainly served to indicate that the method can be used to carry out non-trivial proofs, thereby lending credibility to its pragmatic utility.

I. Proof of Theorem 7

We restate the theorem below.

<u>Theorem 7</u> An implementation map θ such that

$$\text{(for all } g \text{ in } G)(\text{for all } t_g \text{ in } W_g[F^{TOI}])[\theta(t_g) =_g t_g] \qquad \text{(I)}$$

implies the existence of a surjective homomorphism

$$\theta' : W_{dTT} / E^{TT} \rightarrow W_{TOI} / E^{TOI}$$

where W_{dTT}/E^{TT} (respectively W_{TOI}/E^{TOI},) denotes the extraction equivalence classes induced by the functions E^{TT} (respectively E^{TOI}).

The proof of this theorem rests on lemma 9 below. Let [t] denote the equivalence class of the term t.

Lemma 9: Let $\theta(\tilde{t}) = t$, $\tilde{t}$ in W_{TOI}. Define $\theta' : W_{dTT} \rightarrow W_{TOI}$, where $\theta'([t]) = [t]$. Then θ′ is a well defined map.

Proof. In order for θ′ to be well defined, it needs to be shown that

(a) If $\tilde{t}$ is such that

$$\theta(\tilde{t}) = t \qquad (1)$$

then there must not exist $\tilde{t}' \neq_{TOI} \tilde{t}$ such that

$$\theta(\tilde{t}') = t \qquad (2)$$

(b) θ′ is defined for all [t] in W_{dTT}/E^{dTT}.

<u>Proof</u> <u>of</u> <u>part</u> (<u>a</u>) Assume that there exists a $\tilde{t}' \neq_{TOI} \tilde{t}$ such that $\theta(\tilde{t}') = t$. Then, by the definition of extraction equivalence, there must exist t_g in $W_g[F^{TOI}]$ such that

$$t_g[v \ \underline{\text{in}} \ V_{TOI} | \tilde{t}] \neq_g t_g[v \ \underline{\text{in}} \ V_{TOI} | \tilde{t}'] \qquad (3)$$

Intuitively, this implies the existence of a sequence of function applications, terminating in the application of an extraction function, that yields inequivalent results when applied to $\tilde{t}$ and $\tilde{t}'$. But, by the definition of θ and constraint (I) of the theorem,

$$t_g[v \ \underline{\text{in}} \ V_{TOI} | \tilde{t}] =_g \theta(t_g[v \ \underline{\text{in}} \ V_{TOI} | \tilde{t}]) \qquad (4)$$

and

$$t_g[v \ \underline{\text{in}} \ V_{TOI} | \tilde{t}'] =_g \theta(t_g[v \text{ in } V_{TOI} | \tilde{t}']) \qquad (5)$$

By the definition of θ,

$$\theta(t_g[v\ \underline{\text{in}}\ V_{TOI}|\tilde{t}]) =_g \theta(t_g)[v\ \underline{\text{in}}\ V_{TT}|\ \theta(\tilde{t})]$$

and

$$\theta(t_g[v\ \underline{\text{in}}\ V_{TOI}|\tilde{t}']) =_g \theta(t_g)[v\ \text{in}\ V_{TT}|\theta(\tilde{t}')]$$

(3), (4) and (5) imply

$$\theta(t_g)[v\ \underline{\text{in}}\ V_{TT}|\theta(\tilde{t})] \neq_g \theta(t_g)[v\ \underline{\text{in}}\ V_{TT}|\theta(\tilde{t}')] \qquad (6)$$

where $\theta(t_g)$ is in $W_g[F^{dTT}\ U\ F^G, V_{TT}]$.

But (1) and (2) together imply

$$\theta(\tilde{t}) =_{dTT} \theta(\tilde{t}')$$

and consequently, we have

$$(\text{for all } g \text{ in } G)\ (\text{for all } t_g\ \underline{\text{in}}\ W_g[F^{dTT}\ U\ F^G, V_{TT}])$$
$$t_g[v\ \underline{\text{in}}\ V_{TT}|\theta(\tilde{t})] =_g t_g[v\ \text{in}\ V_{TT}|\theta(\tilde{t}')] \qquad (7)$$

which contradicts (6). Hence the assumption that there exists a $\tilde{t}' \neq_{TOI} \tilde{t}$ and such that $\theta(\tilde{t}') = t$ cannot be true. End of Proof.

Proof of Part (b).

By virtue of definition 4, the only terms in dTT are those that images under θ of some term in W_{TOI}. There must therefore exist at least one term $\tilde{t}$ in $W_{TOI}[F^{TOI}\ U\ F^G]$ which the pre-image of t under θ. That is, θ' is defined for every term t in dTT. This completes the proof of the Lemma.

End of Proof.

Proof of the theorem

Consider the map θ' defined in lemma 9. In order to prove the theorem, it needs to be shown that

(A) θ' is onto W_{TOI}/E_{TOI},

(B) θ' is a homomorphism.

Proof of Part (A) To prove that θ' is onto, we have to show that for every $[\tilde{t}]$ in W_{TOI}/E^{TOI}, there is a term in W_{dTT}/E^{dTT} that maps onto $[\tilde{t}]$.

Since, for every term $\tilde{t}$ in W_{TOI}, $\theta(\tilde{t})$ is in W_{dTT}, by definition of θ', we must have,

$$\theta'([\theta(\tilde{t})]) =_{TOI} [\tilde{t}].$$

The proof of part (A) follows immediately.

End of Proof.

Proof of Part (B) We need to show that

$$\theta'([f(\underline{t}')]) =_{TOI} \theta'([f'])(\theta([\underline{t}'])) \tag{8}$$

where f' is a function in dTT, and $\underline{t}'$ represents a tuple of terms.

Let $\tilde{f}'$, $\underline{\tilde{t}}'$ be such that

$$\theta(\tilde{f}') =_{TT} f'$$

and

$$\theta(\underline{\tilde{t}}') =_{TT} \underline{t}'.$$

(Because of the reasons given in the proof of part (b) of the lemma, such a pair $\tilde{f}'$, $\underline{\tilde{t}}'$ must exist.) By definition of θ', we have,

$$\theta'([f']) =_{TOI} [\tilde{f}']$$

and

$$\theta'([\underline{t}']) =_{TOI} [\underline{\tilde{t}}']$$

Thus,

$$\theta'([f'])(\theta'([\underline{t}'])) =_{TOI} [\tilde{f}'[\underline{\tilde{t}}')] \tag{9}$$

Again, by definition of θ,

$$\theta(\tilde{f}'(\underline{\tilde{t}}')) =_{TT} \theta(\tilde{f}')(\theta(\underline{\tilde{t}}')) =_{TT} f'(\underline{t}')$$

Thus, by definition of θ',

$$\theta'([f'(\underline{t}')]) =_{TOI} [\tilde{f}'(\underline{\tilde{t}}')] \tag{10}$$

Together, (9) and (10) imply that θ' satisfies the homomorphism condition (8), thus proving the theorem.

End of Proof.

II. Proof of Theorem 8

We restate the theorem below.

Theorem 8: Let R denote the set of defining equations of TOI. For each defining equation $t_1 = t_2$ in R, where t_1, t_2 are not in W_{TOI}, if

$$t_1 = t_2 \Rightarrow \theta(t_1) =_{dTT} \theta(t_2) \qquad \text{--(A)}$$

and if

$$(\text{for all } g \text{ in } G)\ (\text{for all } t_g \text{ in } W_g[K^{TOI}])\ U\ E^{TOI}])\ t_g = {}_g\theta(t_g) \qquad \text{--(B)}$$

then θ defines a correct implementation.

We first prove four lemmas which formalize some fairly intuitive facts, and which are needed in the proof of Theorem 8.

About the lemma 10. This lemma states that

- if a term t_2 is obtained by instantiating a term t by substituting t′ for the variables of sort T, where
- t′ itself has been obtained by instantiating t″ by substituting t_1 for the variables of sort T, then
- t_2 can also be obtained directly by substituting t_1 for variables of sort T in some t_2'; the term t_2' is actually constructed in the proof of the lemma.

Lemma 10:

Consider t, t_1 in W_T. If $t_2 = [v \text{ in } V_T | t']$ and $t' = t''[v \text{ in } V_T | t_1]$, then there exists t_2' such that $t_2 = t_2'[v \text{ in } V_T | t_1]$.

Proof. The proof is by induction on the structure of t.

(a) Base Case. Let t be in $W_T^{(0)}$.

t in $W_T^{(0)} \Rightarrow t = v$ or $t = f$, where f is in BC^T.

$t = v \Rightarrow t_2 = t' = t''[v \text{ in } V_T | t_1]$.

Hence $t_2' = t''$. If $t_2 = f$ then $t_2' = f$.

(b) Induction Step Assume that the proposition holds for all t in $W_T^{(n-1)}$. Consider t in $W_T^{(n)}$. Then t must be of the form $t = f(x_1, \ldots, x_m)$ where $f : (X_1, \ldots, X_m) \to T$, and x_i in $W_{X_i}^{(n-1)}$ (and such that at least one x_i is not in

$W_{X_i}^{(n-2)}$). Variables of sort T can then occur in $x_1, \ldots, x_m$.

$$\begin{aligned} t_2 &= t[v \text{ in } V_T | t'] \\ &= f(x_1[v \text{ in } V_T|t'], \ldots, x_m[v \text{ in } V_T|t']) \\ &= f(x_1'[v \text{ in } V_T|t_1], \ldots, x_m'[v \text{ in } V_T|t_1]) \qquad \text{(by hypothesis)} \\ &= f(x_1', \ldots, x_m')\ [v \text{ in } V_T|t_1] \end{aligned}$$

Hence $t_2' = f(x_1', \ldots, x_m')$, which completes the proof.

End of Proof.

Lemma 11 states that the terms t_3, t_4 obtained from a common term by instantiating variables with extraction equivalent terms are themselves extraction equivalent (although they might be syntactically distinct). This is illustrated in Figure 5-1.

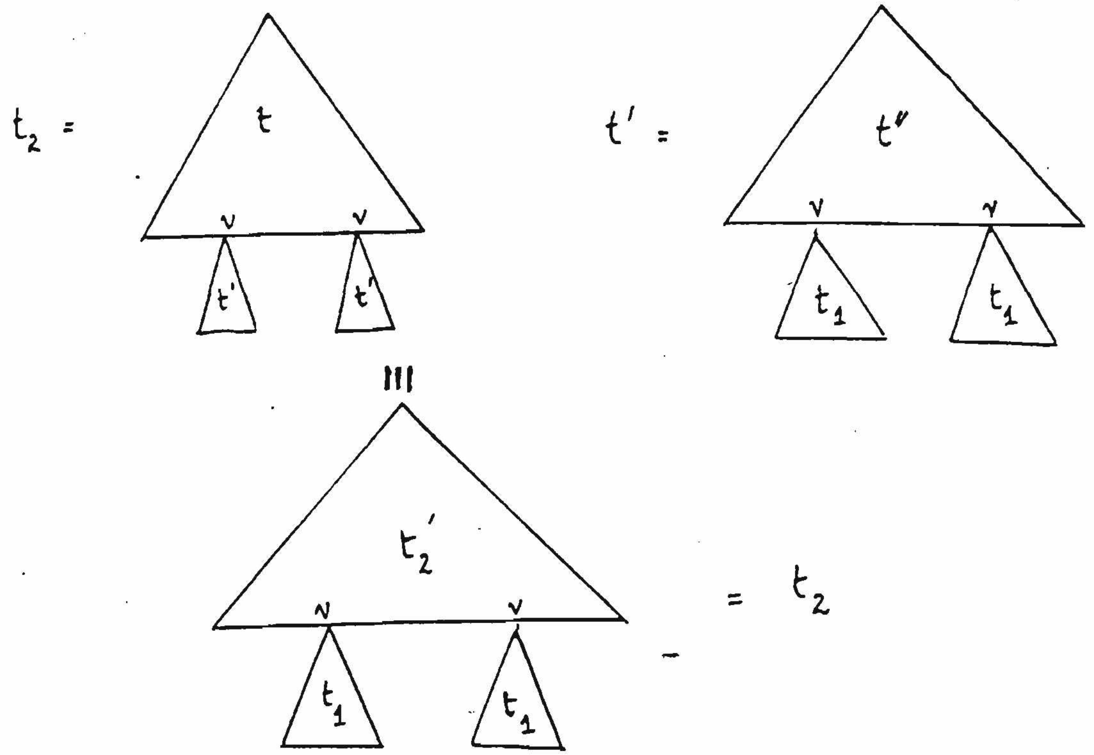

Figure 5-1: Figure illustrating Lemma 11

Lemma 11: Consider t_1, t_2 in W_T. Let $t_3 = t'[v \text{ in } V_T|t_1]$, and t_4 =

$t'[v \text{ in } V_T|t_2]$. Then $t_1 =_T t_2 \Rightarrow t_3 =_T t_4$ (where $=_T$ denotes extraction equivalence).

Proof.

$t_3 =_T t_4$

$\Leftrightarrow$ (for all g in G) (for all t_g in W_g)

$$t_g[v \text{ in } V_T\ |t_3] =_g t_g[v \text{ in } V_T|t_4].$$

By lemma 10, there is some t_g'' in W_g such that $t_g[v \text{ in } V_T|t_3] =_g t_g''[v \text{ in } V_T|t_1]$, and $t_g[v \text{ in } V_T|t_4] = t_g''[v \text{ in } V_T|t_2]$.

Since $t_1 =_T t_2$, it follows that $t_3 =_T t_4$. <u>End of Proof.</u>

Lemma 12: Consider t_1, t_2 in W_T. Then $t_1 = t_2 \Rightarrow t_g[v \text{ in } V_T|t_1] =_g t_g[v \text{ in } V_T|t_2]$, where t_g is in W_g.

Proof: Immediate, from lemma 11, since syntactic equivalence implies extraction equivalence.

<u>End of Proof.</u>

Lemma 13: For all t_g in $W_g[F^T]$, there is a term t in $W_T[FT]$, and a term t_g' in $W_g[E^T]$, such that $t_g = t_g'[v \text{ in } V_T|t]$.

Proof. Every term t_g is of the form $e(t_1,\ldots,t_n)$ where e is in E^T, $e:X_1,\ldots,X_n \to g$, and t_i is in $W_{X_i}[F^T]$. Consider the term $t_g' = e(v_1,\ldots,v_n)$, v_i in V_{X_i}. Then t_g' is in $W_g[E^T]$, and

$$t_g =_g t_g'[v_1|t_1]..[v_n|t_n].$$

<u>End of Proof.</u>

<u>Proof of the Theorem.</u>

By virtue of the definition of K^{TOI}, for every t in $W_{TOI}[F^{TOI}]$ there exists some t_1 in $W_{TOI}[K^{TOI}]$, such that $t = t_1$.

By Lemma 12, it follows that

$$t_g[v|t,\ t \text{ in } W_{TOI}] =_g t_g[v|t_1,\ t_1 \text{ in } W_{TOI}[K^{TOI}]] \qquad (1)$$

Also,

$$\theta(t_g[v|t]) =_g \theta(t_g)[\theta(v)|\theta(t)] \qquad (2)$$

By virtue of (A), we have

$$t = t_1 \Rightarrow \theta(t) =_{dTT} \theta(t_1)$$

Consequently, (2) $\Rightarrow$

$\theta(t_g[v|t,\ t \text{ in } W_{TOI}[F^{TOI}]])$

$= \theta(t_g)[\theta(v)|\theta(t),\ t \text{ in } W_{TOI}[F^{TOI}]]$

$$= \theta(t_g)[\theta(v)|\theta(t_1),\ t_1 \text{ in } W_{TOI}[K^{TOI}],\ t = t_1] \qquad (3)$$

Again, by virtue of (B), we have

$$\text{(for all } g \text{ in } G\text{) (for all } t_g \text{ in } W_g[K^{TOI} \cup E^{TOI}]\text{)}$$
$$t_g[\underline{v}|\underline{t},\ \underline{t} \text{ in } W_{TOI}[K^{TOI}]] = \theta(t_g[\underline{v}|\underline{t},\ \underline{t} \text{ in } W_{TOI}[K^{TOI}]]) \qquad (4)$$

From (1), (2), (3) and (4), we obtain
(for all g in G) (for all t_g in $W_g[K^{TOI} \cup E^{TOI}]$)

$t_g[\underline{v}|\underline{t},\ \underline{t} \text{ in } W_{TOI}[F^{TOI}]]$

$= t_g[\underline{v}|\underline{t}_1,\ \underline{t}_1 \text{ in } W_{TOI}[K^{TOI}],\ \underline{t} = \underline{t}_1]$ by (1)

$= \theta(t_g[\underline{v}|\underline{t}_1,\ \underline{t}_1 \text{ in } W_{TOI}[K^{TOI}]])$ by (4)

$= \theta(t_g[\underline{v}|\underline{t},\ \underline{t} \text{ in } W_{TOI}[F^{TOI}]])$ by (2) and (3)

i.e.,

$$\text{(for all } g \text{ in } G\text{) (for all } t_g \text{ in } W_g[K^{TOI} \cup E^{TOI}]\text{)}$$
$$t_g[\underline{v}|\underline{t},\ \underline{t} \text{ in } W_{TOI}[F^{TOI}]]$$
$$= \theta(t_g[\underline{v}|\underline{t},\ \underline{t} \text{ in } W_{TOI}[F^{TOI}]]) \qquad (5)$$

But by lemma 13, t_g can be expressed as $t_g'[v|t',\ t' \text{ in } W_{TOI}[F^{TOI}]]$ where t_g' is in $W_g[E^{TOI}]$ (and hence in $W_g[K^{TOI} \cup E^{TOI}]$). Consequently, (5) implies that

$$\text{(for all } g \text{ in } G\text{) (for all } t_g \text{ in } W_g[F^{TOI}]\text{)}$$
$$t_g[\underline{v}|\underline{t},\ \underline{t} \text{ in } W_{TOI}[F^{TOI}]] = \theta(t_g[\underline{v}|\underline{t},\ \underline{t} \text{ in } W_{TOI}[F^{TOI}]]) \qquad (6)$$

But (6) is precisely the condition required for correctness of the implementation specified by θ. (Note that the key difference lies in the quantification of the terms t_g.) This proves the theorem.

End of Proof.

III. Definitions of the types Array and Integer

```
Type Integer

Syntax

        ZERO : () -> Integer
        SUCC : (Integer) -> Integer
        PRED : (Integer) -> Integer
        ISZERO : (Integer) -> Boolean

Semantics

for all i in Integer

        ISZERO(ZERO()) = TRUE
        ISZERO(SUCC(i)) = FALSE

        PRED(ZERO()) = ZERO
        PRED(SUCC(i)) = i

End Integer
```

Figure 5-2: Definition of the type Integer

```
Type Array

Generic type parameter : item

Syntax

        NEWARRAY : () -> Array
        ASSIGN : (Array,Integer,Item) -> Array
        DATA : (Array,Integer) -> Item U {UNDEFINED}

Semantics

        DATA(NEWARRAY,j) = UNDEFINED
        DATA(ASSIGN(a,i,x),j) = if i=j then x else DATA(a,j)

end Array
```

Figure 5-3: Definition of the type Array

IV. The Proof of a Queue Implementation

Consider the implementation of the type Queue (see figure 5-4,) using a target type consisting of the triple <Array, Integer, Integer>. Intuitively, the first integer component points to the front of the Queue, while the second integer component points to the tail of the Queue. The implementation map θ for the functions on type Queue is given in figure 5-5.

We note that

BC^{Queue} = {NEWQ}.
C^{Queue} = {ADDQ, DELETEQ}.
E^{Queue} = {FRONTQ, ISEMPTYQ}.

The correctness proof consists of two parts.

(A) The syntactic equivalence induced on the terms of type Queue by the defining equations must be shown to produce extraction equivalent terms in the implementation algebra dAII under the map θ. That is,

DELETEQ(NEWQ) = NEWQ => θ(DELETEQ(NEWQ)) $=_{dAII}$ θ(NEWQ) --(A1)

DELETE(ADDQ(q,x)) = if ISEMPTYQ(q)
then NEWQ
else ADDQ(DELETEQ(q),x)

=> θ(DELETEQ(ADDQ(q,x))) $=_{dAII}$
θ(if ISEMPTYQ(q)
then NEWQ
else ADDQ(DELETEQ(q),x)) --(A2)

(B) By induction on W_g, it must be proved that
(for all g in G)
(for all t_g in W_g[{NEWQ,ADDQ,ISEMPTYQ,FRONTQ} U F^G,V])
t_g= $\theta(t_g)$ --(B)

This involves the following proofs:

Base Case

FRONTQ(NEWQ) = θ(FRONTQ(NEWQ)) --(F1)
ISEMPTYQ(NEWQ) = θ(ISEMPTYQ(NEWQ)) --(I1)

Induction Step

(for all q in $W^{(n+1)}_{Queue}$[(K^{Queue} U E^{Queue}, V])

```
Type Queue

Syntax
        NEWQ : () -> Queue
        ADDQ : (Queue,Item) -> Queue
        DELETEQ : (Queue) -> Queue
        FRONTQ : (Queue) -> Item
        ISEMPTYQ : (Queue) -> Boolean

Semantics

for all q, q1 in Queue, x in Item;

        DELETEQ(NEWQ) = NEWQ
        DELETEQ(ADDQ(q,x)) = if q = NEWQ
                             then NEWQ
                             else ADDQ(DELETEQ(q),x)

        ISEMPTYQ(NEWQ) = TRUE
        ISEMPTYQ(ADDQ(q,x)) = FALSE

        FRONTQ(NEWQ) = UNDEFINED
        FRONTQ(ADDQ(q,x)) = if q = NEWQ
                            then x
                            else  FRONTQ(q)

End Queue
```

Figure 5-4: Definition of the Type Queue

```
We write θ(q) = <a,l,h>

θ(NEWQ) = <NEWARRAY, ZERO, ZERO>
θ(ADDQ(q,x)) = <ASSIGN(a, SUCC(h), x), l, SUCC(h)>
θ(DELETEQ(q)) = if l = h
                then <NEWARRAY, ZERO, ZERO>
                else <a, SUCC(l), h>
θ(FRONTQ(q)) = if l = h
               then UNDEFINED
               else DATA(a, SUCC(l))
θ(ISEMPTYQ(q)) = (l=h)
```

Figure 5-5: An Implementation of the Type Queue

```
FRONTQ(ADDQ(q,x)) = θ(FRONTQ(ADDQ(q,x)))                         --(F2)
ISEMPTYQ(ADDQ(q,x)) = θ(ISEMPTYQ(ADDQ(q,x)))                     --(I2)
```

<u>Proof of (A1)</u>

```
LHS     = θ(DELETEQ(NEWQ))
        = θ(DELETEQ)(θ(NEWQ))
        = θ(DELETEQ) (<NEWARRAY,ZERO,ZERO>)
        = if ZERO = ZERO
          then <NEWARRAY,ZERO,ZERO>
          else <NEWARRAY,SUCC(ZERO),ZERO>
        = <NEWARRAY,ZERO,ZERO>

RHS     = <NEWARRAY,ZERO,ZERO>
        = LHS
```

Since syntactic equivalence implies extraction equivalence, this completes the proof of (A1).

<u>Proof of (A2)</u>

```
LHS     = θ(DELETEQ(ADDQ(q,x))
        = θ(DELETEQ) (<ASSIGN(a,SUCC(h),x),l,SUCC(h)>)
        = if l=SUCC(h)
          then <NEWARRAY,ZERO,ZERO>
          else  <ASSIGN(a,SUCC(h),x),SUCC(l),SUCC(h)>
        = <ASSIGN(a,SUCC(h),x),SUCC(l),SUCC(h)>
```

(where we use the fact that $l \leq h$ is true in any term <a,l,h> in W_{dAII}. This is proved below.)

```
RHS     = if ISEMPTYQ(q) then θ(NEWQ)
                 else θ(ADDQ)(θ(DELETEQ (q),x))
        = if l=h
          then <NEWARRAY,ZERO,ZERO>
          else θ(ADDQ(if l=h then <NEWARRAY,ZERO,ZERO>
                               else <a,SUCC(l),h>),x))
        = if l=h then <NEWARRAY,ZERO,ZERO>
          else if l=h then θ(ADDQ)(<NEWARRAY,ZERO,ZERO,x)
                      else θ(ADDQ)(<a,SUCC(l),h>,x)
        = if l=h then <NEWARRAY,ZERO,ZERO>
                 else θ(ADDQ) (<a,SUCC(l),h>,x)
        = if l=h then <NEWARRAY,ZERO,ZERO>
                 else <ASSIGN(a,SUCC(h),x),SUCC(l),SUCC(h)>
```

The proof of (A2) involves a proof by induction.

<u>Base Case</u>

```
θ(FRONTQ)  (<NEWARRAY,ZERO,ZERO>) =
                              θ(FRONTQ) (<NEWARRAY,ZERO,ZERO>)

θ(ISEMPTYQ) (<NEWARRAY,ZERO,ZERO>) =
                              θ(ISEMPTYQ) (<NEWARRAY,ZERO,ZERO>)
```

Induction hypothesis

```
θ(FRONTQ) (<a,l,h>) [<a,l,h>|
                       <ASSIGN(a,SUCC(h),xl),SUCC(l),SUCC(h)>]
= θ(FRONTQ)(<a,l,h>)
              [<a,l,h>|
                  if l=h
                  then <NEWARRAY,ZERO,ZERO>
                  else <ASSIGN(a,SUCC(h),xl),SUCC(l),SUCC(h)>]

θ(ISEMPTYQ) (<a,l,h>) [<a,l,h>|
                          <ASSIGN(a,SUCC(h),xl),SUCC(l),SUCC(h)>]
= θ(ISEMPTYQ)(<a,l,h>)
                [<a,l,h>|
                   if l=h
                   then <NEWARRAY,ZERO,ZERO>
                   else <ASSIGN(a,SUCC(h),xl),SUCC(l),SUCC(h)>]
```

Induction step

Prove

```
θ(FRONTQ) (<ASSIGN(a,SUCC(h),x),l,SUCC(h)>)
        [<a,l,h>|<ASSIGN(a,SUCC(h),xl), SUCC(l),SUCC(h)>]
= θ(FRONTQ) (<ASSIGN(a,SUCC(h),x),l,SUCC(h)>)
       [<a,l,h>|
           if l=h
           then <NEWARRAY,ZERO,ZERO>
           else <ASSIGN(a,SUCC(h),xl),SUCC(l),SUCC(h)>]                --(A2-1)
```

and

```
θ(FRONTQ) (if l=h
           then <NEWARRAY,ZERO,ZERO>
           else <a,SUCC(l),h>)
                [<a,l,h>|<ASSIGN(a,SUCC(h),xl), SUCC(l),SUCC(h)>]
= θ(FRONTQ)(if l=h
            then <NEWARRAY,ZERO,ZERO>
            else <a,SUCC(l),h>)
             [<a,l,h>|
                if l=h
                then <NEWARRAY,ZERO,ZERO>
                else <ASSIGN(a,SUCC(h),xl),SUCC(l),SUCC(h)>]
                                                                       --(A2-2)


LHS     = θ(FRONTQ) (<ASSIGN(ASSIGN(a,SUCC(h),xl),
                              SUCC(SUCC(h)),
                              x),
                         SUCC(l),
                         SUCC(SUCC(h))>)
        = if SUCC(l) = SUCC(SUCC(h))
          then UNDEFINED
```

```
          else DATA(ASSIGN(ASSIGN(a,SUCC(h),xl),
                            SUCC(SUCC(h)),
                            x),
                     SUCC(SUCC(l)))
        = if SUCC(SUCC(l)) = SUCC(SUCC(h))
          then x
          else if SUCC(h) = SUCC(SUCC(l))
               then xl
               else DATA(a,SUCC(SUCC(l)))
                (using  the invariant l ≤ h)

RHS     = if (l=h)
          then θ(FRONTQ) (<ASSIGN(NEWARRAY,SUCC(ZERO),x),
                            ZERO,SUCC(ZERO)>)
          else θ(FRONTQ) (<ASSIGN(ASSIGN(a,SUCC(h),xl),
                                   SUCC(SUCC(h),x),
                            SUCC(l),SUCC(SUCC(h))>)
        =if (l=h)
         then if ZERO=SUCC(ZERO)
              then UNDEFINED
              else (DATA(ASSIGN(NEWARRAY,SUCC(ZERO),X),
                         SUCC(ZERO))
         else if SUCC(l) = SUCC(SUCC(h))
              then UNDEFINED
              else DATA(ASSIGN(ASSIGN(a,SUCC(h),xl),
                                 SUCC(SUCC(h)),x),
                         SUCC(SUCC(l)))
```

Using the fact that ZERO is not equal to SUCC(ZERO), the definition of DATA, and l ≤ h => SUCC(l) ≠ SUCC(SUCC(h)), we get

```
RHS
= if l=h
  then x
  else DATA(ASSIGN(ASSIGN(a,SUCC(h),xl),SUCC(SUCC(h)),x),
                      SUCC(SUCC(l)))
= if l=h then x
  else if l=h then x
       else if SUCC(h) = SUCC(SUCC(l)) then xl
            else DATA(a,SUCC(SUCC(l)))
= if l=h then x
  else if SUCC(h) = SUCC(SUCC(l))
       then xl
       else DATA(a,SUCC(SUCC(l)))
```

Thus LHS = RHS.

This completes the proof of (A2-1). The proof of (A2-2) can be carried through similarly.

Proof of (F1) : FRONTQ(NEWQ) = θ(FRONTQ(NEWQ))

LHS = FRONTQ(NEWQ) = UNDEFINED.

RHS = DATA(NEWARRAY, SUCC(ZERO)) = UNDEFINED = RHS

<u>Proof of (F2)</u> FRONTQ(ADDQ((q,x)) = θ(FRONTQ(ADDQ(q,x)))

LHS = FRONTQ(ADDQ(q,x)) = if ISEMPTYQ(q)
then x
else FRONTQ(q).

RHS = θ(FRONTQ(ADDQ(q,x)))
= θ(FRONTQ)(<ASSIGN(a,SUCC(h),x),l,SUCC(h)>)
= if l=SUCC(h) then UNDEFINED
else DATA(ASSIGN(a,SUCC(h),x),SUCC(l))).

This proof needs a case analysis. The two cases on the LHS are

ISEMPTYQ(q) : x; --(F2-L1)
not ISEMPTYQ(q) : FRONTQ(q) --(F2-L2)

On the RHS, there are again two cases

l=SUCC(h) : UNDEFINED; --(F2-R1)
not l=SUCC(h) : DATA(ASSIGN(a,SUCC(h),x),SUCC(l))); --(F2-R2)

In order to complete the proof, we can assume the following as induction hypotheses:

FRONTQ(q) = θ(FRONTQ(q)) = DATA(a,SUCC(l))
ISEMPTYQ(NEWQ) = θ(ISEMPTYQ(NEWQ)) = TRUE
ISEMPTYQ(q) = θ(ISEMPTYQ(q)) = (l=h)

By definition of θ(ISEMPTYQ(q)), and the induction hypothesis,

LHS of (F2-L1) = ISEMPTYQ(q) => (l=h) => not (l=SUCC(h)),

hence the second case (F2-R2) on the RHS applies.

Further, (l=h) => DATA(ASSIGN(a,SUCC(h),x),SUCC(l)) = x;

Thus,
ISEMPTYQ(q) => FRONTQ(ADDQ(s,x)) = x, and --(1)
ISEMPTYQ(q) => not (l=SUCC(h)) & (l=h)
=> θ(FRONTQ(ADDQ(q,x))) = x --(2)

Again, not ISEMPTYQ(q) => not (l=h), and
not ISEMPTYQ(q) => FRONTQ(ADDQ(q,x)) = FRONTQ(q)

By the induction hypothesis,

FRONTQ(q) = θ(FRONTQ(q)) = DATA(a,SUCC(l)). --(3)

If we use the fact that l $\leq$ h is an invariant in the derived algebra (see below), the it can never be the case that $l = \mathrm{SUCC}(h)$. Hence, we have

$$\begin{aligned}
&(l \neq \mathrm{SUCC}(h))\ \&\ (l \neq h) \Rightarrow\\
&\text{RHS of (F2-R2)} = \mathrm{DATA}(\mathrm{ASSIGN}(a,\mathrm{SUCC}(h),x),\mathrm{SUCC}(l)\\
&\qquad\qquad = \text{if } \mathrm{SUCC}(h) = \mathrm{SUCC}(l)\\
&\qquad\qquad\quad \text{then } x\\
&\qquad\qquad\quad \text{else } \mathrm{DATA}(a,\mathrm{SUCC}(l))\\
&\qquad\qquad = \mathrm{DATA}(a,\mathrm{SUCC}(l)). && \text{--(4)}
\end{aligned}$$

The proof of (F2) Follows by virtue of (1) and (3) and (4).

Proof of the invariance of $l \leq h$.

The proof is by induction on the structure of the terms of the derived algebra.

Base case The base constructors form the set of terms in $W^{(0)}_{\mathrm{Queue}}[F^{\mathrm{Queue}},V]$. The invariant must be verified for each base constructor (there is only one). We have

$$\begin{aligned}
\theta(\mathrm{NEWQ}) &= \langle \mathrm{NEWARRAY},\ \mathrm{ZERO},\ \mathrm{ZERO}\rangle.\\
l &= \mathrm{ZERO} \leq \mathrm{ZERO} = h.
\end{aligned}$$

Induction step If $\theta(q) = \langle a,l,h\rangle$ then assume as the induction hypothesis $l \leq h$, if q is in $W^{(n)}_{\mathrm{Queue}}$, and is obtained by applying a constructor function to terms in $W^{(n-1)}_{\mathrm{Queue}}[F^{\mathrm{Queue}},V]$.

$$\begin{aligned}
&\theta(\mathrm{ADDQ}(q,x)) = \langle \mathrm{ASSIGN}(a,\mathrm{SUCC}(h),x),l,\ \mathrm{SUCC}(h)\rangle\\
&l \leq h \Rightarrow l \leq \mathrm{SUCC}(h)
\end{aligned}$$

$$\begin{aligned}
&\theta(\mathrm{DELETEQ}(q)) = \text{if } l=h\\
&\qquad\qquad\qquad \text{then } \langle a,\mathrm{ZERO},\ \mathrm{ZERO}\rangle\\
&\qquad\qquad\qquad \text{else } \langle a,\mathrm{SUCC}(l),h\rangle\\
&l \leq h\ \&\ l=h \Rightarrow \mathrm{ZERO} \leq \mathrm{ZERO}\\
&l \leq h\ \&\ \text{not } l=h \Rightarrow l < h \Rightarrow \mathrm{SUCC}(l) \leq h
\end{aligned}$$

Thus, in both cases, the condition ($l \leq h$) is preserved, concluding the proof.

Proof of (I1)

$$\mathrm{ISEMPTYQ}(\mathrm{NEWQ}) = \theta(\mathrm{ISEMPTYQ}(\mathrm{NEWQ})) \qquad \text{--(I1)}$$

LHS = true.
RHS = (ZERO = ZERO) = true.

Proof of (I2)

$$\mathrm{ISEMPTYQ}(\mathrm{ADDQ}(q,x)) = \theta(\mathrm{ISEMPTYQ}(\mathrm{ADDQ}(q,x))) \qquad \text{--(I2)}$$

LHS = false.
RHS = $\theta(\mathrm{ISEMPTYQ})(\langle \mathrm{ASSIGN}(a,\mathrm{SUCC}(h),x),l,\ \mathrm{SUCC}(h)\rangle)$

= (l = SUCC(h))
= false
(Using the fact that (l ≤ h))

REFERENCES

[1] O.J.Dahl, E.W.Dijkstra, C.A.R.Hoare.
Structured Programming.
Academic Press, New York, 1972.

[2] L.Flon, J.Misra.
A Unified Approach to the Specification and Verification of Abstract Data Types.
In *Proceedings of a Conference on Specifications of Reliable Software*, pages 162-169. IEEE Computer Society, April, 1979.

[3] R.W.Floyd.
Assigning Meanings to Programs.
In J.T.Schwartz, editor, *Proceedings of a Symposium in Applied Mathematics, Vol. 19*, pages 19-32. American Mathematical Society, 1967.

[4] J.Goguen, J.Thatcher,E.Wagner.
An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types.
Prentice-Hall, N.J, 1979, pages 80-149.

[5] J.Guttag, E.Horowitz, D.Musser.
Abstract Data Types and Software Validation.
CACM 21:1048-64, 1978.

[6] C.A.R.Hoare.
An axiomatic Basis for Computer Programming.
Communications of the ACM 12(10):576-580,583, October, 1969.

[7] C.A.R.Hoare.
Proof of Correctness of Data Representations.
Acta Informatica 1:271-281, 1972.

[8] B.H.Liskov, A.Snyder, R.Atkinson, C.Schaffert.
Abstraction mechanisms in CLU.
Technical Report Computation Structures Group Memo 144-1, MIT-LCS, Jan, 1977.

[9] D.C.Luckham et al.
Stanford Pascal Verifier User Manual, Edition 1.
Technical Report, Stanford University, April, 1979.

[10] P.A.Subrahmanyam.
Towards Automatic Program Synthesis: Obtaining Implementations from Formal Specifications.
Technical Report, State University of New York at Stony Brook, October, 1977.

[11] P.A.Subrahmanyam.
Perspectives on the use of Abstract Data Types in Programming Methodology.
November 1978, Unpublished Memo, Dept. of Computer Science, SUNY at Stony Brook.

[12] P.A.Subrahmanyam.
Towards a Theory of Program Synthesis: Automating Implementations of Abstract Data Types.
PhD thesis, Department of Computer Science, State University of New York at Stony Brook, August, 1979.

[13] J.Thatcher,E.Wagner,J.Wright.
Data Type Specifications: Parameterization and the Power of Specification Techniques.
In Proceedings, Tenth SIGACT Symp. , pages 119-132. ACM,SIGACT, April 1978, 1978.

[14] W.A.Wulf, R.L.London, M.Shaw.
Abstraction and Verification in ALPHARD.
Technical Report, CMU, ISI, August, 1976.

[15] W.A.Wulf, R.L.London, M.Shaw.
An Introduction to the construction and verification of Alphard Programs.
IEEE Transactions on Software Engineering SE-2(4):253-265, December, 1976.