NONDETERMINISM IN ABSTRACT DATA TYPES

P. A. Subrahmanyam
Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

ABSTRACT

A nondeterministic operation is characterized by the fact that its application to a given set of parameters can yield any one of several possible outcomes. This paper discusses ways to specify, implement, and reason about nondeterministic operations in the context of abstract (algebraic) data types. The notion of an implementation of a data type that includes nondeterministic operations is formalized, and the criteria for judging the "correctness" of such implementations are specified. The formalism developed allows implementations of nondeterministic operations to embody varying degrees of the full extent of nondeterminism allowed by the semantics of a type; in particular, deterministic implementations of nondeterministic operations are allowed.

Keywords nondeterminism, nondeterministic operations, abstract data types, correctness, implementations, extraction equivalence, observable behavior.

1. Introduction

Although the importance of the notion of abstract data types [1], [2], [4], [6] has by now gained almost ubiquitous acceptance, several problematic issues still remain unresolved. Two of the factors that have significantly impeded experimentation are the lack of effective mechanisms for specifying and reasoning about (i) nondeterminism and (ii) exceptional conditions. In this paper, we outline a method for dealing with nondeterminism in the context of abstract algebraic data types. Specifically, we are interested in being able to

- specify (i.e., give semantics for) nondeterministic operations;
- state and prove properties about nondeterministic operations;
- define the notion of implementations of data types that involve nondeterministic operations, and specify criteria for judging the "correctness" of such implementations;
- characterize the notion of the "degree" of nondeterminism embodied in an implementation of a type;
- develop techniques to prove the correctness of implementations in the presence of nondeterminism.

Further, we are here interested in being able to achieve the above objectives by using deterministic characterizations of nondeterministic operations in so far as this is possible.[1] Although there are several reasons for adopting this route, our major rationale here is twofold (i) to preserve the machinery for reasoning about deterministic operations and (ii) to avoid the additional complexity we might incur by introducing a completely different notation for talking about nondeterministic operations.

It is desirable to be able to continue to make statements about nondeterministic operations, their implementations, and the correctness of such implementations in the normal fashion -- by writing (logical) expressions involving them. It will turn out that the only parts of such expressions that have to be carefully interpreted are those that contain the equality symbol: the interpretation of equivalence we adopt is therefore central to our enterprise. In order to achieve the objectives enumerated above, it is therefore necessary to

- define a notion of equivalence of instances of a type in the presence of nondeterministic operations;
- detail how expressions involving nondeterministic operations are to be interpreted.

In the rest of this paper, we flesh out the plan outlined above.

2. Preliminary Definitions

In this section we introduce some needed terminology relating to abstract data types and nondeterministic functions.

2.1. Abstract Data Types, Nondeterministic Operations, Characteristic Predicates

Intuitively, the abstraction of a problem can be viewed as consisting of an appropriate set of operations that manipulate associated sets of objects of various sorts. An abstract data type is nothing but a formal characterization of such a viewpoint, as is reflected by the following definition.

Definition 1: An abstract (algebraic) data type consists of a set X of <u>sorts</u>, a set F of function symbols, and a set of equations relating <u>terms</u> generated by F and containing free variables. Each f in F has an associated <u>arity</u> that is an element $(x_1x_2...x_n,x_{n+1})$ 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.)

[1] Of course, a different way of approaching this problem (and one which may very well provide a better alternative!) is to develop methods that are entirely independent of those in the deterministic domain. We do not do this here.

<u>Type Set-of-Integer</u>

```
Syntax
  EMPTYSET: () -> Set
  INSERT: Set, Integer -> Set
  DELETE: Set,Integer -> Set
  ISEMPTY: Set -> Boolean
  MEMBER: Set, Integer -> Boolean
  CHOOSE: Set -> Integer U {UNDEFINED}
              -- CHOOSE is nondeterministic.
  {P_CHOOSE:Set,Integer -> Boolean is the characteristic predicate of CHOOSE}

Semantics

  for all s in Set, x,y in Integer,

  DELETE(EMPTYSET,x) = EMPTYSET
  DELETE(INSERT(s,x),y) = if x=y then DELETE(s,y) else INSERT(DELETE(s,y),x)

  ISEMPTY(EMPTYSET) = TRUE
  ISEMPTY(INSERT(s,x)) = FALSE

  MEMBER(EMPTYSET,x) = FALSE
  MEMBER(INSERT(s,x),y) = if x=y then TRUE else MEMBER(s,y)

  CHOOSE(EMPTYSET) = UNDEFINED
  P_CHOOSE(s,x) ≡ MEMBER(s,x)
              -- the characteristic predicate of CHOOSE
```

<u>End Set</u>

Figure 2-1: Definition of the Type Set-of-Integer[2]

---

[2]For the purposes of this paper, we ignore the technicalities arising out of the presence of parameterized types and functions returning "exceptional" values (see [2], [9], [11]). However, the reader's intuition will not lead him astray in his comprehension of this paper.

We permit the "functions" in F to be nondeterministic.[3] In the context of abstract data types, nondeterminism implies that there is an element of choice in the outcome of the operation. Whenever it is necessary to highlight the fact that an function f is nondeterministic, we will use a distinguishing font: $\underset{\sim}{f}$. The function $\underset{\sim}{\text{CHOOSE}}$ defined on a Set is a common example of a nondeterministic function: intuitively, $\underset{\sim}{\text{CHOOSE}}$(s) returns a random element picked from the non-empty set s. In general, if $\underset{\sim}{f}:T_1,\dots,T_n \rightarrow T$ is a nondeterministic function, and if $t_i \in T_i$, i=1..n, then $\underset{\sim}{f}(t_1,\dots,t_n)$ can be any element from some set of T which we shall henceforth denote $\{\underset{\sim}{f}(t_1,\dots,t_n)\}$; such a set is invariably circumscribed by a deterministic predicate. We will presume that that the characteristic predicate of this set is always deterministic and we will henceforth refer to it as the characteristic predicate of the function $\underset{\sim}{f}$, and denote it $P_{\underset{\sim}{f}}$.

Definition 2: The characteristic predicate of a nondeterministic function $\underset{\sim}{f}:T_1,\dots,T_n\rightarrow T$ is a deterministic predicate

$$P_{\underset{\sim}{f}}:T_1,\dots,T_n,T \rightarrow \text{Boolean}$$

that is defined by

$$P_{\underset{\sim}{f}}(t_1,\dots,t_n,t) = \text{TRUE iff } t \in \{\underset{\sim}{f}(t_1,\dots,t_n)\}$$

i.e. if $\underset{\sim}{f}(t_1,\dots,t_n)$ can evaluate to t (sometime). ■

The introduction of P_f enables us to explicitly state that a specific application of a nondeterministic function $\underset{\sim}{f}(t_1,\dots,t_n)$ resulted in a value r, and what is important, to refer to this value r in more than one place. Quite often, we will specialize P_f to a fixed set of arguments of f, i.e. we will use $P_{f,t_1,\dots,t_n}:T\rightarrow$Boolean as the characteristic predicate of the set $\{\underset{\sim}{f}(t_1,\dots,t_n)\}$. Henceforth, whenever there is no cause for confusion, we will abbreviate $P_{f,t_1,\dots,t_n}$ as P_f.

As an example, MEMBER(s,$\underset{\sim}{\text{CHOOSE}}$(s)) is the characteristic predicate of the nondeterministic function $\underset{\sim}{\text{CHOOSE}}$; it states that all of the possible outcomes of $\underset{\sim}{\text{CHOOSE}}$ must be contained in s. That is, $P_{\underset{\sim}{\text{CHOOSE}}}$:Set,Integer -> Boolean, and [$\underset{\sim}{\text{CHOOSE}}$(s)=x => MEMBER(s,x)=TRUE].

2.2. Terms of a Type

The "words" or terms of type x generated by the (set of) functions F and the (set of) variables V, consist of the set of syntactic terms of sort x that are obtained by composing the functions in F. This set of terms, denoted $W_x[F,V]$, can be defined inductively as the union of the sets $W_x^{(n)}[F,V]$, n = 0,1,2... where the superscripts indicate the level of "nesting" in the function compositions; $W_x^{(0)}[F,V]$ contains all the variables and constants of type (or sort) x. The algebra obtained by

[3]Since it is quite common (albeit erroneous!) to term nondeterministic operations as "nondeterministic functions", we shall interchangeably use the words function and operation here.

interpreting the functions F over the terms $\{W_x[F,V] \mid x \in X\}$ is called the word algebra defined by F, V.

The data type Set-of-Integer can be viewed as consisting of

- the set of sorts X, X = {Set, Integer, Boolean}, the sorts themselves being Set, Integer, Boolean;
- the set of function symbols F^{Set} = {EMPTYSET, INSERT, DELETE, ISEMPTY, MEMBER, CHOOSE}, 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

  $W_{Set}[F^{Set}, \{x,y,...\}]$ =
  {EMPTYSET, INSERT(EMPTYSET,x),
  INSERT(EMPTYSET,y), ..., INSERT(INSERT(EMPTYSET,x),x), ...,
  DELETE(EMPTYSET,y), ... } etc.;

  $W_{Integer}[F^{Set}, \{x,y,...\}]$ = {CHOOSE(INSERT(EMPTYSET,x)), ...} etc.
- the equations are those shown in figure 2-1.

Figure 2-2: Terms generated by F^{Set}

2.3. 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. Often, when defining a type T (e.g., Set-of-Integer,) other types like Integer and Boolean are presumed to be "known" or "global" types -- we will denote this set of types by G. 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.

3. Characterizing Externally Observable Behavior

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. EMPTYSET), Constructors (C^T), which form new instances of the type from existing ones (e.g. INSERT, DELETE), and extraction functions or extractors (E^T), which return members of other "known" types (e.g. CHOOSE, ISEMPTY, MEMBER).

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.

Informally, two terms t_1 and t_2 are said to be *extraction equivalent* [8] 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.

However, care is required when dealing with expressions involving nondeterministic functions, since $\underset{\sim}{f}(t_1,\ldots,t_n)$ may, in general, yield any element in a *set* $\{\underset{\sim}{f}(t_1,\ldots,t_n)\}$. If we want to preserve the ability to reason inductively using the structure of terms in the word algebra, it is imperative to adopt a notion of equivalence that preserves the "substitution property" [3]. For example, given that $s_1=s_2$ we want to appropriately interpret an equation like $\underset{\sim}{\text{CHOOSE}}(s_1)=\underset{\sim}{\text{CHOOSE}}(s_2)$; in particular, we desire that $\underset{\sim}{\text{CHOOSE}}(s)=\underset{\sim}{\text{CHOOSE}}(s)$ be interpreted to mean that the set of possible outcomes of the two sides of the expression are extraction equivalent, and not that the specific elements each side evaluates to are always identical.

We therefore adopt the following definition: two terms t_1 and t_2 of type T are extraction equivalent, denoted $t_1 =_T t_2$ if the sets $\{t_1\}$ and $\{t_2\}$ are (extraction) equivalent; in the case that t_1 and t_2 do not contain any nondeterministic functions, these sets contain only one element.

As an example, two instances of the type Set (say, s_1 and s_2) are extraction equivalent iff the applications $\underset{\sim}{\text{CHOOSE}}(s_1)$ and $\underset{\sim}{\text{CHOOSE}}(s_2)$, $\underset{\sim}{\text{CHOOSE}}(\text{DELETE}(s_1,y_1))$ and $\underset{\sim}{\text{CHOOSE}}(\text{DELETE}(s_2,y_1))$, ..., $\underset{\sim}{\text{CHOOSE}}(\text{INSERT}(s_1,x_1))$ and $\underset{\sim}{\text{CHOOSE}}(\text{INSERT}(s_2,x_1))$, ..., ISEMPTY($s_1$) and ISEMPTY($s_2$), ISEMPTY(DELETE($s_1$,$y_1$)) and ISEMPTY(DELETE($s_2$,$y_1$)), ..., ISEMPTY(INSERT($s_1$,$x_1$)) and ISEMPTY(INSERT($s_2$,$x_1$)), ..., MEMBER($s_1$,$z_1$) and MEMBER($s_2$,$z_1$), MEMBER(DELETE($s_1$,$y_1$),$z_1$) and MEMBER(DELETE($s_2$,$y_1$),$z_1$), ..., MEMBER(INSERT($s_1$,$x_1$),$z_1$) and MEMBER(INSERT($s_2$,$x_1$),$z_1$), ..., yield equivalent results pairwise. Specifically, equivalent results should obtain when the variables (the x's, y's, and z's) are substituted with Integers.

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 $t[v{\in}V_T|t']$ the term obtained by substituting t' for *all* occurrences, in t, of variables that are contained in V_T.

Definition 3: t_1 and t_2 are said to be *extraction equivalent* in T, denoted $t_1 =_T t_2$ if and only if

either (i) $t_1 = t_2$ (i.e. the terms are syntactically equivalent),

or (ii) $(\forall g \in G)(\forall t_g \in W_g)\ (t_g[v{\in}V_T|t_1] =_g t_g[v{\in}V_T|t_2])$.

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. As we have already stressed, and will further elaborate upon below, *the equality in (ii) has to be properly interpreted in the case when t_g involves* nondeterministic functions. However, 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, ■

$$t_1 = t_2 \Rightarrow t_1 =_T t_2.$$

4. Deterministic Equivalents of Nondeterministic Expressions

If expressions involving nondeterministic functions can be translated into equivalent expressions involving only deterministic functions, then we can benefit from the techniques that are already available for dealing with deterministic expressions. In this section, we indicate how logical expressions that involve the connectives $\wedge$, $\vee$, $\neg$, $=$, the quantifiers $\forall$ and $\exists$, and nondeterministic function symbols can be interpreted deterministically. (Here, the "=" in a logical expression is being to denote equality interpreted as observable equivalence in the appropriate domain; this usage should cause no confusion as it obvious from the context whether equality is being interpreted as syntactic equivalence or observable equivalence.) We shall denote by $\underline{D}$ the function that computes the deterministic equivalent of a nondeterministic expression.

A little reflection will reveal that the only logical symbol that needs to be given special attention is =. We therefore first consider an expression of the form $t_1 =_T t_2$, where $t_1, t_2 \in W_T$.

If neither of t_1, t_2 involves any nondeterministic functions, then no translation is needed, since the expression is already deterministic; thus $\underline{D}(t_1 =_T t_2)$ is $t_1 =_T t_2$.

In the case that one or both sides of such an equation contain nondeterministic functions, we interpret $t_1 =_T t_2$ to mean that for every possible outcome r of t_1 there is a possible outcome of t_2 that is extraction equivalent to r, and vice versa. There are two possible ways to effect a formal translation of $t_1 =_T t_2$ that captures this interpretation. The first is to introduce an artifact that enables us to refer to the value r_i of a single application of a nondeterministic function $f(t_1,\ldots,t_n)$ at two or more places in an expression: note that this is distinct from writing several instances of the subterm $f(t_1,\ldots,t_n)$, as the nondeterministic nature of f then allows the various occurrences to evaluate to different values. An alternative way is to characterize the sets implied associated with nondeterministic expressions and reason in terms of these sets.

Using the definition of a characteristic predicate introduced in section 2, it is quite straightforward to detail a procedure that, given a term t, replaces all the occurrences of subterms t_i that involve nondeterministic functions by (existentially quantified) variables v_i, and tacks on a conjunct that states that the specific evaluations of the t_i resulted in the values v_i. For example, to represent the fact that CHOOSE(s) returns the value v_1, CHOOSE(s) may be converted into $[P_{CHOOSE}(s,v_1) \wedge \text{CHOOSE}(s)=v_1]$. We will denote this transformation by $\underline{S}$ (for Specify); note that $\underline{S}$ will introduce new variables into an expression.

We can now define $\underline{D}$ on $t_1 =_T t_2$ when t_1 or t_2 involve nondeterministic functions.

Let $\{\underset{\sim}{f}_1,\dots,\underset{\sim}{f}_n\}$, $\{\underset{\sim}{g}_1,\dots,\underset{\sim}{g}_m\}$ be the nondeterministic function symbols occurring in the terms t_1, t_2 respectively. Let $\{u_1,\dots,u_n\}$, $\{v_1,\dots,v_m\}$ be all of the new <u>variables</u> introduced when the nondeterministic function symbols in t_1, respectively t_2, are eliminated as above, i.e., when $\underline{S}$ is applied to t_1, t_2. Then, if $\underset{\sim}{f}_i$ is k_i-ary, g_i is q_i-ary, and $t\{t'|t''\}$ represents t with occurrences of t' substituted by t'', we have

$$\underline{D}(t_1 =_T t_2) \Rightarrow$$

$$[(\forall u_1,\dots,u_n)[P_{\underset{\sim}{f}_1}(s^1_1,\dots,s^1_{k_1},u_1) \wedge\dots\wedge P_{\underset{\sim}{f}_n}(s^n_1,\dots,s^n_{k_n},u_n)] \Rightarrow$$
$$(\exists v_1,\dots,v_m)[P_{\underset{\sim}{g}_1}(r^1_1,\dots,r^1_{q_1},v_1) \wedge\dots\wedge P_{\underset{\sim}{g}_m}(r^m_1,\dots,r^m_{q_m},v_m)]$$
$$\wedge\ (t_1\{\underset{\sim}{f}_i(s^i_1,\dots,s^i_{k_i}|u_i\}^{i=n}_{i=1} =_T t_2\{\underset{\sim}{g}_j(r^j_1,\dots,r^j_{q_j}|v_j\}^{j=m}_{j=1})]$$

$$\wedge$$

$$(\forall v_1,\dots,v_m)[P_{\underset{\sim}{g}_1}(r^1_1,\dots,r^1_{q_1},v_1) \wedge\dots\wedge P_{\underset{\sim}{g}_m}(r^m_1,\dots,r^m_{q_m},v_m) \Rightarrow$$
$$[(\exists u_1,\dots,u_n)[P_{\underset{\sim}{f}_1}(s^1_1,\dots,s^1_{k_1},u_1) \wedge\dots\wedge P_{\underset{\sim}{f}_n}(s^n_1,\dots,s^n_{k_n},u_n)]$$
$$\wedge\ (t_1\{\underset{\sim}{f}_i(s^i_1,\dots,s^i_{k_i}|u_i\}^{i=n}_{i=1} =_T t_2\{\underset{\sim}{g}_j(r^j_1,\dots,r^j_{q_j}|v_j\}^{j=m}_{j=1})]$$

As noted previously, the other logical symbols are quite transparent to $\underline{D}$:

$$\underline{D}(t_1 \wedge t_2) = \underline{D}(t_1) \wedge \underline{D}(t_2)$$
$$\underline{D}(t_1 \vee t_2) = \underline{D}(t_1) \vee \underline{D}(t_2)$$
$$\underline{D}(\neg t_1) = \neg\underline{D}(t_1)$$

<u>PROPOSITION 1</u>

$\underline{D}(t)$ is semantically equivalent to t. ∎

<u>Proof</u> Immediate by induction on the structure of a formula. Note that all logical symbols other than "=" are unaffected by $\underline{D}$.

The transformation $\underline{D}$ can always be used to deterministically interpret/translate all of the nondeterministic expressions dealt with in the rest of this paper. We will therefore not elaborate on such translations for the most part, but will instead phrase our arguments using the sets underlying nondeterministic terms. Our motivation in doing this is to convey greater intuition and to improve readability.

5. Implementations of Nondeterministic Operations

Intuitively, 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. (See figure 5-1 for an example.) That is, whenever extractors are applied to objects of TOI, yielding instances of known types, the corresponding computations in the implementation domain should yield equivalent results.

The presence of nondeterministic operations calls for a closer examination of the above interpretation, however, since we wish to allow implementations of nondeterministic functions that do not necessarily preserve the full extent of nondeterminism implied by the original specifications. In particular, we wish to allow for deterministic implementations of nondeterministic functions. This introduces a further nuance -- since there might be several distinct instances tt_1, ..., tt_n in the target type that represent the same instance of the type of interest, a deterministic implementation of a nondeterministic function in TOI might yield observably distinct results when applied to tt_1, ..., tt_n. For example, although <ASSIGN(ASSIGN(NEWARRAY,1,x),2,y),2> and <ASSIGN(ASSIGN(NEWARRAY,1,y),2,x), 2> represent "equivalent" sets {x,y} and {y,x}, the implementation of CHOOSE in figure 5-1 would return different results on the two representative instances. Such behavior is considered quite acceptable, and is fairly common in several implementations of nondeterministic functions. We term such implementations (or behavior) pseudo-nondeterministic to connote the fact a deterministic implementation evidences apparent nondeterminism. Finally, it is possible to have nondeterministic implementations of deterministic functions -- in such a case, the nondeterministic results produced by the implementation must obviously be all extraction equivalent.

We now formalize the notion of an implementation. We can do this with greater precision by introducing the notion of a (restricted) derivor [11]; this is done in Definition 4 below. However, we first need to introduce the notion of a term being viewed as a derived function: informally, a term "DELETE(INSERT(s,x),y)" can be viewed as an function (say INSERT-DELETE) with arity INSERT-DELETE:Set,Integer,Integer->Set, that maps the arguments (s,x,y) to the Set "DELETE(INSERT(s,x),y)". INSERT-DELETE is called a derived operation ("derived" from the term "DELETE(INSERT(s,x),y)" where s, x and y 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 function on all of the global sorts g in G. That is,

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

$$(\forall g \in G)\ [d_a(g) = g]$$

(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 function" must have arity

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

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 $d\text{-}\theta(f)$. 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 $\{d\text{-}\theta(f) \mid f \text{ in } F^{TOI}\}$ that is, the function corresponding to f is the term $\theta(f)$ viewed as a derived function. The equations of dTT are identical to those of TT. ■

Example If we consider the implementation of a Set in terms of an Indexed-Array (see Figure 5-1), the maps comprising the derivor are: d_a(Set) = Indexed-Array, d_a(Integer) = Integer, 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 5-1.

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

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

5.1. The Correctness of an Implementation

We are now in a position to formally state what constitutes an implementation that is consistent with the specifications of the TOI i.e. a "correct" implementation. However, in order to satisfactorily account for pseudo-nondeterministic implementations, we need to treat the implementations of deterministic functions and nondeterministic functions separately.

Intuitively, we view an implementation defined by θ to be correct iff the result (or set of results) yielded by the implementation of a (nondeterministic) function are result(s) that are admissible under the semantics of TOI, and if the implementation produces at least one such result whenever possible. Note that this implies that observable behavior is preserved by the implementations of deterministic functions.

All of the implementations θ, θ_1, and θ_2 described earlier (see figures 5-1, 5-2) are correct implementations of the type Set, although we do not give detailed proofs of this here. Definition 6 formally characterizes this notion of correctness.

We denote by $D\text{-}F^T$ the deterministic functions defined on T, and by $ND\text{-}F^T$ the nondeterministic functions defined on T.

The map θ defining an implementation of a Set using an Indexed Array is defined below. Let θ(s) = <a,i>. 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˙ 1 for monus). DATA(a,i) accesses the value previously ASSIGNed to the i-th element of the Array a; ASSIGN(a,i,x) simply "stores" the value x as the i-th element of a.

```
θ(EMPTYSET) = <NEWARRAY, ZERO>
θ(INSERT(s,x)) = if MEMBERTT(θ(s),θ(x))
                        then <a,i>
                        else <ASSIGN(a,SUCC(i),x), SUCC(i)>
θ(ISEMPTY(s)) = [i = ZERO]
θ(MEMBER(s,x)) = MEMBERTT(θ(s),θ(x))
where
MEMBERTT(<a,i>,x) = if i=ZERO
                    then FALSE
                    else if DATA(a,i)=x
                         then TRUE
                         else MEMBERTT(<a,PRED(i)>,x)
θ(CHOOSE(s)) = DATA(a,i)
```

θ is a <u>pseudo-nondeterministic</u> implementation of the nondeterministic function <u>CHOOSE</u>, i.e., a deterministic implementation that exhibits <u>apparent</u> nondeterminism.

```
θ(DELETE(s,x)) = if ~θ(MEMBER(s,x))
                 then s
                 else DELETETT(<a,i>,i,x)
where
DELETETT(<a,i>,j,x) = if DATA(a,j)=x
                      then COPY(<a,i>,j)
                      else DELETETT(<a,i>,PRED(j),x)
```

- DELETETT(<a,i>,j,x) first locates the index associated with the value x and then invokes COPY to actually "delete" this element from the Array a.

```
COPY(<a,i>,j) = if j=PRED(i)
                then <ASSIGN(a,j,DATA(a,i)),PRED(i)>
                else COPY(<ASSIGN(a,j,DATA(a,SUCC(j)),i>,SUCC(j))
```

- COPY(<a,i>,j) deletes the j-th element in the array a and shifts the values at j+1,...,i-th positions to j,...,i-1.

Figure 5-1: THE IMPLEMENTATION OF A SET USING AN INDEXED ARRAY

θ_1(CHOOSE(a,i)) = MAX(a,i)

- MAX(a,i) is function defined on the type Array that returns the maximum of the first i values in the Array a. θ_1 is a deterministic implementation of the nondeterministic function CHOOSE.

θ_2(CHOOSE(s)) = [x | x=DATA(a,i)   x=DATA(a,SUCC(ZERO))]

θ_2 is a nondeterministic implementation of CHOOSE that does not preserve the full extent of its nondeterminism.

Figure 5-2: ALTERNATIVE IMPLEMENTATIONS OF CHOOSE

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

$$(\forall g \in G)(\forall t_g \in W_g)\ [\{\theta(t_g)\} \subseteq_g \{t_g\}] \wedge [\{t_g\} \neq \emptyset \Rightarrow \{\theta(t_g)\} \neq \emptyset] \qquad \text{--(C)}$$

■

Thus, (C) implies that

$$(\forall g \in G)(\forall t_g \in W_g[D\text{-}F^{TOI}])[\theta(t_g) =_g t_g] \qquad \text{(i)}$$

i.e. that observable behavior is preserved by the implementations of deterministic functions, since in this case, $\{\theta(t_g)\}$ and $\{t_g\}$ both have a cardinality of 1.[4] The first conjunct in (C) is to preclude any erroneous values being returned by the implementation of a nondeterministic function, whereas the second conjunct ensures that at least one value is returned whenever possible. The above definition allows for nondeterministic implementations of deterministic functions: in such a case, it is required that every element of $\{\theta(t_g)\}$ be equivalent (in type g) to t_g.

It is important to note that the definition of correctness given above implies that any "information" that is contained in the elements in $\{t_g\} - \{\theta(t_g)\}$ is unimportant -- in that it is ignored -- unless it is manifest in a deterministic term. For example, if CHOOSE were the only extraction function defined on the type Set, and if it were the case that a pseudo-nondeterministic implementation θ(CHOOSE) always yielded the result 1 when applied to the sets INSERT(EMPTYSET,1) and INSERT(INSERT(EMPTYSET,1),2), then the above definition of correctness would imply that these two sets are equivalent. Usually, however, MEMBER is an integral part of the semantics specification of Set; in such a case MEMBER(s,2) is a deterministic term that yields distinct results when s is instantiated with the two sets in

[4]This cardinality could be ≤ 1 if partial functions are allowed.

question.

5.2. Degrees of Nondeterminism

Since we do not mandate that the implementation of a nondeterministic function preserve the full extent of nondeterminism implied by the specifications, it is possible to conceive of various implementations of the same type that differ in the extent to which they embody the nondeterminism implied by the initial specifications of the type.

Definition 7: If θ_1 and θ_2 are both correct implementations of a type, then we say that θ_1 is <u>more nondeterministic</u> than θ_2, denoted $\theta_1 > \theta_2$, if

$$(\forall g \in G)(\forall t \in W_g)\{\theta_1(t)\} \supseteq_g \{\theta_2(t)\}$$

where

$$\begin{aligned}&\{\theta_1(t)\} \supseteq_g \{\theta_2(t)\} \iff\\ &\quad [y \in \theta_2(t) \Rightarrow [\exists y' \mid y' \in \theta_1(t) \wedge (y' =_g y)]]\\ &\quad [\exists z \mid [z \in \theta_1(t)] \wedge [\nexists z' \mid z' \in \theta_2(t) \wedge (z' =_g z)]]\end{aligned}$$

■

This states that if θ_1 <u>consistently</u> produces more results than θ_2 does, then θ_1 is more nondeterministic than θ_2. Note that the quantification over the known types G ignores any expansion and contraction of the sizes of the sets of any intermediate computations; it is, however, possible to accommodate such an interpretation if desired. Also note that the above interpretation of $\{\theta_1(t)\}\ {}_g\{\theta_2(t)\}$ is different from

$$\{\theta_1(t)\} \supseteq_g \{\theta_2(t)\} \iff [y \in \theta_2(t) \Rightarrow y \in \theta_1(t)] \wedge [\exists z \mid z \in \theta_1(t) \wedge z \notin \theta_2(t)]$$

in that definition 7 precludes a superfluity of extraction equivalent values of type g from contributing to θ_1 being "more" nondeterministic than θ_2.

<u>Example</u> The implementation θ_2 in figure 5-2 is (strictly) more nondeterministic than θ defined in fig 5-1 i.e. $\theta_2 > \theta$. The implementation defined by θ_1 in figure 5-2 is incomparable with both the implementations θ and θ_2.

It is possible to define a weaker notion of the above ordering where only the cardinalities of the sets $\{\theta_1(t)\}$ and $\{\theta_2(t)\}$ are considered, but not their contents. That is, if θ_1 and θ_2 are both correct implementations of a type, then we may define $\theta_1 > \theta_2$, if

$$(\forall g \in G)(\forall t \in W_g)\ |\{\theta_1(t)\}| > |\{\theta_2(t)\}|.$$

5.3. Equivalence Classes Induced on the Representation Type

An implementation map θ serves to partition the terms in the derived algebra dTT (i.e. the representation type) into equivalence classes. The equivalence relations that arise most naturally in this context are:

- the extraction equivalence induced on terms in W_{TT} by the extraction functions E^{TT} defined on TT. We denote the equivalence classes induced by this relation by W_{TT}/E^{TT}, and its restriction to dTT by W_{dTT}/E^{TT}.
- the extraction equivalence induced on the "reachable" terms in the representation type i.e. the terms in dTT by the extraction functions E^{dTT}; the partition W_{dTT}/E^{dTT} induced by this relation on dTT is <u>coarser</u> than the partition W_{dTT}/E^{TT} because it merges those classes that cannot be distinguished by operations in F^{dTT} (but could be distinguished by operations in F^{TT}).

If F^{TOI} contains only deterministic functions, then W_{dTT}/E^{dTT} is the partition that is of primary interest to us [8], since its characteristics determine whether θ is a correct implementation. Specifically, if θ indeed defines a correct implementation then it can be shown that there exists a surjective homomorphism from W_{dTT}/E^{dTT} onto W_{TOI}/E^{TOI} [8]; this homomorphism is referred to as the <u>abstraction function</u> by Hoare in [5] and as the <u>rep</u> function in [12].

If, however, θ contains pseudo-nondeterministic implementations of nondeterministic functions, then the partitioning W_{dTT}/E^{dTT} may often be finer than we strictly desire, since extraction equivalence under pseudo-nondeterministic implementations will enable a distinction between terms in dTT that represent equivalent instances of TOI. As an example, the implementation θ of figure 5-1 enables a distinction between the representations of INSERT(INSERT(EMPTYSET,x),y) and INSERT(INSERT(EMPTYSET,y),x). We now define an equivalence relation E on dTT that serves to merge those instances of dTT that represent equivalent instances of the TOI; E induces <u>the</u> partition on dTT that we are interested in.

Definition 8: Let $\tilde{t}_1$, $\tilde{t}_2$ be terms in W_{TOI} which are pre-images under θ of the terms t_1, t_2 in W_{dTT}, i.e. $\theta(\tilde{t}_1)=t_1$ and $\theta(\tilde{t}_2)=t_2$.

Then $t_1 =_E t_2$ iff

$$\{\tilde{t}_1\} =_{TOI} \{\tilde{t}_2\} \qquad \text{--(A)}$$

i.e. $\tilde{t}_1$ and $\tilde{t}_2$ are equivalent in TOI, and

$$t_1 =_{D\text{-}dTT} t_2 \qquad \text{--(B)}$$

i.e. t_1 and t_2 are extraction equivalent if only the implementations of deterministic functions of TOI are considered, and

$$\{t_1\} \subseteq \{\tilde{t}_1\} \wedge \{\tilde{t}_1\} \neq \emptyset \Rightarrow \{t_1\} \neq \emptyset \text{ and } \{t_2\} \subseteq \{\tilde{t}_2\} \wedge \{\tilde{t}_2\} \neq \emptyset \Rightarrow \{t_2\} \neq \emptyset \text{--(C)}$$

i.e. the set of results $\{t_1\}$, $\{t_2\}$ yielded by implementations is a subset of those in TOI $\{\tilde{t}_1\}$, $\{\tilde{t}_2\}$ respectively, and at least on such result is yielded whenever possible. ∎

Intuitively, the partition induced by E on dTT is obtained by first merging those dTT terms that are extraction equivalent under the implementations of the deterministic functions in F^{TOI}, and then merging those terms that represent equivalent elements of TOI but may themselves be distinct (but are distinguishable only by the pseudo-nondeterministic implementations in θ).

It may be shown that our interpretation of a correct implementation coincides with one defining a surjective homomorphism from the extraction equivalence classes induced by E on dTT to the extraction equivalence classes of TOI; the homomorphism thus induced forms the nondeterministic counterpart of the rep function [12]. That is, a correct implementation map θ satisfying the conditions enumerated in 6 implies the existence of a surjective homomorphism ϕ

$$\phi : W_{dTT}/\ E \text{ -> } W_{TOI}\ /\ E^{TOI}.$$

It must be stressed that the remark following definition 6 is very important in this context: it is presumed that any information contained in $\{t_g\}-\{\theta(t_g)\}$ is irrelevant unless it is also mirrored in a deterministic form. If this assumption is not true, then θ in fact does not induce the above homomorphism.

6. Properties of Implementations of Nondeterministic Operations

We have seen how properties of nondeterministic functions can be expressed as logical expressions in the normal manner, provided equality in such expressions is interpreted appropriately. It is desirable that we have a criterion for deciding what it means for an implementation of a data type to preserve properties involving nondeterministic functions that might hold in the abstract specifications. This is not altogether straightforward: as we will show below, if a nondeterministic function $\tilde{f}$ has a pseudo-nondeterministic implementation, an equality of the form $t_1=t_2$ that is true in TOI may not be preserved in dTT by an otherwise "correct" implementation, since θ may return observably distinct values on t_1 and t_2.

Again, we need only investigate expressions involving = in detail. If t_1 and t_2 are deterministic, then the criterion for judging whether the property $t_1=t_2$ is preserved by the implementation θ is that the corresponding terms in the representation be extraction equivalent, i.e.

$$\theta(t_1) =_{dTT} \theta(t_2).$$

However, unless θ preserves all nondeterminism in TOI, the above need not necessarily be true if t_1, t_2 are nondeterministic. In general, it will the case that

$$\{\theta(t_1)\} \subseteq \{t_1\},$$

and

$$\{\theta(t_2)\} \subseteq \{t_2\}.$$

Thus, although $t_1 =_{TOI} t_2$ implies that $\{t_1\} =_{TOI} \{t_2\}$, $\{\theta(t_1)\}$ need not necessarily be equivalent to $\{\theta(t_2)\}$. We again adopt the interpretation that we chose in section 5.1 i.e. that $t_1 =_{TOI} t_2$ is preserved by θ iff the observable behaviors of $\theta(t_1)$ and $\theta(t_2)$ are equivalent under implementations of all of the deterministic operations in F^{TOI}, and all observable behavior in nondeterministic cases is allowed by the semantics of TOI.

7. Summary

We have delineated in the preceding sections ways to specify, implement, and reason about nondeterministic operations in the context of abstract (algebraic) data types. In essence, nondeterministic operations can be characterized by the possible outcomes when they are applied to a given set of parameters; the characteristic predicate of a nondeterministic function, defined in section 2, serves to circumscribe this set. In section 3 we formally characterized the externally observable behavior of a type that includes nondeterministic functions: intuitively, two terms were defined to be extraction equivalent if the sets of possible outcomes they can yield are equivalent. Expressions involving nondeterministic functions can be interpreted deterministically using the characteristic predicates of the nondeterministic functions: in section 4 we detailed such an interpretation. In section 5 we elaborated on three different kinds of implementations that are possible for nondeterministic functions: deterministic implementations, pseudo-nondeterministic implementations, and nondeterministic implementations. Nondeterministic implementations of nondeterministic functions can embody varying degrees of the full nondeterminism allowed by the semantics of a type; it is possible to order implementations of types by the extent to which they embody the original nondeterminism: two such orderings were considered in section 5.2.[5] In section 5.1 we developed criteria by which to judge the correctness of implementations of types that include nondeterministic functions, while in section 6 we elaborated on what it means for properties of nondeterministic functions to be preserved by "correct" implementations of a type.

The work described in this paper provides only a first step towards the understanding of nondeterminism in the context of abstract data types; several related issues remain to be investigated. Automated deduction systems (e.g. [7]) to aid in the proofs of implementations of nondeterministic functions need to be developed, and further experiments in specifying and proving nondeterministic types need to be conducted.

[5]In [10], we investigate implementations of functions defined on a type that consist of a number of co-operating processes which may execute in nondeterministic fashion.

REFERENCES

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

2. J.Goguen, J.Thatcher,E.Wagner. An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types. In R.Yeh, Ed., Current Trends in Programming Methodology, Vol IV, Prentice-Hall, N.J, 1979, pp. 80-149.

3. G.Gratzer. Universal Algebra. Van Nostrand, 1968.

4. J.V.Guttag. The Specification and Application to Programming of Abstract Data Types. Ph.D. Th., Computational Sciences Group, University of Toronto, 1975.

5. C.A.R.Hoare. "Proof of Correctness of Data Representations." Acta Informatica 1 (1972), 271-281.

6. B.Liskov, S.Zilles. "Specification Techniques for Data Abstractions." IEEE Trans. on Soft. Engg. SE-1 (1975), 7-19.

7. David R. Musser. A Data Type Verification System Based on Rewrite Rules. USC/Information Sciences Institute, October, 1977.

8. P.A.Subrahmanyam. On Proving the Correctness of Data Type Implementations. Dept. of Computer Science, University of Utah, September, 1979.

9. P.A.Subrahmanyam. A New Method for Specifying and Handling Exceptions. Dept. of Computer Science, University of Utah, January 1980.

10. P.A.Subrahmanyam. Cooperating Nondeterministic Processes and Nondeterministically Cooperating Processes. Dept. of Computer Science, University of Utah, 1980, forthcoming.

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

12. W.A.Wulf, R.L.London, M.Shaw. Abstraction and Verification in ALPHARD. CMU, ISI, August, 1976.