Implementation of Data Types by Algebraic Methods*

EDWARD K. BLUM AND FRANCESCO PARISI-PRESICCE

Department of Mathematics, University of Southern California,
Los Angeles, California 90089

Received September 30, 1982; revised January 28, 1983

1. INTRODUCTION

The specification of algebraic and relational structures by means of axioms has a long history in mathematics and logic. The recognition that data types in programming languages can be viewed as algebras or, more generally, as relational structures is of more recent origin. Guttag and Horning [1] and Zilles [3] were among the first to make use of this view and to develop axiomatic specification techniques for what have come to be called abstract data types (ADTs). Goguen, Thatcher, Wagner, and Wright [4, 5] have developed the algebraic method of treating data types to a high level of sophistication. The trend toward greater abstraction in the specification of data types, as provided by axioms for example, is a natural concomitant to the development of the top–down approach to the design and implementation of data types and complex computer programs which perform computations on data types. Although there are still unresolved questions about algebraic specification of ADTs [7, 8], the use of algebraic methods in the implementation of ADTs can be contemplated and, indeed, has received the attention of various researchers, one of the early papers being that of Guttag, Horowitz, and Musser [2].

If a data type is to be used in actual computation, it must be implemented (eventually) at an executable level, for example, as a program package consisting of (1) machine representations of the elements in the data type and (2) procedures which define algorithms for the operations/reations of the type. One approach to producing such an executable implementation is to proceed “top–down” by “stepwise-refinement” from a very abstract specification of the data type through various implementations at decreasingly abstract levels until an executable implementation is attained. In proceeding from one level to the next, it is important to have some way to verify the correctness of each succeeding implementation against the previous one. A good methodology of stepwise-refinement should be based on a precise formulation of the notions of “implementation” and “correctness.” Various formulations based on algebraic techniques have been suggested. For example, Guttag et al. [2] use ADTs

* This research was supported in part by the National Science Foundation under Grant MCS 82 03666.
both as the implemented type $B$ and as the implementing type $A$ each being specified by a form of equational axioms which includes an If–Then–Else operation. In [2], an implementation is given as a set of axioms involving the operations of $A$ and $B$ as well as an operation $\tau$ which maps (a subset of) $A$ onto $B$. Ehrig et al. [13, 14, 21] have developed this idea further and given it a precise algebraic setting. A somewhat different approach was taken by Goguen et al. [5] using algebraic derived operations. In [13], it is observed that the algebraic derived operations of [5] are not sufficient to express the implementation of symbol-tables by stacks of mappings given in [2], since the equations specify recursive derived operations. This led Ehrig et al. [13] to replace derived operations by functors between categories of algebras, but in so doing, they raise the question of whether arbitrary functors are too general. Ehrich [11, 12] takes a similar category-theoretic approach.

In [21, 22], Ehrig et al. extend their approach to parametrized types. Hupbach [3] and Ganzinger [20] also treat implementation of parametrized types, using signature morphisms to establish a correspondence between the types in $A$ and $B$, but omitting an explicit definition of the mapping $\tau: A \to B$. Implementation of parametrized specifications is also considered by Sannella and Wirsing [44] who present a notion of implementation of one theory by another one, similar to the one in Broy et al. [45]. Although in [20–22] there are definitions of “programs” which define implementations, these are essentially equation systems like those in [2].

In this paper, we take a somewhat different approach to programs and to the basic concept of implementation, emphasizing an explicit construction of the mapping $\tau$. Furthermore, we formulate the implementation concept for arbitrary algebras $A$ and $B$, independent of their presentations, whether by abstract equational axioms or by concrete program packages in some actual language like Ada, for example.

Suppose that we are given (somehow) a many-sorted algebra $B$ of signature $\Omega$ (see Section 2) and we wish to implement it by another many-sorted algebra $A$ of signature $\Sigma$. In our view, the key condition in any concept of implementation is that it must be possible to define, by using the operations within $A$, another $\Omega$-algebra $F(A)$ which is homomorphic to $B$. This condition has three subconditions:

(i) there exists a subfamily of sets $F(A) \subseteq A$ and a surjective mapping $\tau: F(A) \to B$ which is such that it maps a subset of a sort of $A$ onto each sort of $B$ (thereby inducing a correspondence between sorts);

(ii) for each operator $\omega \in \Omega$ there must be an operation $f_\omega$ on $F(A)$ having the rank and sort determined by the correspondence in (i);

(iii) $\tau$ must be a $\Omega$-epimorphism, that is, for all $\omega \in \Omega$, $\tau f_\omega(a_1, \ldots, a_n) = \omega_B(\tau a_1, \ldots, \tau a_n)$, where $\omega_B$ is the operation on $B$ corresponding to $\omega$.

We stress that $\tau$ need not be injective. Also note that relations are treated as operations of Boolean sort, so that (iii) is a somewhat stricter kind of homomorphism than is usually treated in model theory. (More about this later.) This is similar to the concept used by Blum and Lynch in [6], but there the simulators $f_\omega$ are defined either by flowcharts or recursive schemes over $A$. Here, we shall use Godel–Herbrand–
Kleene (GHK) schemes [15]. In [7], Blum and Estes use a more general algebraic formulation of $f_\omega$. In our opinion, both the latter formulation and others based on functors between categories, although of theoretical interest, appear to be too general and too far removed from practical implementations. However, [6, 7] focus on the mapping $\tau$, whereas [2, 5, 13–23] do not, although it is implicitly included in some equations of a “specification.” In [2], $\tau$ is not required to be a homomorphism, which has the consequence that the authors must go through a rather long proof of “correctness” of each implementation, by which they mean that the algebra $F(A)$ satisfies the axioms which specify $B$. This is also the meaning of correctness in [5, 11, 12, 13]. We do not require quite as much in our concept of correctness, since $\tau$ need only be a homomorphism. We shall illustrate the consequences of this below.

It is our position that an explicit definition of $\tau$ is as much a part of an implementation as is the definition of the simulators $f_\omega$. Furthermore, it is natural to impose the homomorphism condition (iii), since this ensures that any value $\omega_B(b_1, ..., b_n)$ in $B$ can be computed by applying a prescribed sequence of operations of $A$ to a set of representing elements $a_i = \tau_R^{-1}(b_i), 1 \leq i \leq n$, in $A$. Here, $\tau_R^{-1}$ is any right inverse of the surjective mapping $\tau$. The prescribed sequence of operations is specified in the definition of the operation $f_\omega$ which “simulates” $\omega_B$. In our view, this role of $\tau$ is the essence of the notion of implementation and it may sometimes be obscured by the machinery of the algebraic equational presentation of implementations. We summarize this role in Fig. 1.

$$
\begin{array}{c}
B \xrightarrow{\tau_R^{-1}} F(A) \subseteq A \\
\downarrow \omega_B \downarrow \\
B \xleftarrow{\tau} F(A)
\end{array}
$$

Figure 1

We do not demand that $\tau$ be an isomorphism because practical implementations frequently allow several elements in $A$ to represent the same element in $B$. For most applications, it suffices that $\tau$ be an epimorphism, for then any equational identity on $B$ is a “congruential identity” on $F(A)$. An equational identity is defined by an equation $s(x) = t(x)$, where $s(x)$ and $t(x)$ are terms (or finite trees) over $\Omega \cup X$, $X$ being a set of variables. If $x \in X^n$ denotes the $n$ variables that occur in the equation, then $s(b) = t(b)$ holds for all $b \in B^n$. (See also Section 2.) Since $\tau$ is a homomorphism, $s(\tau(a)) = \tau(t(a))$ holds for all $a, a' \in F(A)^n$ such that $\tau(a) = \tau(a')$. This is what we mean by congruential identity. For example, if $\tau$ maps sequences of integers $(A)$ to the set of integers in the sequence, then concatenation can be the operation in $F(A)$ representing union in $B$; $x \cup x = x$ is true in $B$ where $x \parallel x = x$ is false in $F(A)$. (If $s$ and $t$ are Boolean-valued, then $s(a) = t(a)$, since we require $\tau(\text{True}) = \text{True}$ and $\tau(\text{False}) = \text{False}$. We shall now state this a bit more precisely in terms of “evaluation” homomorphisms.
In computing the values $s(a)$ and $t(a')$, the trees $s(x)$ and $t(x)$ are mapped onto trees over $\Sigma \cup X$ by (loosely speaking) replacing the $B$ operation nodes by (the trees of) their respective simulators in $F(A)$. Thus, let $T_{\Sigma \cup X}$ and $T_{Q \cup X}$ be the initial algebras of finite trees over $\Sigma \cup X$ and $Q \cup X$, respectively, as in [4]. Let $V_A$ and $V_B$ be the usual evaluation homomorphisms determined by assigning values $a_i \in A$ and $b_i \in B$, respectively, to the variables, where $a_i = \tau^{-1}(b_i)$. When the simulator can be represented by finite trees, we can depict the equational equivalence property of implementations with the aid of Fig. 2.

$$
\begin{aligned}
T_{Q \cup X} & \xrightarrow{\theta} T_{\Sigma \cup X} \\
\downarrow V_B & \quad \downarrow V_A \\
B & \leftarrow F(A)
\end{aligned}
$$

Figure 2

Here, $\theta$ is the mapping of trees determined by the simulators. (As we shall see, $\theta$ is also an $\Omega$-homomorphism.) The congruential identity property of homomorphisms cited above ensures that any two terms in $T_{Q \cup X}$ which define the same function on $B$ are implemented by “equivalent terms” in $T_{\Sigma \cup X}$ which yield values in $F(A)$ which are mapped onto the same value in $B$. In this sense, an implementation, as we define it, is correct. As we shall show, the “correctness” proofs of [2] are replaced by proofs that $\tau$ is a well-defined mapping.

The situation becomes more complicated when the simulators cannot be represented by finite terms, but must be obtained by some recursive computation, defined, say, by recursive schemes. This leads us to a consideration of infinite trees and continuous algebras as, for example, in [4, 19]. (In the preceding diagram, $T$ must be replaced by $CT$, the algebra of infinite trees.) If a minimal fixed-point semantics is used for recursive schemes, we must require further that $\tau$ be a continuous homomorphism. If a unique fixed-point semantics is adopted, the continuity of $\tau$ is not required for correctness. Note that we must consider $CT$ even when the simulators are given by finite trees because we usually wish to allow recursive computations on $B$.

An implementation given by simulators and a homomorphism $\tau$ as in (i)–(iii) is also correct in the wider sense that any first-order property of $B$ holds in $F(A)$ up to $\tau$-equivalence and again this is sufficient for most purposes; e.g., a verification proof of the correctness of a program over $B$ with respect to pre- and post-conditions will apply to the implementation of the program over $A$.

In the sections which follow, we shall make all these notions precise and prove our assertions about them. We also illustrate them with two (by now) well-known examples of implementations given in [2], stacks and symbol-tables.

As the reader may have surmised, our approach to implementation of data types can be applied to programs and compilers as well. Furthermore, it suggests some
important problems that have not yet been investigated, such as, for example, existence and uniqueness of implementations. It also may suggest a method of constructing implementations from axiomatic specifications.

2. An Example of an Implementation: The Polynomial Case

As stated in the Introduction, we regard data types as algebras having several sorts of elements, including the Boolean sort \{True, False\}, and possibly having Boolean-valued operations (i.e., relations). Some of these algebras may be abstract in that their operations are not given explicitly, but rather are specified by equational axioms, including the If–Then–Else operator as in [2] and [4]. To specify an algebra, we start with a signature \( \Sigma \) having a set of sorts \( S \) and a set of operators. We shall also use \( S \) to denote the operators. As is well known [4], each operator \( \sigma \in \Sigma \) has a prescribed rank \( s_1 \cdots s_n \in S^* \) and a sort \( s \in S \). If \( n = 0 \), the rank is empty and \( \sigma \) is a 0-ary operator. A \( \Sigma \)-algebra \( A \) consists of a family of nonempty sets \( (A_s) \) indexed by \( S \) and a family of operations indexed by \( \Sigma \). We find it convenient to let \( \sigma_A \) denote the operation indexed by \( \sigma \). If \( \sigma \) has rank \( s_1 \cdots s_n \) and sort \( s \), then \( \sigma_A \) is a function on \( A_{s_1} \times \cdots \times A_{s_n} \) to \( A_s \). The carrier of \( A \) is the family \( (A_s) \). We shall denote both \( (A_s) \) and \( \sigma_A \) by \( A \). We shall sometimes find it convenient to denote an algebra by \( (A; F) \), where \( A \) is the carrier and \( F \) the set of operations.

As an illustration of a signature, we take the example of stacks essentially as given in [2]. There are three sorts: Stack, Boolean, and Element and a signature, STK, consisting of the following operators (displayed as operations to show their ranks and sorts):

<table>
<thead>
<tr>
<th>Operator</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push: Stack \times \text{Element} \to \text{Stack},</td>
<td></td>
</tr>
<tr>
<td>Pop: Stack \to \text{Stack},</td>
<td></td>
</tr>
<tr>
<td>Top: Stack \to \text{Element},</td>
<td></td>
</tr>
<tr>
<td>Isnew: Stack \to \text{Boolean},</td>
<td></td>
</tr>
<tr>
<td>True: \to \text{Boolean}, False: \to \text{Boolean},</td>
<td></td>
</tr>
<tr>
<td>Newstack: \to \text{Stack},</td>
<td></td>
</tr>
<tr>
<td>Undefined: \to \text{Element},</td>
<td></td>
</tr>
<tr>
<td>( e_i ): \to \text{Element}, for ( i \in I ).</td>
<td></td>
</tr>
</tbody>
</table>

Thus, the sort Boolean consists of the two "constants," True and False. Since we shall not be concerned with the question of parameter types, the specification of the algebra of Element is not relevant. For our purposes, it suffices to take each member of Element to be a 0-ary operator \( e_i \), \( i \in I \), where \( I \) is some suitable indexing set.

There are numerous implementations of stacks in real programming systems. Each consists of some representation of a stack by a data structure (e.g., an array with pointer) and procedures which perform the operations Push, Pop, etc. Each such implementation is a STK-algebra. Abstracting from these instances of stacks, one can deduce some common properties which are expressible in the form of equational
IMPLEMENTATION OF DATA TYPES

axioms. These axioms can then be used to characterize the class of all stack implementations. The axioms given in [2] are essentially as follows:

(S1) Pop(Newstack) = Newstack,
(S2) Pop(Push(s, e)) = s,
(S3) Top(Newstack) = Undefined,
(S4) Top(Push(s, e)) = e,
(S5) Isnew(Newstack) = True,
(S6) Isnew(Push(s, e)) = False.

Here, s is a variable of sort Stack and e is a variable of sort Element. We shall refer to the set of equations (S1)–(S6) as E(STK).

For any signature \( \Sigma \) there is an algebra \( T_\Sigma \) consisting of all well-formed terms built up from the 0-ary operators by applying operations of appropriate rank to terms, application being concatenation of the operator and the terms. A typical element of \( T_\Sigma \) will be denoted by \( \sigma_1, \ldots, t_n \), where the \( t_i \) are terms of the sorts required by the rank of \( \sigma \). The algebra \( T_\Sigma \) is called the initial term algebra over \( \Sigma \). For the signature STK it consists of terms like Pop(Newstack), Push(Pop(Newstack), e_i), where \( e_i \) is a 0-ary operator in Element.

Assumption: For the most part, we shall assume that \( \Sigma \) has at least one 0-ary operator, so that \( T_\Sigma \neq \emptyset \). Furthermore, we shall usually assume that there are terms of every sort in \( T_\Sigma \). (We call such \( \Sigma \) complete signatures.)

To define functions on a \( \Sigma \)-algebra, we introduce a set \( X \) of typed variables, each of a specified sort in \( S \), and form terms over the signature \( \Sigma \cup X \). (The variables can be regarded as new 0-ary operators of specified sorts.) The elements of the free term algebra \( T_{\Sigma \cup X} \) are called polynomial terms over \( \Sigma \cup X \) [16]. Note that \( T_\Sigma \subset T_{\Sigma \cup X} \). For example, take the signature STK with \( X = \{ s, e \} \). The left and right sides of the equations in \( E(STK) \) are polynomial terms. Now, let \( A \) be a \( \Sigma \)-algebra. A mapping \( f: X \to A \) is called an assignment of values to the variables if \( x \) and \( f(x) \) have matching sorts, where \( f(x) \) is the value assigned to \( x \). There is a natural extension of \( f \) to a \( \Sigma \)-homomorphism, \( V_{Af} \), of \( T_{\Sigma \cup X} \) into \( A \) defined by

(1) \( V_{Af}(\sigma) = \sigma_A \) for every 0-ary operator \( \sigma \);
(2) \( V_{Af}(x) = f(x) \) for \( x \in X \);
(3) \( V_{Af}(\sigma_1, \ldots, t_n) = \sigma_A(V_{Af}(t_1), \ldots, V_{Af}(t_n)) \).

We call \( V_{Af} \) an evaluation. Note that if \( X \) is empty, we omit (2) and obtain a \( \Sigma \)-homomorphism of \( T_\Sigma \) into \( A \) which we denote simply by \( V_A \). (If \( X \) is not empty, we can also regard the elements \( f(x) \) as new 0-ary operators in \( A \) and then \( V_{Af} \) is a \( \Sigma \cup X \)-morphism.) If \( A \) is generated by its 0-ary operators, then \( V_{Af} \) is surjective. This is the case for many data types, such as stacks. Note that when \( A = T_\Sigma \) the mapping \( V_A \) is just the identity mapping.

By a slight modification of the method in [16] to take into account the many-sorted case, we associate a polynomial operation on \( A \) with each polynomial term \( t \in T_{\Sigma \cup X} \). We do this by partitioning \( X \) into sets of typed variables \( X_\sigma \), one for each
sort $s$, and then indexing each $X_i$ by the positive integers. Thus, $X_s = \{x_{s1}, x_{s2}, \ldots, \}$. Now, let $s_1, \ldots, s_k$ be the variable sorts which occur in $t$. Let $n_i, 1 \leq i \leq k$, be the highest index of the variables of sort $s_i$ which occur in $t$. The polynomial operation on $A$ (also called a derived operation on $A$ [17]) denoted by $t$ is the function $F_{tA} : A_{s_1} \times \cdots \times A_{s_k} \rightarrow A$ defined by

\[ F_{tA}(a_{11}, \ldots, a_{1n_1}, a_{k1}, \ldots, a_{kn_k}) = V_{tA}(t), \]

where $f$ is an assignment of values with $f(x_{sij}) = a_{ij}, 1 \leq i \leq k, 1 \leq j \leq n_i$. In the special case where $A = T_\Sigma$, we shall denote the derived operation by $F_{t\Sigma}$. The rank of $F_{tA}$ is $s_1 \cdots s_1 s_2 \cdots s_2 \cdots s_k \cdots s_k$, where $s_i$ occurs $n_i$ times. Its sort is the sort of $t$.

Remark 0. Let $F = \{t_1, \ldots, t_m\}$ be a set of distinct trees in $T_\Sigma$. For each $\Sigma$-algebra $A$, there is a corresponding set of derived operations, $F_A = \{F_{t1A}, \ldots, F_{tmA}\}$. Of course, the $F_{tA}$ need not be distinct. When $A = T_\Sigma$, the set $F_{tA} = \{F_{t1\Sigma}, \ldots, F_{tm\Sigma}\}$ does indeed consist of distinct operations, since $T_\Sigma$ is free. $F$ determines a derived signature $\Sigma'$ consisting of $m$ operators, $\sigma'_1, \ldots, \sigma'_m$ say, such that the rank and sort of $\sigma'_i$ is that of $F_{tA}$, $1 \leq i \leq m$. Let $A'$ be the family of sets $A$, such that sort $s$ occurs in the rank of some $\sigma'_i$ or is the sort of some $\sigma'_i$. Call this set of sorts $S'$. The algebra $A' = (A'; F_A)$ is a $\Sigma'$-algebra with nonempty carrier $A' \subseteq A$. For example, when $A = T_\Sigma$ we obtain the derived $\Sigma'$-algebra $T'_\Sigma = (T'_\Sigma; F_{T_\Sigma})$, where $T'_\Sigma \subseteq T_\Sigma$. Since $\Sigma$ is complete, $T'_\Sigma$ is non-empty. The derived $F$-algebra of $A$, is defined to be the intersection of all $\Sigma'$-subalgebras of $A'$ and is denoted by $F(A)$. Thus, $F(A) \subseteq A'$ as $\Sigma'$-subalgebras. The special case $F(T_\Sigma)$ is called the freely derived $F$-algebra. Note that $F(T_\Sigma)$ need not be a free $\Sigma'$-algebra.

Remark 1. If $\Sigma'$ is complete, then the algebra generated by the $0$-ary operators of $F_A$ (and the operations in $F_A$) must include elements of every sort in $S'$. Hence, this algebra must be $F(A)$. In this case, $F(A)$ is the image in $A'$ of the term algebra $T_\Sigma$ under the evaluation $\Sigma'$-homomorphism $V_{A'}$. In general, $V_{A'}$ is not injective. This may be the case even for $A = T_\Sigma$. Thus, $T'_\Sigma$ need not be isomorphic to $F(T_\Sigma)$. (It is easy to construct examples of noninjective and injective $V_{A'}$ with $A = T_\Sigma$.)

Remark 2. It is easy to see that the evaluation $V_A : T_\Sigma \rightarrow A$, given as a $\Sigma$-homomorphism, when restricted to $T'_\Sigma$ is also a $\Sigma'$-homomorphism which maps $T'_\Sigma$ into $A'$. The restriction of $V_A$ to $F(T_\Sigma) \subseteq T_\Sigma$ is likewise a $\Sigma'$-homomorphism. Let $V' : T'_\Sigma \rightarrow F(T_\Sigma)$ be the $\Sigma'$-evaluation for $F(T_\Sigma)$. Again, assuming $\Sigma'$ is complete, this evaluation is an epimorphism. Considering $V_A$ as the restricted $\Sigma'$-homomorphism, we obtain

\[ V_A \cdot V'(T'_\Sigma) = V_A(F(T_\Sigma)) \subseteq A'. \]

Hence, the composite mapping $V_A \cdot V'$ is a $\Sigma'$-homomorphism of $T_\Sigma$ into $A'$. Since this must be unique, $V_A \cdot V' = V_{A'}$, and, therefore, its image is $F(A)$, that is,

\[ V_A(F(T_\Sigma)) = F(A). \]

An equation over $\Sigma \cup X$ is a pair $(t, s)$ of polynomial terms. We shall write it as
An equation is valid in a $\Sigma$-algebra $A$, or is an equational identity on $A$, if $V_A(t) = V_A(s)$ for every evaluation $V_A$. We also call $A$ a model of the equation in that case. A class of $\Sigma$-algebras consisting of all $\Sigma$-algebras which are models of a set $E$ of equations is called a variety [16]. Thus, for $E(\text{STK})$ we obtain the variety of stack algebras. As we know, $E$ generates a congruence, $\equiv_E$, on $T_{\Sigma}$ and the quotient algebra $T_{\Sigma}/\equiv_E$ is an initial algebra [4] in the variety. When considering computations on a data type specified by equational axioms such as $E(\text{STK})$, one usually chooses an algebra in this variety. Frequently, the algebra is taken to be an initial algebra. For example, consider the initial stack algebra, $B = T_{\text{STK}}/\equiv_E(\text{STK})$. To obtain stacks in $B$, we form congruence classes of terms in $T_{\text{STK}}$. The evaluation $V_B: T_{\text{STK}} \rightarrow B$ maps a term onto its congruence class. For convenience, we can choose a representative term from each congruence class to denote the class. Thus, terms such as $\text{Push(\text{Push} \ldots (\text{Push(\text{Newstack}, e,) \ldots e,) \ldots e,) \ldots e,) \ldots e,...}$ would serve to represent stacks in the initial algebra of stacks, since axiom (S2) allows us to eliminate all Pop's from any term.

Now, to illustrate our concept of implementation of an $\Omega$-algebra $B$ by a $\Sigma$-algebra $A$, as sketched in the Introduction, we shall use the example in [2] in which the implementee $B$ is a stack algebra satisfying axioms (S1)-(S6). In fact, we shall take $B = T_{\text{STK}}/\equiv_E(\text{STK})$. As $A$, we take an algebra of pointer-arrays given by a signature and axioms similar to those in [2]. We introduce the sorts Array, Element, Integer, and Boolean. As in [2], Array is an abstraction of one-dimensional arrays. Element is the same sort as in STK, so that its members are again denoted by $e_i$, $i \in I$. (As before, Element is a parameter sort, to be replaced by an actual sort. Since we do not treat parameter passing within our implementation definition, we adopt this method of referring to Element members.) Boolean is as in STK. We shall regard the sort Integer as “pre-defined,” that is, it has the standard 0-ary operation 0 and the operations successor (written $n + 1$), predecessor (written $n - 1$) and integer equality (EQ) with the usual axioms for these operations, or some standard implementation of these operations. Rather than use array-integer pairs informally as in [2], we introduce another sort, Ptrarr, to play the role of arrays with integer pointers. We shall use $a, j$ and $aq$ as variables of sort Array, Integer, and Ptrarr, respectively. We also use $n$ and $m$ as Integer variables and $e$ as a variable of sort Element. We define a signature, PTAR, having the following operators:

- **Newarray**: $\rightarrow$ Array,
- **Assign**: $\text{Array} \times \text{Integer} \times \text{Element} \rightarrow \text{Array},$
- **Access**: $\text{Array} \times \text{Integer} \rightarrow \text{Element},$
- **Pr1**: $\text{Ptrarr} \rightarrow \text{Array}$, (projection),
- **Pr2**: $\text{Ptrarr} \rightarrow \text{Integer}$, (projection),
- $\langle \rangle$: $\text{Array} \times \text{Integer} \rightarrow \text{Ptrarr}$, (pairing).

The axioms, $E(\text{PTAR})$, are as follows:

(P1) $\text{Access(\text{Newarray}, n) = Undefined},$

(P2.1) $\text{Access(\text{Assign}(a, n, e), n) = e},$
(P2.2) \[ \text{Access}(\text{Assign}(a, n, e), m) = \text{Access}(a, m) \] \text{if } EQ(n, m) = \text{False},

(P3) \[ \text{Pr1}(\langle a, j \rangle) = a. \]

(P4) \[ \text{Pr2}(\langle a, j \rangle) = j. \]

(P5) \[ \langle \text{Pr1}a_j, \text{Pr2}a_j \rangle = a_j. \]

Pr1, Pr2, and \( \langle \, \rangle \) are sometimes treated as "hidden" operators. For our purposes, it is convenient to make them explicit parts of PTAR. In [2], (P2.1) and (P2.2) are combined into one axiom using an If–Then–Else operator. This does not affect our main results. As the implementor algebra, we take an initial algebra, \( A = \mathcal{T}_{\text{PTAR}}/\equiv_{\text{PTAR}} \). Condition (ii) (see Introduction) requires that any specification of an implementation of \( B \) by \( A \) must define a set \( F_A \) of simulators, that is, for each operation, \( \omega_B \), in STK, the specification must define a simulator function, \( f_{\omega_B} \), in \( A \) which will behave as shown in Fig. 1 and satisfy conditions (ii) and (iii) for some mapping \( r \). In this example, it is possible to define the simulators to be derived operations on \( A \). Each simulator will be denoted by a "derived operator" consisting of the capitalized name of the simulated operator in \( B \) preceded by the letter \( F \) as follows:

\begin{align*}
\text{FNEWSTACK:} & \quad \to \text{Ptrarr}, \\
\text{FPUSH:} & \quad \text{Ptrarr} \times \text{Element} \to \text{Ptrarr}, \\
\text{FPOP:} & \quad \text{Ptrarr} \to \text{Ptrarr}, \\
\text{FTOP:} & \quad \text{Ptrarr} \to \text{Element}, \\
\text{FISNEW:} & \quad \text{Ptrarr} \to \text{Boolean}.
\end{align*}

The rank and sorts of these simulators determine (or are determined by) a correspondence between the sorts of PTAR and STK. Thus, \( \text{Ptrarr} \) corresponds to \( \text{Stack} \). The simulators are defined by a "program scheme" (over PTAR and the derived operators) consisting of the following equations:

\begin{align*}
\text{FNEWSTACK} &= \langle \text{Newarray}, 0 \rangle, \\
\text{FPUSH}(a_j, e) &= \langle \text{Assign(Pr1}(a_j), \text{Pr2}(a_j) + 1, e), \text{Pr2}(a_j) + 1 \rangle, \\
\text{FPOP}(a_j) &= \langle \text{Pr1}(a_j), \text{Pr2}(a_j) - 1 \rangle \text{ if } \text{Pr2}(a_j) \neq 0, \\
\text{FPOP}(a_j) &= a_j \text{ if } \text{Pr2}(a_j) = 0, \\
\text{FPOP}(a_j) &= \text{Access}(\text{Pr1}(a_j), \text{Pr2}(a_j)) \text{ if } \text{Pr2}(a_j) \neq 0, \\
\text{FTOP}(a_j) &= \text{Undefined} \text{ if } \text{Pr2}(a_j) = 0, \\
\text{FISNEW}(\langle a, 0 \rangle) &= \text{True}, \\
\text{FISNEW}(a_j) &= \text{False} \text{ if } \text{Pr2}(a_j) \neq 0.
\end{align*}

We shall refer to the above scheme as SIM(STK).

Strictly speaking, we should introduce the If–Then–Else operator in SIM(STK), following [2], since this would yield a single equation with a polynomial term on the right side for each derived operator. Assume this to be done. Let \( F \) be the set of those
polynomial terms. Regarding the derived operators as function variables, we can "solve" \text{SIM(STK)} on any PTAR algebra \emph{A} and obtain a derived operation on \emph{A} for each derived operator. This derived operation is defined by (4) and the corresponding polynomial term in \text{SIM(STK)}. Referring back to Remarks 1 and 2, let us put \( \Sigma = \text{PTAR} \) and \( \Sigma' = \text{PTAR}' = \{ \text{FNEWSTACK}, \text{FPUSH}, \text{FPOP}, \text{FTOP}, \text{FISNEW} \} \). The set of sorts for \text{PTAR}' is \{ \text{Ptrarr}, \text{Element}, \text{Boolean} \}. The set of derived operations on \emph{A} which yield \( F(\text{A}) \) is \( F.A = \{ \text{FNEWSTACK}_A, \text{FPUSH}_A, \text{FPOP}_A, \text{FTOP}_A, \text{FISNEW}_A \} \), where, for example, \( \text{FPUSH}_A(aj, e) = \langle \text{Assign}(\text{Pr}_1(aj), \text{Pr}_2(aj) + 1, e), \text{Pr}_2(aj) + 1 \rangle_A \). Similarly, \( F(T_{\text{PTAR}}) \) is the freely derived \( F \)-algebra in \( T_{\text{PTAR}} \). Now, there is an obvious signature isomorphism between \text{STK} and \text{PTAR}' which allows us to consider any \text{PTAR}'-algebra as a \text{STK}-algebra, and conversely. In particular, \( T_{\text{STK}} \) and \( T_{\text{PTAR}} \) are isomorphic. Therefore, there is a unique \text{STK}-evaluation homomorphism \( \theta: T_{\text{PTAR}} \rightarrow F(T_{\text{PTAR}}) \). For example, for any Stack term \( s \) and Element term \( e \),

\[
\theta \text{Push}(s, e) = \text{FPUSH}_{F(T_{\text{PTAR}})}(\theta s, \theta e) = \langle \text{Assign}(\text{Pr}_1(\theta s), \text{Pr}_2(\theta s) + 1, e), \text{Pr}_2(\theta s) + 1 \rangle
\]

illustrating how \text{STK}-terms map onto \text{PTAR}-terms in \( F(T_{\text{PTAR}}) \). This motivates our first condition for a (polynomial) implementation. If we put \( \Omega = \text{STK} \), we may state it as follows.

(I) Let \( \Omega \) and \( \Sigma \) be complete signatures. Let \( F \) be a scheme, consisting of equations involving terms over \( \Sigma \), variables and derived operator symbols, which defines corresponding derived operations on \( T_{\Sigma} \) such that the freely derived algebra \( F(T_{\Sigma}) \) is an \( \Omega \)-algebra and, therefore, the evaluation \( \theta: T_{\Omega} \rightarrow F(T_{\Sigma}) \) is an \( \Omega \)-epimorphism.

(In Section 3, we shall give a specific syntax and semantics of \( F \).) If \( B \) is an \( \Omega \)-algebra which is to be implemented by a \( \Sigma \)-algebra \emph{A}, then conditions (i)–(iii) of the Introduction must hold. Thus, we would like the scheme \( F \) to also define operations on \emph{A} which give rise to an \( \Omega \)-algebra \( F(\text{A}) \). In this example of the polynomial case, \text{SIM(STK)} satisfies this requirement. Furthermore, Remark 2 applies. However, for general schemes we must state this as a second condition.

(II) Let \( \emph{A} \) be a \( \Sigma \)-algebra. The scheme \( F \) defines derived operations on \emph{A} such that the derived algebra \( F(\text{A}) \) is an \( \Omega \)-algebra and \( V_{\emph{A}}(F(T_{\Sigma})) = F(\text{A}) \), where \( V_{\emph{A}} \) is the evaluation \( \Sigma \)-homomorphism (also an \( \Omega \)-homomorphism).

In our example, if \( aj \) is a \text{Ptrarr} term and \( e \) is an Element term, we have

\[
V_{\emph{A}} \text{FPUSH}_{F(T_{\text{PTAR}})}(aj, e) = V_{\emph{A}} \langle \text{Assign}(\text{Pr}_1(aj), \text{Pr}_2(aj) + 1, e), \text{Pr}_2(aj) + 1 \rangle_A = \langle \text{Assign}(\text{Pr}_1(V_{\emph{A}} aj), \text{Pr}_2(V_{\emph{A}} aj) + 1, e), \text{Pr}_2(V_{\emph{A}} aj) + 1 \rangle_A = \text{FPUSH}_{V_{\emph{A}}}(V_{\emph{A}} aj, e)
\]

as required in (II), and similarly for the other operations.
Since $V_B: T_\Omega \to B$ is an $\Omega$-epimorphism when $B$ is generated by its 0-ary operators, as it is for the initial STK-algebra, it seems reasonable to obtain the homomorphism, $\tau$, in (i) and (iii) by requiring the diagram in Fig. 3 to commute.

![Diagram](image)

We can achieve this by defining $\tau$ as follows.

(III) Let $B$ be an $\Omega$-algebra such that $V_B$ is onto. For $a \in F(A)$, let $t \in F(T_{\Sigma})$ be such that $V_A(t) = a$. Further, let $s \in T_\Omega$ be such that $\theta(s) = t$. Define $\tau(a) = V_B(s)$.

In general, this makes $\tau$ a relation on $F(A)$ to $B$. To make it a function, we require one more condition.

(IV) For any $s_1$ and $s_2$ in $T_\Omega$,

$$V_A \cdot \theta(s_1) = V_A \cdot \theta(s_2) \to V_B(s_1) = V_B(s_2)$$

If (I)–(IV) hold, then $\tau$ is an $\Omega$-epimorphism. To verify this, simply observe that in the commutative diagram

![Diagram](image)

both $V_B$ and $V_A \cdot \theta$ are $\Omega$-epimorphisms. Hence, for any $\omega \in \Omega$ and $a_1, \ldots, a_n \in F(A)$, where $a_i = V_A \cdot \theta(t_i)$ and $\tau(a_i) = b_i$, we have $V_B(t_i) = b_i$ and $\tau(f_{\omega} a_1 \cdots a_n) = \tau(V_A \cdot \theta(\omega t_1 \cdots t_n)) = V_B(\omega t_1 \cdots t_n) = \omega_B b_1 \cdots b_n$. Thus, conditions (i)–(iii) of the Introduction are satisfied.

**DEFINITION.** Let $\Omega$ and $\Sigma$ be signatures. Let $F$ be a scheme satisfying (I). We say that $F$ specifies an $\Omega$-by-$\Sigma$ simulation. Let $A$ be a $\Sigma$-algebra and $B$ and $\Omega$-algebra. If $A$, $B$, $F$ and $\tau$ satisfy (I)–(IV), we call $(F(A), \tau)$ a $B$-by-$A$ implementation. The derived operations on $A$ are called simulators.

Note that if we take $A = T_{\Sigma}$ and $B = T_\Omega$, then $F(A) = F(T_{\Sigma})$ and $V_A$, $V_B$ are identity mappings. Condition (IV) then becomes the condition that $\theta$ be an
IMPLEMENTATION OF DATA TYPES

isomorphism. In fact, this is the case of our example, but it need not be true in general. To verify that our example is a B-by-A implementation, we shall prove that (IV) holds. (See the Appendix.) This proof replaces the proof of "correctness" given by Guttag et al. in [2], since a B-by-A implementation is correct in the sense explained in the Introduction. In our example we have taken B to be the initial algebra satisfying the axioms. This has the advantage that if any algebra C is an \(\Omega\)-homomorphic image of B, say \(\phi: B \rightarrow C\), then \((F(A), \phi \cdot \tau)\) is a C-by-A implementation.

3. SCHEMES OVER ALGEBRAS

In the preceding discussion, we intentionally left open the matter of prescribing the precise syntax and semantics of the scheme \(F\) which defines the simulators. We required only that \(F\) consist of equations of the form \(t = s\), where \(t\) and \(s\) are terms built up from the operators in \(\Sigma\), variables of the requisite sorts and symbols for derived operators (which can be thought of as function variables), as in the SIM(STK) example. In [2, 11–14, 20–23], the equations in \(F\) may include operators in both \(\Omega\) and \(\Sigma\) and it is required that all \(\Omega\) terms be reducible to \(\Sigma\) terms (called "OP-completeness" in [14, 21]). This implicitly imposes restrictions on \(F\). While this approach seems attractive as a purely algebraic formulation of some generality, we propose an approach which is conceptually different and, possibly, different in technical detail. However, we are aware of certain technical similarities. We believe these will become more evident when we consider questions of existence and uniqueness of implementations—apparently not studied previously—and questions of complexity. (See [6] and [14] for some preliminary results on complexity.) We shall defer to future papers a study of these questions in order to focus more sharply on the concept of an implementation and its correctness. We now clarify the role of the scheme \(F\) in our definition. As the example in the next section will demonstrate, this demands that we consider general computations and recursive functions on algebras. This subject has a fundamental importance of its own, independent of implementation of data types, because data types in programming languages are meant to be used in computations. If computations on data types can involve recursive procedures, then the semantics of programs which reference data types must be concerned with recursive functions on algebras. A program which references data types is meant to denote a computation (operational semantics) or a function (denotational semantics) on those data types. In both kinds of semantics, it is desirable to treat the data types as algebras at various levels of abstraction, so that the meaning of a program is to be sought within a framework of computations and functions on algebras. In the previous section, we encountered the special case of polynomial functions denoted by polynomial terms ("straight-line" computations and programs). To give meaning to general programs we must consider recursive functions and recursive computations on data types. The semantics of recursive programs has been studied intensively. In the denotational approach, the semantics of recursive programs essentially becomes "fixed-point semantics." It has long been recognized, from the beginning of the
development of classical recursive function theory (on the integers), that a recursive function can be thought of as a fixed point of a transformation on a domain of functions defined by a recursive scheme. This is the content of the classical Recursion Theorem of Kleene [40] (see also [41]). The extension of this theorem to classes of algebras has been the subject of much recent research. We cannot cite all pertinent references, but among those that have some bearing on the present paper are [24–34]. We observe in passing that fixed-point theory, as a mathematical discipline, has an existence of its own, and much of it, however beautiful, may not bear directly on the subject of interest here. To put our approach to schemes in perspective, we consider briefly some of the results in the cited references. Many of these deal with the problem of defining classes of algebras for which certain functionals have fixed points. Equivalently [39], others deal with algebraic theories within which certain kinds of fixed-points exist and are unique as, for example, Elgot's iterative theories [24]. An algebraic theory, $\text{Th}$, in its concrete form, prescribes (or is) a family of functions defined on a family of sets indexed by a set $S$ of sorts. Each function in $\text{Th}$ has a rank in $S^*$ and a sort in $S^*$. Thus, it may be vector-valued consisting of a tuple of functions each of single sort. $\text{Th}$ must contain the ordinary projection functions (corresponding to variables) and be closed under tupling and composition of functions. Composition must be associative (when defined). Among the interesting concrete theories are those generated by a signature $\Sigma$ (indexed by a subset of $S^* \times S$ as in Section 2). One such theory is the “free theory,” $\text{Th}_\Sigma$, consisting of tuples of polynomial terms (finite trees) over $\Sigma \cup X$. Composition in $\text{Th}_\Sigma$ is effected by ordinary substitution (of trees for variables). $\text{Th}_\Sigma$ models the way functions are constructed in any $\Sigma$-algebra. More interesting theories are obtained from $\text{Th}_\Sigma$ by introducing a set $E$ of equational identities over $\Sigma \cup X$. To model functions in the algebras of the variety defined by $E$, we form the quotient theory $\text{Th}_\Sigma/E$, which consists of congruence classes of tuples of trees. If $A$ is any algebra in this variety, then we can think of $\text{Th}_\Sigma/E$ as the family of all (tuples of) polynomial functions (i.e., derived operations) on $A$. It is obvious that to obtain (nonpolynomial) recursive functions we must enlarge this theory, that is, if recursive programs are to be regarded as defining fixed points of certain transformations on a family of functions on $A$, then obviously the family must be large enough to admit fixed points of such transformations. Much of the research in [24–37] is concerned with constructing theories, or classes of algebras, in which transformations defined by recursive schemes of various kinds have fixed points. This usually requires introducing a topology into the theory and the algebras of the theory so that a notion of limit and continuous transformation can be defined. One way to do this is to introduce a natural partial order $\leq$ on trees (e.g., $[27,42]$), where $t_1 \leq t_2$ roughly means that $t_1$ is a subtree of $t_2$ having the same root. Then use the sup to define limits. For infinite chains $t_1 \leq t_2 \leq \cdots \leq t_n \leq \cdots$ of finite trees in $T_{\Sigma \cup X}$ the sup construction generally leads to infinite trees with nodes labeled by $\Sigma$ and leaf nodes labeled by (a finite subset of) $X$ and a special symbol $\bot$ denoting “undefined.” The set of all such finite and infinite trees is denoted by $CT_{\Sigma \cup X}$. (When $X$ is empty, we write simply $CT_\Sigma$.) It is well known that $CT_{\Sigma \cup X}$ is a (chain-) complete poset (i.e., chains have sup’s) as well
as an algebra. The corresponding free theory denoted by $CTh_\Sigma$, consists of all (finite) tuples of infinite trees and finite trees over $\Sigma \cup X \cup \{\bot\}$ with composition again defined as substitution (of trees for variable leaf nodes). Further details can be found in [27] or [39]. Thus, it can be shown that $CTh_\Sigma$ is an ordered theory (i.e., each set $T(u,v)$, of trees of rank $u \in S^*$ and sort $v \in S^*$ is a poset and composition and tupling are monotonic). Furthermore, it is an $\omega$-continuous theory (i.e., each set $T(u,v)$ is chain-complete and composition is (chain-) continuous). Similarly, $CT_{\Sigma \cup X}$ is a complete poset (assumed to have a minimum element $\bot$ denoting the totally undefined function) and its operations are continuous ($\sigma(\sup\{t_i\}) = \sup\{\sigma(t_i)\}$). By considering continuous algebras and algebraic theories which are chain-complete (or equivalently, directed-set complete) or complete in some more general topology, one can relate the operational semantics of recursive schemes on algebras to their fixed-point semantics as, for example, in [19]. Now, algebraic theories can be presented in various ways. One of the most common ways is by prescribing a signature and a set $E$ of equational axioms. Thus, one can consider the theory $Th_\Sigma \equiv E$ corresponding to a variety [39]. However, the extension of the quotient construction to continuous theories is encumbered with technical difficulties [43]. To avoid these difficulties, we shall not attempt to formulate our concept of a scheme $F$ in the framework of general continuous theories. For our purposes, it suffices to consider schemes over $\Sigma$-algebras and to give them an operational semantics. Furthermore, we shall not require the $\Sigma$-algebras to be continuous. However, if a $\Sigma$-algebra $A$ is not continuous, we shall introduce the special element $\bot$ and define an extension of $A$ which is continuous, as in [38]. Thus, $A \cup \{\bot\}$ is defined to be a "flat domain" (with $\bot \sqsubseteq a$ for all $a \in A$ and all elements of $A$ incomparable). For any nonboolean operation, $\omega: A^n \to A$, we extend $\omega$ to $A \cup \{\bot\}$ by defining $\omega(x_1, \ldots, x_n) = \bot$ if any $x_i = \bot$. Any axioms for $A$ would have to be suitably modified. (For example, $\text{POP}(\text{PUSH}(s,e)) = s$ if $e \neq \bot$ and $=\bot$, otherwise). However, we retain the definitions (see [38]). If--Then--Else (True, $q$, $r$) = $q$, If--Then--Else (False, $q$, $r$) = $r$, regardless of whether $q$ or $r$ are $\bot$. For any two functions on $A$, say $f$ and $g$, having the same signature, we define the induced ordering $f \sqsubseteq g$ if $f(x) \sqsubseteq g(x)$ for all $x$; i.e., in this case $f \sqsubseteq g$ if and only if $g$ is an extension of $f$. We call this the inclusion ordering on functions. The resulting poset of functions on $A$ is chain-complete. If $A$ is given as a continuous algebra, then the ordering on functions is the ordering induced by the given ordering in $A$. This is a complete partial order also. For certain schemes over $A$, it is then not difficult to prove that their operational and least fixed-point semantics (with respect to the induced ordering) are equivalent [19]. In the case where $A$ is an iterative $\Sigma$-algebra, not only fixpoint equations in $A$ have a unique solution, but also recursive schemes, of a restricted form, have unique fixpoints in the iterative algebraic theory generated by $A$. This theory can be obtained as a quotient of $RTh_\Sigma$, the free subtheory of $CTh_\Sigma$ consisting of all (finite) tuples of finite and "regular" trees over $\Sigma \cup X$ [36]. The homomorphism property of $r$ then requires that $B$ be "rationally closed," that is, that every fixpoint equation have a solution, not necessarily a unique one. This, in turn, imposes restrictions on the type of algebra $A$ which can implement a given $B$. If the algebra $B$ to be implemented in also iterative, then no additional conditions (such as
continuity) need be imposed on \( \tau \) since every \( \Sigma \)-homomorphism must preserve unique solutions. Again, several technical difficulties (similar to those arising in continuous theories) are present when dealing with quotients of iterative theories and algebras.

With the preceding discussion as a background, we proceed now to give our definition of a scheme over a signature \( \Sigma \), where \( \Sigma \) is assumed to contain the If–Then–Else operator.

Let \( \Sigma \) be a signature indexed by \( S^* \times S \) and let \( X_s \) be a set of typed variables for each \( s \in S \). Put \( X = (X_s) \). Let \( F = \{ f_1, \ldots, f_m \} \) be a set of “function variables,” each \( f_i \) being of a declared rank \( r_i \in S^* \) and sort \( s_i \in S \). Consider the extended signature \( \Sigma \cup X \cup F \) and form \( T_{\Sigma \cup X \cup F} \). Following [15], we generalize the Godel–Herbrand–Kleene (GHK) system [40] for defining recursive functions (on integers) to a system called \( \Sigma \)-GHK, for defining functions on any \( \Sigma \)-algebras. A \( \Sigma \)-GHK scheme is a set of equations of the form \( t = s \), where \( t, s \in T_{\Sigma \cup X \cup F} \) for some \( F \) as above. We designate such a scheme by \( F \) again.

We note that \( \Sigma \)-GHK schemes have a more general syntax than the usual recursive schemes studied, say, in [19, 27, 28, 38]. Actually, to be sure that a scheme defines a function, we shall restrict its form. (See theorem below.) However, the general syntax allows us to write a scheme as a system of equations as in [2, 11–14, 20–23] with the operators in \( \Omega \), the signature to be implemented, replaced by function variables. In fact, this is the syntax of GHK schemes over the integers given in [40, p. 264], where \( \Sigma = \{0, \' \} \).

We define an operational semantics for a \( \Sigma \)-GHK scheme \( F \) for any \( \Sigma \)-algebra \( A \) which is more general than the semantics of the GHK schemes in [40] and resembles the operational semantics in [19] in that it permits arbitrary computation sequences. We shall assume that \( A \) has been extended, if necessary, to include the special bottom element \( \bot \), as explained above. We shall also assume that the above axioms for If–Then–Else (e.g., see [2]) apply to terms over the extended signature \( \Sigma \cup F \cup X \). Under these assumptions, the operational semantics of the restricted \( \Sigma \)-GHK schemes is equivalent to that in [19]; i.e., the value of a defined function is the sup of all “intermediate results” of computations. However, rather than speaking of the Algol copy rule as is done in [19], we shall follow [40] and view a computation as a logical derivation of equations from the given equations in a scheme \( F \) using essentially equational logic. Now, the semantics of \( F \) is given relative to an arbitrary \( \Sigma \)-algebra \( A \) which provides an “interpretation” of \( F \) by associating an operation on \( A \) with each operator in \( \Sigma \). The semantics is based on two rules of derivation called substitution and replacement.

**Substitution.** Let \( e \) be an equation in \( F \) which contains one or more occurrences of a variable \( x \). Let \( t \in T_\Sigma \) represent an element, \( a = V_A(t) \), of \( A \). The equation, \( e(x \mid t) \), obtained by substituting \( t \) for every occurrence of \( x \) is derivable from \( F \). We write \( F \vdash e(x \mid t) \). (We shall also say that \( a \) is substituted for \( x \) in \( t \) in this case.)

**Replacement.** Let \( t = s \) be an equation in \( F \). If \( e \) is an equation in \( F \), then the equation \( e(t \mid s) \) obtained by replacing any occurrence of \( t \) by \( s \) is derivable from \( F \).
(Likewise, if \( e \) contains \( s \), then \( e(s \mid t) \) is derivable.) We write \( F \vdash e(t \mid s) \). We also define every equation in \( F \) to be derivable from \( F \).

A derivation from \( F \) is a sequence of equations \((e_1, \ldots, e_n)\) such that \( F \vdash e_1 \) and for \( 1 \leq i \leq n - 1 \), \( F \vdash e_i \leftarrow e_{i+1} \). We again write \( F \vdash e_n \).

Now, let \( f \) be a function variable in \( F \). Suppose that whenever \( F \vdash f(t_1, \ldots, t_n) = t_0 \) and \( F \vdash f(s_1, \ldots, s_n) = s_0 \), where the \( t_i \) and \( s_i \) are terms in \( T_\Sigma \) such that \( V_\Sigma(t_i) = V_\Sigma(s_i) \) for \( 1 \leq i \leq n \), then \( V_\Sigma(t_0) = V_\Sigma(s_0) \). In that case, we say that \( F \) defines a (partial) function, \( f_\Sigma : A^n \rightarrow A \), with \( f_\Sigma(V_\Sigma(t_1), \ldots, V_\Sigma(t_n)) = V_\Sigma(t_0) \). Such a function is called a recursive derived operation (rdo) on \( A \). When \( F \vdash f(t_1, \ldots, t_n) = t_0 \), where \( V_\Sigma(t_i) = a_i \), 0 \( \leq i \leq n \), we shall write \( F \vdash f(a_1, \ldots, a_n) = a_0 \), and say that \( f(a_1, \ldots, a_n) = a_0 \) is also derivable from \( F \). However, note that \( \vdash \) involves \( V_\Sigma \) as well as \( F \).

Derived operations are a special case of rdo's. A derived operation is defined by a polynomial term in \( T_{\Sigma \cup X} \) and can be represented by a finite tree. For a recursive operation \( f \), it is convenient to represent it by an infinite tree \( t_f \) in \( CT_{\Sigma \cup X} \) as in [19]. The tree \( t_f \) is obtained by the usual "unfolding" of the terms in \( F \). This will always be possible for the restricted from of \( F \) considered below. More precisely, \( t_f \) is the minimal fixed point of the transformation defined by such \( F \) on the free theory \( CT_\Sigma \).

We now state sufficient conditions that ensure that \( F \) defines a function on any \( \Sigma \)-algebra \( A \).

**Theorem.** Let \( A \) be a \( \Sigma \)-algebra. Let \( F \) be a \( \Sigma \text{GHK} \) scheme satisfying the following three conditions:

(i) every equation in \( F \) has the form \( f_i(t_1, \ldots, t_k) = t \), where \( f_i \) is a function variable, \( t \in T_{\Sigma \cup X \cup F} \) and the \( t_i \) are terms in \( T_{\Sigma \cup X} \) which define injective polynomial functions, \( t_{\Sigma X} \), on \( A \);

(ii) for any \( k \)-ary function variable \( f_i \) and \( a \in A^k \), there is at most one equation \( e \) in \( F \) such that \( a \) occurs on the right side of \( e \) and is derivable from \( e \);

(iii) every variable \( x \in X \) that occurs on the right side of an equation occurs at least once on the left side.

Let \( f \) be a \( k \)-ary function variable. Let \( a \in A^k \), \( b, c \in A \) be such that \( F \vdash f(a) = b \) and \( F \vdash f(a) = c \). Then \( b = c \).

**Proof.** Let \( D \) and \( D' \) be derivations of \( f(a) = b \) and \( f(a) = c \), respectively. The proof is by induction on the length of the derivation \( D = e_0 \cdots e_n \). (Application of \( V_\Sigma \) is not counted as a derivation step.)

**Basis.** \( n = 0 \). In this case, \( f(a) = b \) must be (obtained by applying \( V_\Sigma \)) to an equation of \( F \). By (ii), there is at most one equation in \( F \) such that \( f(a) = c \) is derivable from it and therefore it must be \( f(a) = b \). Hence, \( b = c \).

**Induction on \( n \).** Assume the theorem true for all derivations of length at most \( n \) and let \( e_{n+1} \) be \( f(a) = b \). By (ii), there is exactly one equation \( e \) from which \( f(a) = b \) and \( f(a) = c \) are derivable. By (i), the equation \( e \) must be of the form \( f(t_1, \ldots, t_k) = t \) with \( t \in T_{\Sigma \cup F \cup X} \) and \( t_i \in T_{\Sigma \cup X} \). We are going to show that both derivations
transform $t$ into the same element of $A$. Define the depth, $d(t)$ of a term by letting:

1. $d(x) = d(\sigma) = 0$ for all $x \in X$ and 0-ary $\sigma$,
2. $d(\sigma(t_1, \ldots, t_n)) = \max \{d(t_i) : i = 1, \ldots, n\}$ for $n$-ary $\sigma$,
3. $d(f(t_1, \ldots, t_n)) = \max \{d(t_i) : i = 1, \ldots, n\} + 1$ for $f \in F$.

If $d(t) = 0$, we have two possibilities. Either $t \in T_\Sigma$, in which case $V_A(t) = b$, or $t$ contains some variable $x_i$ and by (iii), $x_i$ must occur at least once on the left side of $e$, say in $t_j$. Because both derivations must transform $t_i$ into $a_j$, and $t_iA$ is injective, the same element of $A$ must be substituted for $x_i$ in $t_i$, and therefore in $t$, in both $D$ and $D'$. Hence, both derivations transform $t$ into the same element $b$.

Assume now that $d(t) > 0$. We shall prove that every subterm of $t$ is transformed into the same element of $A$ by both $D$ and $D'$. The proof is by induction on the depth of the subterm, starting with depth 1. Let $g(u_1, \ldots, u_r)$ be any subterm of $t$ of depth 1, with $g \in F$ and $d(u_i) = 0$, $i = 1, \ldots, r$. If some $u_i$ contains a variable, this variable must appear at least once on the left side of $e$, say in $t_j$. Since $d(t_j) = 0$ and $t_jA$ is injective, both derivations must substitute the element of $A$ for the variable to transform $t_j$ into $a_j$. Therefore the same substitutions must be made in $D$ and $D'$ in transforming $g(u_1, \ldots, u_r)$ into a term of the form $g(d_1, \ldots, d_r)$, with $d_i \in A$. If none of the $u_i$'s contains a variable, we are already in this situation. Because both derivations must transform $g(d_1, \ldots, d_r)$ into an element of $A$, there must be equations of the form $g(d_1, \ldots, d_r) = d$ in $D$ and $g(d_1, \ldots, d_r) = d'$ in $D'$ with $d \in A$ and $d' \in A$, each obtained by a derivation of length at most $n$. Therefore, by the main induction hypothesis, $d = d'$. Hence, every term of depth 1 in $t$ is replaced by the same element of $A$ in both derivations. Assume now that every subterm of $t$ of depth $\leq l$ is replaced by the same element of $A$ by both $D$ and $D'$. Let $g(v_1, \ldots, v_s)$ be of depth $l + 1$. Since $d(v_i) \leq l$, each $v_i$ is replaced by the same element $d_i$ in both derivations and furthermore, by the same reasoning as before, both derivations must have equations of the form $g(d_1, \ldots, d_s) = d$ and $g(d_1, \ldots, d_s) = d'$, respectively. Since the first one has been obtained by a derivation of length $\leq n$, by the main induction hypothesis we have $d = d'$. This completes the induction on the depth. Therefore every subterm of $t$, including $t$ itself, must be replaced by the same element of $A$ in both derivations and the induction on $n$ is complete.

In the preceding theorem, we have intentionally avoided the introduction of an ordering or continuity requirement on $A$. As stated earlier, we can do so when necessary by adjoining a bottom element $\bot$ and extending the operations of $A$ as explained earlier. Considering $CT_{\Sigma \cup X}$, we obtain an infinite tree $t_f$ as the minimal (and unique) fixed point of $F$. In dealing with the infinite tree $t_f$ and relating it to the recursive derived operation $f_A$, as defined by the theorem, we shall use the fact that (in the natural ordering in $CT_{\Sigma \cup X}$) $V_A(t_f(t_1, \ldots, t_k)) = f_A(V_A(t_1), \ldots, V_A(t_k))$, where $V_A$ is a continuous $\Sigma$-evaluation morphism determined by some assignment of values in $A$ to the variables in $t_f$. 
4. Implementation in the Recursive Case

As an example of implementation involving recursively defined simulators, consider the implementation, taken essentially from [2], of a symbol-table of Identifiers and Attribute-lists (for a compiler of a block-structured language) by means of a stack of mappings of domain $\text{Dom}$ (corresponding to the Identifiers) to codomain $\text{Range}$ (in 1–1 correspondence with the Attribute-list). We start with a signature $\text{SYMB}$ with sorts $\text{Sy}$ (symbol table), $\text{Id}$ (Identifier), Boolean and $\text{Atl}$ (Attribute list) containing a distinguished element called $\text{Und}$, and operators:

- $\text{Und} : \rightarrow \text{Atl}$,
- $\text{Le} : \text{Sy} \rightarrow \text{Sy}$ \text{(LEAVEBLOCK)},
- $\text{In} : \rightarrow \text{Sy}$ \text{(INIT)},
- $\text{En} : \text{Sy} \rightarrow \text{Sy}$ \text{(ENTERBLOCK)},
- $\text{Is} : \text{Sy} \times \text{Id} \rightarrow \text{Boolean}$ \text{(ISINBLOCK)},
- $\text{Ad} : \text{Sy} \times \text{Id} \times \text{Atl} \rightarrow \text{Sy}$ \text{(ADDID)},
- $\text{Re} : \text{Sy} \times \text{Id} \rightarrow \text{Atl}$ \text{(RETRIEVE)}.

(The capitalized names are those used by Guttag [2].) The Boolean type is the same as in the STACK example. As in the polynomial case, we are not concerned with the specification of the parameter types $\text{Id}$ and $\text{Atl}$ and we will assume that a set of 0-ary operators is available to represent them. We take as the $\Omega$-algebra $B$, to be implemented, the initial $\text{SYMB}$-algebra characterized by the set $E(\text{SYMB})$ of axioms, taken from [2]:

- $(\text{SY1})$ \text{Le(In)} = \text{In},
- $(\text{SY2})$ \text{Le(En(s))} = s,
- $(\text{SY3})$ \text{Le(Ad(s, x, a))} = \text{Le(s)},
- $(\text{SY4})$ \text{Is(In, x)} = \text{False},
- $(\text{SY5})$ \text{Is(En(s), x)} = \text{False},
- $(\text{SY6})$ \text{Is(Ad(s, x, a), y)} = \text{If } x \text{ Eq } y \text{ Then True Else Is(s, y)},
- $(\text{SY7})$ \text{Re(In, x)} = \text{Und},
- $(\text{SY8})$ \text{Re(En(s), x)} = \text{Re(s, x)},
- $(\text{SY9})$ \text{Re(Ad(s, x, a), y)} = \text{If } x \text{ Eq } y \text{ Then a Else Re(s, y)},

where $s$ is a variable of sort $\text{Sy}$, $x$ and $y$ of sort $\text{Id}$, and $a$ of sort $\text{Atl}$.

As mentioned earlier, we should have introduced two If–Then–Else operators (one of sort $\text{Boolean} \times \text{Boolean} \times \text{Boolean} \rightarrow \text{Boolean}$ and one of sort $\text{Boolean} \times \text{Atl} \times \text{Atl} \rightarrow \text{Atl}$) and corresponding axioms for the axioms $(\text{SY6})$ and $(\text{SY9})$, but, since it would not change our discussion, we choose this more informal and simple notation. Still following [2], the implementor $\Sigma$-algebra $A$ is taken to be an initial algebra $T_{\text{STMP}}/\equiv_{E(\text{STMP})}$, where $\text{STMP}$ and $E(\text{STMP})$ are as follows. The signature
STMP has sorts Stack, Boolean, Dom (domain type), Range (range type), and Map representing Mappings from Dom to Range, and operators

\[
\begin{align*}
\text{Newmap:} & \quad \to \text{Map} \quad \text{(NEWMAP)}, \\
\text{Def:} & \quad \text{Map} \times \text{Dom} \times \text{Range} \to \text{Map} \quad \text{(DEFMAP)}, \\
\text{Ev:} & \quad \text{Map} \times \text{Dom} \to \text{Range} \quad \text{(EVMAP)}, \\
\text{Isdef:} & \quad \text{Map} \times \text{Dom} \to \text{Boolean} \quad \text{(ISDEFINED)},
\end{align*}
\]

in addition to the operators of STK, with the parameter sort Element replaced by the sort Map. Once more, the method used to generate the elements of the parameter types Dom and Range is not essential to our discussion and we will use 0-ary operators to denote them when needed. In fact, we shall assume that Dom is isomorphic to Id and Range to Atl. To define the axioms, let \( m \) be a variable of sort Map, \( x \) and \( y \) of sort Dom and \( r \) of sort Range. Then \( E(\text{STMP}) \) contains the axioms (S1)–(S6) of Section 2 with \( m \) replacing \( e \) and the following:

\[
\begin{align*}
\text{(M1)} & \quad \text{Ev}(\text{Newmap}, x) = \text{Und}, \\
\text{(M2)} & \quad \text{Ev}(\text{Def}(m, x, r), y) = \text{If } x \text{ Eq } y \text{ Then } r \text{ Else } \text{Ev}(m, y), \\
\text{(M3)} & \quad \text{Isdef}(\text{Newmap}, x) = \text{False}, \\
\text{(M4)} & \quad \text{Isdef}(\text{Def}(m, x, r), y) = \text{If } x \text{ Eq } y \text{ Then True Else Isdef}(m, y).
\end{align*}
\]

(We note that (M1), (M2) correspond to (SY7), (SY9), respectively; likewise, (M3), (M4) correspond to (SY4), (SY6). The operations \( En \) and \( Le \) will be simulated using Push and Pop. Thus it should be intuitively "fairly clear" that the implementation is correct in this illustrative example. Yet, the proof requires some care.) To specify an implementation of \( B \) by \( A \), we need to define, for each operation \( \omega_\alpha \) in SYMB, a function \( f_\alpha \) in \( A \) which will make the diagram in Fig. 1 commutative. In addition, the set of simulators \( F_A \) must satisfy conditions (ii) and (iii) for some mapping \( \tau \). While in the implementation of the initial STK-algebra by the initial PTR-algebra given in Section 2 we were able to restrict our attention to polynomial terms in \( T_{\Sigma U \Lambda} \), to define the simulators of the SYMB-operators in \( A \) we need recursive schemes. Although in the previous section we considered more general \( \Sigma \text{GHK} \) schemes, we will limit ourselves here to schemes in a restricted form (as was done in [19]), where either \( t \) or \( s \) is of the form \( f(x_1,\ldots,x_n) \), for some function variable \( f \) and variables \( x_1,\ldots,x_n \) of the appropriate sort. As discussed in Section 3, such a scheme \( F \) defines a function on \( CT_\Sigma \) which can be represented by an infinite tree \( t_f \) in \( CT_{\Sigma U \Lambda} \). The scheme also defines a recursive derived operation \( f_A \) on \( A \cup \{ \bot \} \). However, note that \( f_A \) may be the empty (totally undefined) function. For any assignment of values \( g: X \to A \), we can easily extend the evaluation \( V_{Ag}: T_{\Sigma U \Lambda} \to A \) to \( CT_{\Sigma U \Lambda} \) by letting \( V_{Ag}(t_f(t_1,\ldots,t_n)) = f_A(V_{Ag}(t_1),\ldots,V_{Ag}(t_n)) \). We still use \( V_A \) to denote the \( \Sigma \)-homomorphism corresponding to an empty set \( X \). The whole argument of Section 2 (that is not directly related to the example) still holds after we replace \( T_\Sigma \) by \( CT_\Sigma \). In particular, given a set \( F = \{ t_1,\ldots,t_m \} \) of trees in \( CT_{\Sigma U \Lambda} \) with a corresponding set
IMPLEMENTATION OF DATA TYPES

323

FA = \{FA_1, \ldots, FA_m\} of nonempty (recursive) derived operations on A, we define a derived signature \(\Sigma'\) and the algebras \(A' = (A', FA)\) and \(CT' = (CT', F)\) as in Remark 0. The derived \(F\)-algebra of \(A\) is again defined to be the smallest \(\Sigma'\)-subalgebra of \(A'\) and is denoted by \(F(A)\). Similarly, we call \(F(CT)\) the generalized freely derived \(F\)-algebra. Note that \(CT'\) is nonempty, since \(\Sigma\) is complete. However, for \(A\) we must explicitly assume that the \(FA_i\) are nonempty. Both Remarks 1 and 2 still hold with \(CT,\) replacing \(T,\). In fact, Remark 2 is a special case of the following

Remark 3. Let \(A\) and \(B\) be two \(\Omega\)-algebras for some complete signature \(\Omega\). Let \(F(A)\) and \(F(B)\) be the smallest \(\Omega\)-subalgebras of \(A\) and \(B\), respectively. Then for any \(\Omega\)-homomorphism \(h: A \rightarrow B\) the image of \(F(A)\) is \(F(B)\), that is \(h(F(A)) = F(B)\). In proof, observe that \(h(F(A))\) is an \(\Omega\)-subalgebra of \(B\), since it is the homomorphic image of an \(\Omega\)-subalgebra. Hence, \(F(B) \subseteq h(F(A))\), since \(F(B)\) is the intersection of all \(\Omega\)-subalgebras of \(B\). Similarly, the pre-image, \(h^{-1}(F(B))\), of \(F(B)\) is an \(\Omega\)-subalgebra of \(A\). Therefore, \(h^{-1}(F(B)) \supseteq F(A)\) and so \(F(B) = h^{-1}(F(B)) \supseteq h(F(A)) \supseteq F(B)\), which yields \(h(F(A)) = F(B)\).

In Remark 2, the role of \(\Omega\) is played by \(\Sigma'\), \(A\) corresponds to \(T,\), and \(B\) to \(A'\). The evaluation \(V,\) is the homomorphism \(h\). For the recursive case, we take \(A\) to be \(CT,\), and again \(h\) is the evaluation \(V, : CT, \rightarrow A\).

Going back to our example, for each operator in SYMB, let us define the corresponding STMP-simulators, which will be denoted by the capitalized name of the SYMB-operator preceded by the letter \(F\), by the scheme SIM(SYMB) given as follows:

\[
\begin{align*}
FIN &= \text{Push}(\text{Newstack}, \text{Newmap}), \\
FEN(st) &= \text{Push}(st, \text{Newmap}), \\
FAD(st, id, r) &= \text{Push}(\text{Pop}(st), \text{Def}(\text{Top}(st), id, r)), \\
FIS(st, id) &= \text{Isdef}(\text{Top}(st), id), \\
FLE(st) &= \text{If Isnew (Pop(st)) Then Push (Newstack, Newmap) Else Pop(st)}, \\
FRE(st, id) &= \text{If Isnew(st) Then Und Else If Isdef(Top(st), id)} \\
& \quad \text{Then Ev(Top(st), id) Else FRE(Pop(st), id)},
\end{align*}
\]

where \(st, id\), and \(r\) are variables of sort Stack, Dom, and Range, respectively. As for the polynomial case, there exists a unique SYMB-evaluation homomorphism \(\Theta: T_{\text{SYMB}} \rightarrow F(CT_{\text{STMP}})\) mapping, for example, \(\text{En(In)}\) to \(\text{Push(Push(Newstack, Newmap), Newmap)}\).

For the definition of Implementation, we only need to substitute \(CT,\) for \(T,\) in conditions (I)-(IV). Note that the requirement that \(F(A)\) be an \(\Omega\)-algebra together with (IV) implies that \(FA_i \neq \perp\) for all \(i\) unless \(B\) is a trivial \(\Omega\)-algebra (i.e., having a totally undefined operation). We still have \(\Theta: T_{\Omega} \rightarrow F(CT)\) an \(\Omega\)-epimorphism, \(V, (F(CT)) = F(A)\), etc. Figure 3 is replaced by
However, we require further that all homomorphisms be strict (i.e., map \( \bot \) onto \( \bot \)). To verify that our example is a \( B \)-by-\( A \) implementation, we only need to prove that (IV) holds and this will be done again in the Appendix.

We conclude this section with a comment regarding computations on data types and composition of implementations. After an \( \Omega \)-algebra \( B \) has been implemented by a \( \Sigma \)-algebra \( A \) via the construction of \( F(A) \) and \( \tau \), we want to be able to simulate in \( A \) derived operations on \( B \). The translation in \( F(A) \) of derived operations on \( B \) defined by polynomial terms (straight-line computations) is straightforward, while a little care is needed in dealing with more general procedures defined, say, using \( \Sigma \)GHK schemes. More work needs to be done in this direction. Nevertheless, it can be shown that if \( F \) is a program scheme of the restricted form in the theorem defining a recursive derived operation \( f_B \), on \( B \), then there exists a derived operation \( f_A \), defined by a program scheme on \( F(A) \), which we denote by \( \Theta(F) \), such that if \( F \vdash f_B(b_1, \ldots, b_n) = b_0 \), then for \( a_1, \ldots, a_n \in F(A) \) such that \( \tau(a_i) = b_i \), \( \Theta(F) \vdash f_A(a_1, \ldots, a_n) = a_0 \) and \( \tau(a_0) = b_0 \). This guarantees then that if \( (F_B(A), \tau_B) \) is a correct implementation of \( B \) by \( A \) and \( (F_C(B), \tau_B) \) a correct implementation of \( C \) by \( B \), then \( (F_C(F_B(A)), \tau_B \circ \tau_A) \) is a correct implementation of \( C \) by \( A \). Details will appear in a forthcoming paper.

5. Comparison with Other Approaches

A major difference between our development and the approach in Guttag et al. [2] and Ehrig et al. [13] is that we always keep the algebra \( B \) to be implemented and the implementor \( A \) separated. The sorts and operations of the two algebras are not combined and the simulators are constructed using only the operators in \( \Sigma \) and the \( \Sigma \)GHK program schemes, avoiding thereby the problem of type-protection. The correspondence between sorts and operators is then established explicitly via the maps \( F \), \( \tau \), and \( \Theta \). Although the algebras we used in our examples are presented by equational specifications, our approach does not require that the algebras \( A \) and \( B \) be given by equational initial algebra semantics as is the case in Ehrig et al. [13]. This is a crucial point for, as shown in Bergstra and Tucker [10], there are algebras (in particular, the algebra of primitive recursive functions) which fail to possess a recursive enumerable conditional hidden enrichment specification with respect to initial algebra semantics. What we need is to be able to determine whether two free \( \Sigma \) (or \( \Omega \)) terms are equivalent in \( A \) (or \( B \)). Furthermore, since the algebras are not required to be specified by initial algebra semantics, we do not have to restrict our attention to axioms in equational (including conditional) form and can allow more general axioms, such as, for example, Horn-like sentences.
In Guttag et al. [2] the mapping \( \tau \) is included in the "representation" of the implementation and it is not required to be homomorphism, forcing Guttag to prove the "correctness" of \( F(A) \), that is, to prove that \( F(A) \) satisfies the axioms used to specify the algebra \( B \). Besides the advantage of keeping the two algebras separated mentioned above, the requirement that \( \tau \) be a well-defined homomorphism guarantees that \( F(A) \) "satisfies" the equational axioms in \( B \) up to the kernel of \( \tau \), in the sense that if \( B \) satisfies the axiom \( t = s \) (i.e., \( V_B \tau t = V_B \tau s \)), then \( \tau V_A \Theta t = \tau V_A \Theta s \). The inclusion of infinite trees in the syntactical definition of simulators has allowed us to extend the notion of derived algebras. We believe this extension to be sufficient and close to the idea of "inductively specified operators" of Ehrig et al. in [13]. There are some similarities between our approach and that of Ehrig et al. [21] if we restrict our attention to initial algebra semantics. We explain the similarities by analyzing in detail their approach in the case where the algebras \( A \) and \( B \) are given by an equational initial semantics specification. Let us assume then that \( A \) is the initial algebra of the specification \( \text{SPEC}1 = \text{SPEC} + \langle S1, \Sigma1, E1 \rangle \) and \( B \) the initial algebra specified by \( \text{SPEC}0 = \text{SPEC} + \langle S0, \Sigma0, E0 \rangle \), with \( \text{SPEC} \) a specification of a common subtype. (\( S \) denotes sorts, \( \Sigma \) signatures, and \( E \) equations. See [21] for details.) The first step in the Ehrig et al. approach is to construct \( T_{\text{SORTIMPL}} \), the initial algebra specified by \( \text{SORTIMPL} = \text{SPEC}1 + \langle S0, \Sigma\text{SORT}, E\text{SORT} \rangle \). Here is where the correspondence between the sorts of \( A \) and those of \( B \) is established by using the "sort-implementing" operations \( \Sigma\text{SORT} \), like the pairing and projection operations used in our Example 1 to relate the sort Pointer-Array to the sorts Integer and Array. \( E\text{SORT} \) in our Example 1 are the axioms \((P3)-(P5)\). The algebra \( T_{\text{SORTIMPL}} \) corresponds, in our approach, to \( A \) extended by the sorts of \( B \) together with the sort part of the homomorphism \( \tau \). The next stage in their development is the construction of \( T_{\text{OPIMPL}} \), the initial algebra of the specification \( \text{OPIMPL} = \text{SORTIMPL} + \langle \Sigma0, E\text{OP} \rangle \), where the \( \Sigma0 \)-operators are added, along with the correspondence, through the equations \( E\text{OP} \), between the operators in \( B \) and those in \( A \). The \( \Sigma1 \)-simulators of the \( \Sigma0 \)-operations are not explicit but are obtained in the terms used to form the \( E\text{OP} \) equational axioms. In our approach, \( T_{\text{OPIMPL}} \) corresponds to the enrichment of the extension of \( A \) (obtained in Step 1) by the simulators along with the operations part of \( \tau \), given in operational form through the use of the sort-operations \( \Sigma\text{SORT} \) and the axioms in \( E\text{OP} \). The next step consists of two distinct "reductions" leading to the algebra \( \text{REP}_{\text{IMPL}} \). The first is accomplished by using the functor \( \text{FORGETTING} \), which eliminates all the elements which are not of sort \( S0 \) (or belonging to the common subtype) and the operators in \( \Sigma1 \) and \( \Sigma\text{SORT} \). The algebra so obtained, denoted in the Ehrig et al. approach by \((T_{\text{OPIMPL}})_{\text{SPEC}0} \), is still too large: it contains, for example, any element of sort \( S1 \) to which an appropriate operation from \( \Sigma\text{SORT} \) has been applied, regardless of the way the simulators have been defined. The second reduction is obtained by selecting only those elements in \((T_{\text{OPIMPL}})_{\text{SPEC}0} \) which can be generated using the simulators alone. It is only at this point that the two approaches coincide: \( \text{REP}_{\text{IMPL}} \) is equivalent to our "derived algebra" \( F(A) \).

Since we are not extending the signatures \( \Omega \) and \( \Sigma \), the notion of type-protection does not occur. However, we require that the simulators induce well-defined (for
consistency) and totally defined (for completeness) functions on the implementor algebra, (possibly after introducing a flat ordering). The ZO-completeness is "built-in," in our approach, since we use Σ-operators (and ΣGHK program schemes) to define the simulators.

The protection of a common subtype (such as Boolean) is guaranteed by the fact that the restriction of τ to elements of the subtype is the identity homomorphism. Finally, the RI-correctness of EKP is equivalent (because of [19, Theorem 5.5]) to our requirement that τ be a well-defined homomorphism.

**APPENDIX**

Proof that SIM(STK) is a B-by-A-Implementation. We verify that θ: T_{stk} → F(T_{ptar}) is injective. Consider terms of sort Stack, beginning with Newstack. We have θ(Newstack) = (Newarray, 0). Since F(T_{ptar}) ⊆ T_{ptar} and since T_{stk} and T_{ptar} are free algebras, we see from SIM(STK) that for any other Stack term t we must have θ(t) ≠ θ(Newstack). Indeed, if t = Push(s, e), where s is a stack term, then

\[ θ(t) = \text{FPUSH}_{F(T_{PTAR})}(θ(s), θ(e)) = \langle \text{Assign}(\text{Pr1}(θ(s)), \text{Pr2}(θ(s)) + 1, θ(e)), \text{Pr2}(θ(s)) + 1 \rangle. \]

(Henceforth, we shall designate the derived operations in F(T_{PTAR}) simply by the derived operator without the subscript F(T_{PTAR}).) If t' = Push(s', e'), then θ(t) = θ(t') implies θ(s) = θ(s'). By induction on the depth of a Push Stack term, we have that t ≠ t' implies s ≠ s' which implies θ(s) ≠ θ(s') by induction hypothesis. By the above, this implies θ(t) ≠ θ(t'). Similarly, if t = Pop(s), then θ(t) = FPOP(θ(s)) = If Pr2(θ(s)) = 0 Then (Pr1(θ(s)), Pr2(θ(s)) - 1) Else θ(s). Again, we see that for two such Pop Stack terms θ(t) = θ(t') implies θ(s) = θ(s'). By a similar induction on depth, θ(s) ≠ θ(s') implies θ(t) ≠ θ(t') for t ≠ t'. Further analysis of SIM(STK) shows that θ is also 1-1 for terms of Element sort. Thus θ is injective and to prove that (IV) holds, it suffices to prove that V_\ast(t) = V_\ast(s) → V_\ast_B(θ^{-1}(t)) = V_\ast_B(θ^{-1}(s)) for t, s ∈ F(T_{PTAR}). Now, V_\ast(t) = V_\ast(s) means that t =_A s, where =_A is the equivalence on T_{PTAR} generated by the axioms in E(P TAR). Since t can be of three different sorts, we consider three cases.

**Case 1.** Sort (t) = Boolean. Since FISNEW is the only derived operation of sort Boolean, t = FISNEW(t'), where sort(t') = Ptr-arr. Hence, V_\ast(t) = V_\ast(t').

**Case 1a.** Suppose first that V_\ast(t) = True. From the defining equations for FISNEW it follows that V_\ast(t') = (a, 0) for some a ∈ Array. From the definition of θ and the simulators FPUSH and FPOP, and the axioms in E(STACK), it is quickly seen that V_\ast_B(θ^{-1}(t')) = Newstack. (A rigorous proof would use induction on the number of FPUSH and FPOP operations in t'.) Hence, V_\ast_B(θ^{-1}(t')) = V_\ast_B(θ^{-1}(t')) = Isnew(Newstack) = True. Thus, τV_\ast(t) = True is well defined for such t.
Case 1b. Suppose $V_A(t) = \text{False}$. Then $V_A(t') = (a, j)$, where $j \neq 0$. By a similar argument, we show that $V_B(\theta^{-1}t') \neq \text{Newstack}$. Hence, $V_B(\theta^{-1}\text{FISNEW}(t')) = \text{Isnew}(V_B(\theta^{-1}t')) = \text{False}$ and so $\theta^{-1}t \equiv_b \text{False}$ for all such $t$.

Case 2. Sort($t$) = Sort($s$) = Ptr-arr. From the defining equations for the simulators FPUSH and FPOP and the fact that the only axiom in $E(\text{PTAR})$ that could produce equivalence is (P5), it follows that there must exist an $n$-ary polynomial $p \in F(T_{\text{PTAR}} \cup X)$ of sort Ptr-arr and terms $r_i, q_i \in F(T_{\text{PTAR}})$, $1 \leq i \leq n$, of sort Element such that $t = p(r_1, \ldots, r_n)$, $s = p(q_1, \ldots, q_n)$, and $r_i \equiv_a q_i$ for $1 \leq i \leq n$. Since $\theta^{-1}t = \theta^{-1}p(\theta^{-1}r_1, \ldots, \theta^{-1}r_n)$ and $\theta^{-1}s = \theta^{-1}p(\theta^{-1}q_1, \ldots, \theta^{-1}q_n)$, this case reduces to proving that $V_B(\theta^{-1}r_i) = V_B(\theta^{-1}q_i)$.

Case 3. Sort($t$) = Sort($s$) = Element. Suppose $t \equiv_A s$. Then there is a derivation sequence $t = t_1 \rightarrow t_2 \rightarrow \cdots \rightarrow t_m = s$, where $t_i \rightarrow t_{i+1}$ means that term $t_{i+1}$ is obtained from $t_i$ by applying an equation in $E(\text{PTAR})$ as a rewrite rule. We wish to prove that $V_B(\theta^{-1}t) \equiv B \theta^{-1}s$. We use induction on $m$, the length of a derivation. This is clearly true for $m = 1$, since then $t = s$. Assume that it is true whenever two terms are equivalent in $A$ by a derivation of a length $< m$ and suppose $t \equiv_A s$ by a derivation of length $m$ as above. There are two subcases.

Case 3a. $t_{m-1} = \text{Access}(\text{Assign}(a, n+1, s), n+1)$, where $a$ is a term in array and $n \in \text{Integer}$, so that $t_{m-1} \rightarrow s$ by axiom (P2.1). Now, from the structure of $F(T_{\text{PTAR}})$, the axioms for $A$ and the equations in $\text{SIM(STACK)}$, we see that there must be a term in the sequence of the form $t_i = \text{FTOP}(\text{FPUSH}((a, n), e))$, where $(a, n)$ is a term in Ptrarr such that $\theta^{-1}(a, n)$ is defined and $e$ is a term in Element such that $e \equiv_A s$ by a derivation of length $< m$. By condition I, and axiom (S4), $V_B(\theta^{-1}t_i) = V_B(\text{Top}(\text{Push}(\theta^{-1}(a, n), \theta^{-1}e'))) = V_B(\theta^{-1}e')$. By the induction hypothesis, $V_B(\theta^{-1}e') = V_B(\theta^{-1}s)$ and $V_B(\theta^{-1}t) = V_B(\theta^{-1}t_i)$. Hence, $V_B(\theta^{-1}t) = V_B(\theta^{-1}s)$.

Case 3b. $t_{m-1} = \text{Access}(\text{Assign}(a, n, e), p)$, where $n \neq p$, and $s = \text{Access}(a, p)$, so that $t_{m-1} \rightarrow s$ by axiom (P2.2). From the equations in $\text{SIM(STACK)}$, we see that no sequence of derived operations can produce a pointer value $p$ in a term like $t_{m-1}$, which is greater than $n$. Hence, $p = n - k$, $k > 0$. Also, note that the only axioms in $E(\text{PTAR})$ for equivalence of arrays of pointer arrays are the trivial ones (P3) (P5). Thus, $t_{m-1}$ must be derived from a prior term $t_i$ of the form, $t_i = \text{FTOP}(\text{FPUSH}((a', n-1), e'))$, where $(a', n-1) \equiv_A (a, n-1)$, $e' \equiv_A e$ and $a'$ is an Array term built up from at least $n - 1$ FPUSHs applied to Newarray. Hence $\theta^{-1}t_i = \text{Top Pop}^k \text{Push}(\theta^{-1}(a', n-1), \theta^{-1}e')$. Using the induction hypothesis on $a'$ and $a$,

\[
V_B(\theta^{-1}t_i) = \text{Top Pop}^k \text{Push}(V_B(\theta^{-1}(a', n-1)), V_B(\theta^{-1}e')) = \text{Top Pop}^{k-1} \text{Push}(V_B(\theta^{-1}(a, n-1)), V_B(\theta^{-1}e')) = \text{Top Pop}^{k-1}(V_B(\theta^{-1}(a, n-1))) = V_B(\theta^{-1}\text{FPOP}^{k-1}((a, n-1))) = \text{Top}(V_B(\theta^{-1}(a, p))) = V_B(\theta^{-1}\text{FTOP}(a, p)) = V_B(\theta^{-1}) \text{Access}(a, p) = V_B(\theta^{-1}s).
\]
As in the previous example, it is not too hard to verify that $\Theta: T_{\text{SYMB}} \rightarrow F(\text{CT}_{\text{STMP}})$ is injective, allowing us to consider the inverse SYMB-homomorphism $\Theta^{-1}$. To show that condition (IV) is satisfied, it suffices to prove that $V_\alpha(t) = V_\alpha(s) \rightarrow V_\beta \Theta^{-1}(t) = V_\beta \Theta^{-1}(s)$ for all $t, s \in F(\text{CT}_{\text{STMP}})$. We will denote $V_\alpha(t) = V_\alpha(s)$ by $t \equiv_\alpha s$, where $\equiv_\alpha$ is the congruence generated by the axioms $E(\text{STMP})$. By a simple inductive argument, it suffices to show that (IV) holds when $t \equiv_\alpha s$ via a single application of one axiom of $E(\text{STMP})$ and, since $\Theta^{-1}$ is a homomorphism, we only need to consider the outermost application of the axiom.

**Case 1: Sort ($t = \text{Stack}$).** Since Newstack $\not\in F(\text{CT}_{\text{STMP}})$ (the proof is in [2]), the only axiom which can be used is (S2) and, by the way the simulators are defined, it can be applied only if $t = \text{FLE}(st)$ and either $st = \text{FEN}(s)$ or $st = \text{FAD}(\text{Push}(s, m), \text{id}, r)$. (The condition If-Then-Else of FLE($st$) prevents $st$ from being FIN). In the first instance, $t = \text{Pop Push}(s, \text{Newmap}) \equiv_\alpha s$; in the second one, $t = \text{Pop FAD}(\text{Push}(s, m), \text{id}, r) = \text{Pop Push}(\text{Pop Push}(s, m), \text{Def}(\text{Top Push}(s, m), \text{id}, r))$. If $t = \text{FLE}(\text{FEN}(s))$, then $\Theta^{-1}(t) = \text{Le}(\text{En}(\Theta^{-1}(s))) \equiv_\beta \Theta^{-1}(s)$. Otherwise, if $t = \text{FLE}(\text{FAD}(\text{Push}(s, m), \text{id}, r))$, then $\Theta^{-1}(t) = \text{Le}(\text{Ad}(\Theta^{-1}(\text{Push}(s, m), \text{\text{id}, r})) \equiv_\beta \Theta^{-1}(\text{Push}(s, m))$ by axiom (SY3). Hence $\Theta^{-1}(t) = \Theta^{-1}(\text{FLE}(\text{Push}(s, m)))$.

Now, a number $k$ of operations Def is used to construct $m$ by applying FAD to FEN($s$). It is easy to see that applying axiom (SY3) $k$ times yields $\Theta^{-1}(\text{FLE}(\text{Push}(s, m))) = \Theta^{-1}(\text{FLE}(\text{FEN}(s))) \equiv_\beta \Theta^{-1}(s)$.

**Case 2: Sort ($t = \text{Range}$).** By the remark at the beginning of the previous case, the only two axioms of this sort that can be applied are $\text{Ev}(\text{Def}(m, x, r), x) = r$ and $\text{Ev}(\text{Def}(m, x, r), y) = \text{Ev}(m, y)$. By the definition of the simulators, we must have either $t = \text{FRE}(\text{FAD}(t', x, r), x)$ or $t = \text{FRE}(\text{FAD}(t', x, r), y)$, where $m = \text{Top}(t')$. In the first case, $t \equiv_\alpha r$ and $\Theta^{-1}(t) = \text{Re}(\text{Ad}(\Theta^{-1}(t'), \Theta^{-1}x, \Theta^{-1}r), \Theta^{-1}x) \equiv_\beta \Theta^{-1}r$. In the second case $t \equiv_\alpha \text{Ev}(\text{Top}(t'), y)$ and $\Theta^{-1}(t) = \text{Re}(\Theta^{-1}(t'), \Theta^{-1}x, \Theta^{-1}r, \Theta^{-1}y) \equiv_\beta \text{Re}(\Theta^{-1}t', \Theta^{-1}y) = \Theta^{-1}(\text{FRE}(t', y))$.

Notice that the definition of $\Theta$ as a bijective map from Dom to Id and from Range to Atl has been used here in stating that $\Theta^{-1}(x) \neq \Theta^{-1}(y)$ whenever $x \neq y$.

**Case 3: Sort ($t = \text{Boolean}$).** Axiom (M3) can be applied only if $t = \text{FIS}(st, \text{id})$ with $\text{Top}(\text{St}) = \text{Newmap}$. By the way the simulators were defined, $st$ can be either FIN or FEN($t'$). In either case, $t \equiv_\alpha \text{False}$. In the former case, $\Theta^{-1}(t) = \text{Is}(\text{In}, \Theta^{-1}\text{id}) \equiv_\beta \text{False}$ and in the latter one $\Theta^{-1}(t) = \text{Is}(\text{En}(\Theta^{-1}t'), \Theta^{-1}\text{id}) \equiv_\beta \text{False}$.

Axiom (M4) can be applied only if either $t = \text{FIS}(\text{FAD}(t', x, r), x)$, in which case $t \equiv_\alpha \text{True}$ and $\Theta^{-1}(t) = \text{Is}(\text{Ad}(\Theta^{-1}t', \Theta^{-1}x, \Theta^{-1}r), \Theta^{-1}x) \equiv_\beta \text{True}$ or $t = \text{FIS}(\text{FAD}(t', x, r), y)$ in which case $t \equiv_\alpha \text{Isdef}(\text{Def}(t', x, r), y) \equiv_\alpha \text{Isdef}(t', y)$ and $\Theta^{-1}(t) = \text{Is}(\text{Ad}(\Theta^{-1}t', \Theta^{-1}x, \Theta^{-1}r), \Theta^{-1}y) \equiv_\beta \text{Is}(\Theta^{-1}t', \Theta^{-1}y) = \Theta^{-1}(\text{FIS}(t', y)) = \Theta^{-1}\text{Isdef}(t', y)$.
Finally, it is clear that FRE is defined for all $st \in F(A)$. Thus, there is no $t \in T_{SYM}$ such that $V_A t = \bot$ and so $\{\tau(\bot)\} = \{\bot\}$.

REFERENCES