LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Expressiveness of the Operation Set of a Data Abstraction

Computation Structures Group Memo 179-1
June 1979
Revised January 1980

Deepak Kapur
Srivas Mandayam

A condensed version of this paper was presented at the Seventh ACM Symposium on Principles of Programming Languages.

This research was supported in part by the Advanced Reserach Projects Agency of the Department of Defense, monitored the by the Office of Naval Research under contract N00014-C-0661, and in part by the National Science Foundation under grant MCS 74-21892 A01.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

1. Introduction

An important feature of a data abstraction is the constraint that the values of the abstraction can be constructed and observed only by the applications of its operations. This feature, by decoupling the use of a data abstraction from its implementation, supports modular development of software [5, 10, 17]. It also aids program verification by helping to decompose proofs into small and independent units [5, 4]. However, the very same feature can make the use of a data abstraction restrictive if the operation set of the abstraction is not designed carefully. This is especially so in a programming environment where a programmer is encouraged to build on the abstractions provided by other programmers. If the operation set of an abstraction is not expressive enough to discover all the properties of the values of the abstraction, it might be impossible or inconvenient to implement several useful functions on the values.

Consider the following design of the familiar immutable [10] **set [N]** abstraction where N stands for the abstraction *natural number.* (The definition of **set [N]** is given in Appendix I. An algebraic specification of it appears in Appendix II.)

(1) set [N] is null, insert, remove, has, empty, max, equal

null returns the empty set; **insert** returns the set obtained by inserting a given number into a given set; **remove** returns the set which is obtained by removing a given number from a given set; **has** tests the membership of a given number in a given set; **empty** tests if a given set is empty; **max** returns the largest number of a given set; **equal** tests if its two given sets are identical. With a little bit of thought it is possible to convince oneself that the above set of operations is adequate to discover all the properties of **set [N]** values.

Now, consider the following two alternative designs of **set [N]** which are obtained by dropping some of the operations from the above design. The operations common among the three designs have the same meaning. Note that all three designs have the same value set, which is the set generated by **null** and **insert**, since all the

values are distinguishable by the operations in each of the cases.

(2) set [N] is null, insert, has

(3) set [N] is null, insert, equal

In the second design the operation set is not expressive enough to discover all properties of the set values; for instance, it is impossible to implement the **remove** operation in terms of the given operations. (Note that termination becomes a problem for any algorithm that attempts to construct the required set by using the **has** operation to include all the elements except the one that is being removed.) The third design can be shown to be logically as powerful as the first one, but in this case, the implementations of even simple functions not provided as primitives, such as **remove**, turn out to be so inconvenient and unnatural that the design is uninteresting from a practical point of view.

How can one characterize this notion of expressiveness of the operation set of a data abstraction ? How does one determine if the operations of a data abstraction form a fully expressive set or not ? Can one distinguish between the expressiveness of the operation sets of different designs of the same data abstraction, such as designs (1) and (3) ? Morris [13] is the first one to have posed some of these questions. He proposed that "a new data type should be *transferable* in the sense that the primitives (i.e., the operations) are adequate to translate between the new type and any other existing type, such as integers." He introduced *write* and *read* transfer functions from the new type to integers. The difficulty of implementing the primitive operations of the new type in terms of the existing type using the transfer functions gives an indication of how expressive the operations of the new type are. His characterization is very informal, and it does not distinguish between situations such as design (1) and design (3) in the above example.

In this paper, we formally characterize what it means to say the operation set of a data abstraction is fully expressive. We define two properties of a data abstraction

related to the expressiveness of its operation set - *expressive completeness*[1] and *expressive richness*. The second property is stronger than the first. We will be able to distinguish among the three designs of set [N] mentioned earlier using the above two properties. We will see that the second design is not even expressively complete, while designs (1) and (3) are. However, design (1) is expressively rich, but design (3) is not.

The purpose of introducing the notion of a design is to compare the expressiveness of subsets of the operation set of a data abstraction. A design of a data abstraction is defined so that it has the same behavior of the data abstraction, while its operation set is a subset of that of the data abstraction. The properties of expressiveness discussed in the paper are, strictly speaking, the properties of a particular design of a data abstraction. However, at several places in the paper, the reader may find the properties being associated with a data abstraction. When we do this, we actually mean the expressiveness of the design whose operation set is identical with that of the data abstraction.

1.1 An Extended Overview

In the next chapter, we discuss a few preliminary concepts and state the assumptions made in the paper about the behavior of a data abstraction. We precisely define the value set of a data abstraction, and describe what we mean by different designs of a data abstraction.

In the third chapter, we formalize the notion of expressive completeness. We wish that the operation set of an expressively complete data abstraction allow us to discover all interesting properties of the values. Since programmers are going to be interested only in computable properties, we require that the operation set of an expressively complete abstraction be expressive enough to implement all computable functions over the value set of the data abstraction. We specify what it means for "a

---

1. This notion of completeness should not be confused with the completeness property of a specification of a data abstraction. The properties discussed in the paper are properties of data abstractions, and are independent of the specification technique used to specify data abstractions.

function to be implementable" by stipulating the mechanisms from which the function ought to be built.

The remaining step in the process of formalization, therefore, is to define computability and computable functions over the value set of a data abstraction. We do this by reducing the computability over an abstract domain to computability over the set of natural numbers. For this we use a scheme to encode the values of the data abstraction as natural numbers. Every function on the abstract value set can then be mapped to a corresponding function on natural numbers. A function over the abstract value set is considered to be computable if its image function is computable over the set of natural numbers.

The main result of this chapter is **Theorem 2**, which states that if the operation set of a data abstraction **D** consists of computable operations only, and the **equal** predicate on **D**, which computes the identity relation on the value set of **D**, can be implemented in terms of the operation set, then **D** is expressively complete. The theorem enables us to define a minimal set of operations that makes an abstraction expressively complete.

In the fourth chapter we introduce the notion of *expressive richness* of the operation set of a data abstraction. The motivation for this arises because of the existence of several expressively complete data abstractions whose operation sets are not versatile enough to be of any practical use. We wish that an expressively rich data abstraction be expressively complete with an operation set that is rich enough to extract all relevant information from a value conveniently.

To formalize this notion of richness, we identify a set of functions, called *distinguished functions*, associated with every abstraction. The distinguished functions permit one to extract from any given value, all information required to reconstruct it. There are two kinds of information that one needs to know to construct a value. Firstly, we need to have a set of values of other types which is used in the construction of the value. Secondly, we need to know the constructors and the order in which they should be used in creating the value. We define two kinds of distinguished functions - the *d-functions* and the *p-functions* - to extract the above two pieces of information,

respectively.

An expressively rich data abstraction is defined as one in which every distinguished function can be implemented in terms of the operation set using only composition and conditional expressions. We believe that if the distinguished functions can be conveniently implemented in terms of the operation set, then so can most of the other useful functions on the data abstraction. We show that a data abstraction that is expressively rich has the desired logical power by proving that it is also expressively complete.

The final chapter discusses a few practical applications of data abstractions that are expressively rich according to our characterization.

2. Assumptions and Preliminary Concepts

2.1 Assumptions

Informally speaking, we view a data abstraction (abstract data type, data type, type) D as consisting of a set of values, and a finite set of operations to create and manipulate those values. In this paper, we consider only immutable [10] data types. We denote the operation set of D by Ω. Normally, the definition of D involves other data types; these data types appear as the domains and/or range of the operations of D. We call these types the *defining types* of D, and denote their collection by Δ. We refer to D itself as the *defined type.*

We assume that every operation in Ω yields exactly one value. The operations that yield values of type D are called *constructors* ; the remaining operations are called *observers.* Constructors which do not take any arguments of the defined type are called *basic constructors* ; the remaining constructors are called *non-basic constructors.*

We assume that every operation of D is total and does not signal any exception [11]. This assumption is made only for convenience, since the properties proposed in the paper can be extended to handle exceptions once a suitable model for characterizing the exceptional behavior of a data type is adopted. In the examples discussed in the paper, we have arbitrarily decided on some normal behavior for an

operation on certain inputs on which the operation would otherwise have signalled an exception.

2.2 Definition of a Data Abstraction

Heterogeneous algebras are a natural way to model the behavior of a data type [18, 2, 3]. A heterogeneous algebra for a data type **D** consists of (i) a domain corresponding to D and a domain corresponding to every defining type in **Δ**, and (ii) a function corresponding to every operation in **Ω**.

We take a *behavioral* view for defining the semantics of a data type, first advocated by Guttag [3], and later developed by Kapur [8]. According to this view, every value of **D** is created by finitely many applications of the constructors of **D**, and the values are distinguishable only by means of the operations of **D**. A data type is defined as a set of *behaviorally equivalent* heterogeneous algebras.[2] Every algebra in the set is called a *model* of the data type. Two algebras are, informally speaking, behaviorally equivalent if they have the same observable behavior as expressed by their observers. The domain corresponding to **D** in a model defines a value set of **D**. In the context of a model, by a value of **D** we mean an element of the value set.

The observable behavior of a model is characterized in terms of the *distinguishability* relation on values. The distinguishability relation is defined inductively in terms of the distinguishability of the values of the defining types. (The basis of this induction is the data type **bool** that does not have any defining types; the only two values, **true** and **false**, of **bool** are assumed to be distinguishable.) Two values of a model are distinguishable if and only if there is a sequence of operations of **D** with an observer as the outermost operation, that produces distinguishable results when applied separately on the values. If two values are not distinguishable, they are *observably*

---

2. This view is a further abstraction on the view of ADJ [2] and Zilles [18] which merely abstracts from the representations of the values in an algebra: a data type in their view is defined as a set of isomorphic heterogeneous algebras. The behavioral view is closer to the view taken in programming languages supporting data types.

equivalent. Observable equivalence is an equivalence relation, and hence can be used to define a quotient set (a set of equivalence classes) on every domain of a model; furthermore, the observable equivalence relations are preserved by the functions in the model. Two models are behaviorally equivalent if the quotient models induced by the observable equivalence relations are isomorphic to each other.

We can specify the behavior of a data type by presenting a model for it. Data types used as examples in the paper are all defined in Appendix I in this fashion.

2.3 A Standard Value Set

In this paper, when we discuss an arbitrary data type D, we use a standard model for it whose value set is constructed in terms of sequences of constructors of D. The advantage in using this value set is that its method of construction is generally applicable and is also well suited to the formalization of computability on data types. The construction of this value set is explained below.

Every sequence (composition) of constructors of D creates a value of D, and hence can be used to denote that value. We call a sequence of constructors a *word*; several different words may create observably equivalent values. Let W_D denote the set of all finite words of D (we drop the index whenever it is evident from the context). W is constructed inductively by assuming that the word set is given for each of the defining types; a data type with no defining types serves as the basis. Let E stand for the observable equivalence relation on W. We use the quotient set W/E as the standard value set[3] (also denoted as V). Except in case of the models given in Appendix I, whenever we refer to the value set of D, we mean its standard value set.

By a function of D, we mean a function on the word set W that preserves the equivalence relation E. So a function f on W that preserves E can be viewed as a

---

3. In ADJ and Zilles's view, the distinguishability of values is not defined in terms of the behavior of the operations, instead an equivalence relation on words is independently defined; if two words are not related, they are distinguishable irrespective of whether they can be distinguished by the operations in the behavioral sense. The discussion and the results of the paper can be applied to this view as well by using the given equivalence relation in place of the observable equivalence relation E.

function f' on V (= W/E) such that f'([w]) = [f(w)], where [w] is the equivalence class containing w. ([w] is a value of D.) The same view can be extended to functions with several arguments. For example, the equivalence relation **E : W X W -> bool** can be viewed as the equality predicate on V that tests whether two values are identical. Henceforth, we denote the equality predicate on **D** by **equal**, or **=**.

2.4 Different Designs of a Data Type

Given a definition of **D**, a design of **D** is a variant of **D** whose operation set Ω' is a subset of Ω; the operations in Ω' should be such that they can create and distinguish all the values in W/E. We will present a design by giving its operation set. For example, for **set[N]** with Ω = **{null, insert, remove, has, empty, max, equal}**, the following are some of the possible designs.

Ω is itself a design

Ω_1 = **{null, insert, has}**

Ω_2 = **{null, insert, equal}**

However, Ω_3 = **{null, insert, remove, max}** is not a design because it cannot distinguish all the values of W/E. For instance, two different sets with identical maximal elements are not distinguishable by Ω_3.

The idea behind the definition of the design of a data type is to capture the evolutionary nature of the designing process of the data type. When a designer designs a data type, he normally starts out by visualizing a value set for it and a set of constructors to generate the values; then he starts designing other operations. Since operations are the only means of creating and observing the values, the designer at the least needs to provide enough operations to distinguish among the values. Later, with increasing experience, the designer enriches the operation set by adding more operations. All along his initial conception of the value set remains unchanged. We study the expressiveness of various operation sets by comparing the expressiveness property of different designs of a data type.

As pointed out earlier, the properties of expressiveness are properties of a

particular design of a data type. At several places in the paper, when we are not comparing the expressiveness of subsets of the operation set of a data type, we simply associate the property with the data type and make references, such as "an expressively complete data type", etc. When we do this we actually mean the expressiveness of the design whose operation set includes all the operations of the data type.

3. Expressive Completeness

In this chapter, we discuss the expressive completeness of the operation set of a data type. In the first section, we formally define *expressive completeness* in terms of computability over the value set of a data type. In the second section, we characterize computability over an abstract domain by reducing it to computability over the set of natural numbers. The third section proves a useful result about expressive completeness. The last section illustrates the definitions and results by discussing several examples.

3.1 Definition of Expressive completeness

We wish an expressively complete data type **D** to permit all computable functions of **D** to be implementable in terms of its operations.

Definition 0 A function is *implementable* in terms of a set of functions **F** iff its definition can be written using the mechanisms of functional composition, conditional expression, and recursion in terms of functions in **F** and a set of auxiliary functions, if any, also implementable in terms of **F**.

Let $\mathfrak{G}(F)$ denote the collection of functions implementable in terms of **F**.

It is important to notice a subtle difference in the method of definition for functions used by us and the one widely used in the literature on computability. In the latter, functions are defined using a scheme that assumes a pattern matching mechanism to reveal the top level structure of the argument value. For example, the addition function, denoted by +, on **N** is defined as :

$$x + 0 \triangleq x \qquad (*)$$

$x + S(y) \triangleq S(x + y)$

We instead take an abstract view, which indeed is the view taken by a programming system supporting data types. The internal structure of a value is not explicit; any information about a value must be obtained through its operations. Every operation we use has to be either an explicit part of the operation set of D, or be implementable in terms of the operation set. In our approach, for example, + would be implemented in a programming-language-like manner as :

$+(x, y) \triangleq$ if $y = 0$ then x else $S(+(x, p(y)))$, where

$p(x) \quad \triangleq$ if $x = 0$ then 0 else $p'(x, 0)$

$p'(x, z) \triangleq$ if $x = S(z)$ then z else $p'(x, S(z))$

Notice that the above implementation of + is given in terms of **0, S, =,** and **p**; **p** and **p'** are defined as an auxiliary functions in terms of **0, S,** and **=**. However, it seems that the definition (*) of + above did not need **=** and **p**. In fact, these functions are implicit in the pattern matching mechanism used by the definition (*).

Let Comp(D) denote the class of all computable functions on the value set of D. (Computability on D is defined in the next section.) Then we have

Definition 1 A data type D is expressively complete if $\mathbf{Comp(D)} = \mathfrak{G}(\Omega)$

For example, it is known that $\mathbf{Comp(N)} = \mathfrak{G}(\{0, S, =\})$ [9, 12]. Thus

Corollary 1 The data type **N** whose operation set $\Omega = \{ 0, S, = \}$ is expressively complete.

Using the definition of computability over **s_expression** (of **nils**) [12], we also get the following :

Corollary 2 The data type **s_expression** whose operation set
$\Omega = \{$ **nil, car, cdr, cons, null?** $\}$ is expressively complete.

We were able to obtain the results in the corollaries readily because computability over **N** and **s_expression** has been formally characterized in the literature. However, the notion of computability has not been formally characterized

for an arbitrary data type D. The next section addresses this issue.

3.2 Computability over Abstract Domains

If the value set V is finite, the notion of computability is trivial since every function on D can be specified as a finite table, and is thus computable. Below, we shall concentrate on a D whose V is infinite.

Our strategy is to reduce computability on D to computability over natural numbers [14]. We define computability on W; a computable function on W that preserves E is a computable function on D.

We define computability on W by means of an effective bijective function η : W -> N that encodes every word into a natural number. We call η an *encoding* function for D; and η^{-1} a *decoding* function for D. Every function on W can be viewed via η as a function on N. An η for D is constructed inductively using an η for each of its defining types. The basis of this induction is a data type having no defining types for which an η can be constructed in a straight forward manner.

Definition 2 Given a function f : W -> W, its *image* $\overline{f}$: N -> N via η is defined as $\overline{f}(n) = \eta(f(\eta^{-1}(n)))$.

The concept of an image of a function on D can be extended to functions defined over several domains by similarly making use of an encoding-decoding scheme for each of the domains involved in the definition. Using Church's thesis, we have

Definition 3 A function f : W -> W is *computable* if there exists an encoding η such that the image $\overline{f}$ of f via η is computable over N.

The above definition of computability on W has the following desirable property.

Theorem 1 The set of all computable functions on W is invariant with respect to the encoding function chosen.

Proof We show that any two different encoding functions define the same set of computable functions on W. We show that for every pair of encodings η_1 and η_2 there exist computable functions T_{12} and T_{21} on N which behave as follows: T_{12} maps the

code of a word associated by η_1 to that associated by η_2; T_{21} does the mapping in the opposite direction. It can be shown that the image of a function f via η_1 can be constructed from the image of f via η_2 and vice versa using T_{12} and T_{21}.

Definition 4 A function f of D is *computable* if f on W is computable and f preserves the equivalence relation E.

Weihrauch [16] defines computability on V directly; his abstract computability thesis can be shown to be equivalent to our definition of computability. We believe our approach is simpler and more natural.

Henceforth, we concern ourselves only with data types having computable operations, as the ones with noncomputable operations are of no practical interest. Furthermore, if a data type has noncomputable operations, it cannot be expressively complete. Since the word set of a data type is always recursive, having all the operations computable makes the value set recursively enumerable.[4]

3.3 A Useful Result

Here, we prove a useful result about the expressive completeness of the operation set of a subclass of data types for which **equal** is decidable.

Theorem 2 Assuming that (i) for D and for each of its defining types the **equal** predicate is decidable, and (ii) every defining type of D is expressively complete, D is expressively complete if the **equal** predicate on D is implementable in terms of its operation set Ω.

The proof of the above theorem follows immediately from the following Theorem, which states that if **equal** is an operation of D, then D is expressively complete. This is so because if **equal** $\in \mathfrak{C}(\Omega)$, then $\mathfrak{C}(\Omega) = \mathfrak{C}(\Omega \cup \{ \textbf{equal} \}) = \textbf{Comp(D)}$.

---

4. A set S is *recursive* if its characteristic function, which checks whether an arbitrary element is in S or not, is total computable. S is *recursively enumerable* (r.e.) if it is the range of a total computable function. In other words, an r.e. set S can be generated by a total computable function.

Theorem 3 Assuming that (i) for each of the defining types of **D** the **equal** predicate is decidable, and (ii) every defining type of **D** is expressively complete, **D** is expressively complete if the operation set of **D** includes the **equal** predicate on **D.**

Proof See Appendix III.

Theorem 3 can be used to show the following :

Corollary 3 A minimal set of operations that makes a data type **D** expressively complete is $\Omega = \Omega_c \cup \{ \textbf{equal} \}$, where Ω_c is a minimal set of constructors sufficient to generate the value set of **D.**

3.4 Examples

In the following examples we investigate the expressive completeness of different designs of **stack [N]** and **set [N]** data types. (See Appendix-I for definitions of these data types.) The operation set of each of them includes only computable operations. This can be inferred by intuition, as well as shown formally using the method discussed above. In each of the following cases, we just need to see if it is possible to implement the **equal** predicate for the type or not.

The following designs are expressively complete.

```
(1) stack [N] is null, push, pop, top, empty
    equal(v1, v2) ≜ if empty(v1) then empty(v2)
                    else if empty(v2) then false
                         else (top(v1) = top(v2) &
                              equal(pop(v1), pop(v2)))

(2) set [N] is null, insert, remove, has, empty
    equal(v1, v2) ≜ check_in_order(v1, v2, 0)
```

```
check_in_order(v1, v2, i) ≙
    if empty(v1) then empty(v2)
    else if empty(v2) then false
    else if (has(v1, i) = has(v2, i))
        then check_in_order(remove(v1, i),
                            remove(v2, i),
                            i + 1)
        else false
```

The following designs are not expressively complete because **equal** is not implementable in terms of their operations. In both the following cases it is possible to apply the above algorithm to find an answer when the two sets are unequal; however, when the sets are equal, the algorithm does not terminate because of the absence of the operation **empty** in the first case, and the absence of the operation **remove** in the second case.

(3) set [N] is null, insert, remove, has
(4) set [N] is null, insert, has, empty

4. Expressive Richness

In this chapter we introduce the notion of expressive richness. The motivation for this stems from the existence of several expressively complete data types whose operation sets are not rich enough to be of any practical use. For such types, implementation of even simple and useful functions that are not provided as operations of the types can turn out to be extremely tedious and unnatural. For instance, consider the following design of **set [N]**.

set [N] is null, insert, remove, has, empty

This design has been widely used in the literature. It is an expressively complete design, as was shown in section 3.4 above, since the **equal** predicate is implementable in terms of the operations. Note that the implementation of **equal** required enumeration of **N** upto the minimum of the maximal elements of the two sets. The following is an implementation of the function **size** that computes the size of a set. This implementation also needs an enumeration of **N**. The auxiliary function **count** in the implementation does this job; every time it finds a number that belongs to the set it

removes the number from the set, and increments the count (accumulated in the variable cnt) by 1.

```
size(v) ≜ count(v, 0, 0)
count(v, i, cnt) ≜ if empty(v) then cnt
                   else if has(v, i)
                        then count(remove(v,i), i+1,cnt+1)
                        else count(v, i+1, cnt)
```

The enumeration in the above example would be more complicated if the elements of the set were of an arbitrary type that does not have a natural ordering defined on it. In that case we would need to encode the values of the element type to perform the enumeration.

The enumeration was necessary in the above implementation because the data type set [N] does not provide any operation to pick an element of a set conveniently. In general, this problem arises when the operation set is not rich enough to extract all relevant information from a value conveniently. We would like an expressively rich data type to avoid the need for such enumeration while extracting all relevant information from a value. To characterize this aspect of richness of the operation set, we first introduce the concept of distinguished functions. We later formulate a definition for expressive richness in terms of them.

4.1 Distinguished Functions

The *distinguished functions* for every data type D are defined corresponding to a minimal subset Ω_c of constructors of D that can generate the whole value set of D. A family of distinguished functions defines a set of manipulations sufficient to extract all relevant information from a given value, necessary to reconstruct the value back from scratch.

In general, there can be more than one minimal subset of constructors, i.e., Ω_c, for D. So D can have more than one family of distinguished functions, with every family being associated with a particular Ω_c. Furthermore, D can have more than one family of distinguished functions corresponding to the same Ω_c. We will illustrate this

point by means of an example later.

There are two kinds of information that one needs to know to construct a value. Firstly, we need to have a set of values of the defining types which is used in the construction of the value. Secondly, we need to know the constructors, and the order in which they should be used in creating the value. Based on which of the above two kinds of information they extract, the distinguished functions are classified into two kinds - the *d-functions* and the *p-functions.*

4.1.1 The d-functions

The d-functions permit us to extract from an arbitrary value of **D**, a set of values of the defining types necessary to reconstruct the value using the constructors in Ω_c. The d-functions are such that every such value of the defining type can be extracted by means of a finite composition of the d-functions.

4.1.1.1 Definition

Let σ_i be an n-ary ($n > 0$) operation of **D** in Ω_c such that $\sigma_i : D_1 X \ldots X D_n \rightarrow D$. $D_1, \ldots, D_n$ are either **D** or the defining types of **D**. Associated with σ_i there are **n** d-functions, $d_i^1, \ldots, d_i^n$, such that for every $1 \leq k \leq n$, $d_i^k : D \rightarrow D_k$ and d_i^k satisfies the following property:

(P1) For every value of **D**, there exists a family of finite compositions of d_i^k's, denoted by $\mathfrak{S} = (S_1, \ldots, S_M)$, where every S_j is of the form $S_j = d_{i_1}^{k_1} \cdot \ldots \cdot d_{i_t}^{k_t}$, such that

(i) $S_j(v) = a_j$ is a value of one of the defining types of **D**, and

(ii) it is possible to construct **v** from $\{a_1, \ldots, a_M\}$ using the constructors of Ω_c.

If σ_i is 0-ary, then it does not have any associated d-functions.

4.1.1.2 Explanation

To keep the exposition simple, we first explain the definition of d-functions for the simple case of a data type where every constructor in its Ω_c takes at most one argument of a defined type. (We discuss the general case in a subsequent section.)

In the simple case, every value of the defined type can be constructed using a sequence of constructors from Ω_c; we call such a sequence a *constructing sequence* for the value. The first constructor in every constructing sequence is a basic constructor, and the rest of them are all non-basic constructors. Every constructor in the sequence uses a set of values of the defining types as arguments to it. For instance, consider the following design of **stack [N]**.

```
stack [N] is null, push, pop, top, empty
```

For stack [N], Ω_c = {**null, push**}. A stack value with **n** elements has the (unique) constructing sequence that has the form $(\textbf{push})^n \cdot \textbf{null}$. ($(\textbf{push})^n$ denotes a sequence of **n push** operations.)

The d-functions associated with a constructor σ_i can be viewed as functions acting as "inverses" for σ_i; they can be considered as functions that "undo" the effects of a particular instance of the constructor σ_i in a constructing sequence for the given value. The d-functions for every constructor are designed by fixing the instance of the constructor (in the constructing sequence) one wishes to undo by a single application of the d-functions. The design should be such that all instances of the constructor are undone after a finitely many applications of the d-functions on the value.

For **stack [N]**, there are no d-functions associated with **null**, since **null** is a zero-ary constructor. The operation **push** has two associated d-functions - $\mathbf{d}^1_{\mathbf{push}}$ and $\mathbf{d}^2_{\mathbf{push}}$. By selecting the d-functions to undo the effects of the latest instance of **push** in the constructing sequence, we require the two d-functions to satisfy the following properties.

$$\mathbf{d}^1_{\mathbf{push}}(\mathbf{push(v, e)}) = \mathbf{v}$$

$$\mathbf{d}^2_{\mathbf{push}}(\mathbf{push(v, e)}) = \mathbf{e}$$

Note that **pop** and **top** satisfy these properties, and hence can be used as the d-functions.

To see how the d-functions can be used to extract all the values of the defining types, we classify the d-functions into *decomposers* and *extractors*: a decomposer yields a value of the defined type, and an extractor yields a value of a defining type. In the case of **stack [N]**, $\mathbf{d}^1_{\mathbf{push}}$ is the decomposer and $\mathbf{d}^2_{\mathbf{push}}$ is the extractor for **push**.

To extract the values of the defining types we go through a reverse process using the decomposers and extractors instead of the constructors. We first find a sequence of decomposers to "decompose" the given value to a basic value (i.e., a value constructed by a basic constructor in Ω_c); we call such a sequence a *decomposing sequence* for the value. A decomposing sequence can be easily derived from a constructing sequence for the value: it is the reverse of the sequence of decomposers that is obtained by substituting every constructor in the constructing sequence by its corresponding decomposer. Every initial subsegment of the decomposing sequence (when applied on the value under question) yields a value of the defined type that was generated at some point during the construction of the value under question. The values of the defining types are obtained by applying appropriate extractors to each of these values. For instance, for **stack [N]**, there is only one way of decomposing a stack value using **pop**; the decomposing sequence for a stack of depth **n** is $\mathbf{pop}^{\mathbf{n}}$. The following set of sequences of d-functions extracts all the numbers used in the construction of the stack :

$\{\ \mathbf{top}\ ,\ \mathbf{top}\cdot\mathbf{pop},\ \ldots\ ,\ \mathbf{top}\cdot(\mathbf{pop})^{n-1}\}$.

The set of d-functions associated with a particular Ω_c need not be unique. For instance, for **stack [N]**, a function **remove_first** that removes the last element of a stack, and a function **get_first** that fetches the deepest element of a stack could just as well have acted as d-functions for **push**. Notice that these d-functions undo the effect of the earliest instance of **push** in the constructing sequence.

4.1.1.3 The File Example

We further elucidate the definition of d-functions by designing a set of d-functions for a reasonably sophisticated **file [t]** example. The definition of **file [t]** appears in Appendix I; the following is an informal description of it.

file [t] is null, insert, rewind, skip, delete, read, pos, empty, eof, length

A file value can be considered to be a sequence of records with an imaginary pointer, where every record is of type t. The pointer could be pointing to one of the records in the file, or to an imaginary position, called *end_of_file*, beyond the last record in the file; the pointer in an empty file is always at the *end_of_file*. **skip** moves the pointer forward by a specified number of records. **rewind** resets the pointer to the first record. **insert** inserts a record into the file immediately before the record to which the pointer is pointing (it leaves the pointer pointing to the same record.); if the pointer is at the *end_of_file*, the record is inserted at the end. **delete** deletes the record (if any) pointed to by the pointer. **read** fetches the record pointed to by the pointer. **pos** returns the current position of the imaginary pointer. **empty** and **eof** are predicates which have the behavior naturally implied by their name. **length** returns the number of records in the file.

For **file [t]**, Ω_c = { **null, insert, rewind, skip** }. In this case, it is not very obvious what the d-functions can be. The difficulty arises because the constructors in Ω_c can, in general, be used in several different ways to create a file value. (In contrast, for **stack [N]**, there is exactly one constructing sequence for every value.) In such cases it is useful to select a canonical constructing sequence for the values, and then design d-functions to undo the effects of specific instances of the constructors inside the canonical sequence. Every non-empty file value with **n** records can always be constructed by a constructing sequence of the form **skip** · **rewind** · **(insert)^n** · **null**. The integer argument to **skip** is the position of the pointer in the file. Based on this canonical sequence, we propose the following design for the d-functions.

(i) The d-functions corresponding to **skip** are chosen to undo the effects of the single

instance of **skip** in the canonical sequence. So the d-functions have to satisfy the following properties.

$$d^1_{skip}(skip(rewind(v), n)) = rewind(v)$$
$$d^2_{skip}(skip(rewind(v), n)) = n$$

It is easy to see that the operations **rewind** and **pos** satisfy the above properties; hence they can serve as the d-functions.

(ii) Assuming that the effects of **skip** are already undone, the d-function of **rewind** can expect to receive only values that have a constructing sequence of the form **rewind** · **insert**^{n}, i.e., the file is rewound. The operation **rewind** is many-to-one; it resets the pointer no matter where the pointer was (even if the file is already reset) prior to its application. d^1_{rewind} should be designed such that it acts as an inverse of **rewind**. We choose to design it as an identity function since that leads to a more natural set of distinguished functions. So we have the following trivial definition for d^1_{rewind}.

$$d^1_{rewind}(rewind(v)) = rewind(v)$$

(iii) Assuming that the effects of **skip** and **rewind** are already undone as explained above, the d-functions of **insert** can expect only values that have a constructing sequence of the form **rewind** · **insert**^{n}. We choose the d-functions to undo the effects of the rightmost instance of **insert** in the above constructing sequence. Informally, we want d-functions for **insert** to behave as follows:

$$d^1_{insert}(rewind(insert^{n-1}(insert(null, r1),...rn))) =$$
$$rewind(insert^{n-1}(null, r2), ... rn))$$
$$d^2_{insert}(rewind(insert^{n-1}(insert(null, r1), ... rn))) = r1$$

The above properties can be expressed algebraically as follows :

$$d^1_{insert}(rewind(insert(rewind(v), r))) = rewind(v)$$
$$d^2_{insert}(rewind(insert(rewind(v), r))) = r$$

The operations **delete** and **read** can serve as the two d-functions, respectively.

The decomposers we have chosen for the **file [t]** type are such that there are, in general, several decomposing sequences for a file value. For instance, since d^1_{rewind} is an identity function, it can be applied an arbitrary number of times at any point in the decomposition. However, the canonical form we have chosen suggests an obvious decomposing sequence. This is to undo the effects of the constructors in the order in which they appear in the canonical form; this strategy will decompose a file starting from the leftmost record. Since d^1_{rewind} is deliberately chosen to be an identity function it need not be used in the decomposition process at all. A possible set of sequences of d-functions that extracts all the records from a non-empty file of length **n** is

$\{ d^2_{skip} , d^2_{insert} \cdot d^1_{skip} , \ldots , d^2_{insert} \cdot (d^1_{insert})^{n-1} \cdot d^1_{skip} \}$

The design of the d-functions above was guided by the particular canonical form of a file value we chose. If we had chosen a different canonical form for a file for a different application, it would have perhaps led to a different set of d-functions. Note that the definition of d-functions does not require that they have to be designed with respect to any canonical form of a value; however, the methodology based on a canonical form simplifies the design process for d-functions, as illustrated above.

4.1.1.4 Discussion

In the general case of a data type, where the constructors can take more than one argument of the defined type, a value might have to be constructed starting from arbitrarily many basic values. A binary tree is an example of such a situation. Then we need a set of decomposing sequences to decompose a value; every decomposing sequence in the set will be generating one of the several basic values needed to construct the given value. We have to use each of these decomposing sequences in a manner explained before to get all the sequences of d-functions that extract the values of the defining types.

In the above definition of d-functions, we associate **n** d-functions for every n-ary constructor in Ω_c; however, this association is not essential. The only requirement needed is that the set of d-functions satisfy the property P1. The advantage of having

this association is that it encourages the methodology for designing d-functions that was discussed in the preceding sections. We believe that the methodology is simple and elegant.

4.1.2 The p-functions

The property (P1) in section 4.1.1.1 that defines the d-functions guarantees the existence of a set of sequences of d-functions that extracts all the values of the defining types from a given value. Given the information about the structure (i.e., a constructing sequence) of a value, we saw how the values of the defining types can be extracted from it. But, for the d-functions to be useful in implementing other functions on the defined type, we should be able to extract the values of the defining types without any a priori knowledge about the structure of the value of the defined type. For this, we need a device that can help us extract the structural information of a value. We introduce a set of functions, called *p-functions*, as a part of the set of distinguished functions for this purpose.

P-functions take the form of predicates. They are defined so as to guide us in the decomposition process; they help us pick the appropriate d-functions in an appropriate order to decompose any given value to a basic value. For this, the p-functions have to satisfy the following two properties:

(a) They should help us terminate the decomposition process. That is, they should help us determine whether we have decomposed the given value to a basic value.

(b) At every step in the decomposition process, they should help us pick a decomposer which makes us move "closer" to a basic value. Note that in the absence of this information, the decomposition process cannot be guaranteed to terminate.

The first requirement can be handled easily by having predicates to test if a value is a basic value. So we have :

(P2) Associated with every basic constructor σ_i in Ω_c , there exists a p-function, p_i , such that $p_i(v)$ iff v is the value constructed by σ_i

To formalize the second requirement, we need to define a relation on the value

set of the data type that reflects how "close" a particular value is to a basic value. We call this relation **is_closer_than**. For ease of exposition, we first define the relation for the simple case where every constructor in Ω_c of the data type takes at most one argument of the defined type. We extend the definition to the general case later.

For the simple case, we need only a single decomposing sequence to decompose a value to a basic value. We define the *distance* of a value to be the least number of decomposers necessary to decompose a value to a basic value, i.e., the length of a shortest decomposing sequence for the value. Then, **v is_closer_than v'** iff the distance of v is less than the distance of v'. We define a set of p-functions that helps to pick the appropriate decomposer.

(P3) Associated with every nonbasic constructor σ_j, there exists a p-function, p_j, such that $p_j(v)$ iff $d_j^k(v)$ **is_closer_than** v

Assuming that **stack [N]** and **file [t]** have d-functions as chosen earlier, we define p-functions for them below. For **stack [N]**, the p-function associated with **null** should test if a value is the one created by **null**, i.e., if a stack is empty. The decomposer associated with **push**, i.e., **pop**, reduces the distance of a stack value if it is non-empty; so p_{push} has to determine if a stack is non-empty. We have the following implementations.

$p_{null}(v) \triangleq$ **empty(v)**

$p_{push}(v) \triangleq$ **not(empty(v))**

For **file [t]**, there are several decomposing sequences based on the d-functions we have proposed in section 4.1.1.2 Let us design a set of p-functions that guides us through a decomposing sequence of the form $d^1_{skip} \cdot (d^1_{insert})^n$. According to this strategy, we want d^1_{skip} to be applied right at the beginning but only when the file is not rewound, i.e., when the position of the pointer is greater than one. Secondly, we do not intend to apply d^1_{rewind} at all; so p_{rewind} can always be **false**. Lastly, our intention is to decompose the file starting from the leftmost record; so d^1_{insert} should be applied if the file is non-empty and the file is rewound.

$p_{skip}(v) \triangleq$ **(pos(v) > 1)**

$p_{insert}(v) \triangleq not(empty(v)) \;\&\; (pos(v) = 1)$

$p_{rewind}(v) \triangleq false$

For the general case, we need a set of decomposing sequences to decompose a value, since a value could be constructed from a set of zero or more basic values. Every value may have several such sets of decomposing sequences. So, we have to change the definition of the distance of a value. The length of a longest decomposing sequence inside a set of decomposing sequences for a value is called a *relative distance* for the value. We then define the *distance* to be the minimum of all relative distances for the value. (Note that the definition of distance in the general case reduces to the definition given before in the simple case.) The rest of the definitions for p-functions remain as before. So (P3) above would be interpreted as follows: if $p_j(v)$ is true then the set of decomposers associated with σ_j could be applied to v successfully.

4.2 Definition of Expressive Richness

Based on the notions of d-functions and p-functions, we capture the informal notion of richness of the operation set of D by means of the following definition for expressive richness.

Definition 5 The operation set of a data type D is *expressively rich* if every function in a set of distinguished functions with respect to a minimal subset of constructors, Ω_c, can be implemented in terms of the operations in Ω using only the mechanisms of composition and conditional expressions.

The above definition satisfies both the goals that motivated the extension of the notion of expressive completeness. The theorem to follow shows that the definition maintains the logical power of a type as before. The richness aspect is taken care of by the distinguished functions. The distinguished functions characterize all desirable basic manipulations that one might want to perform on values. All useful functions on values can be implemented conveniently in terms of them, since they permit one to extract all relevant information from a value easily. The requirement that the distinguished functions be implementable in terms of the operation set without the use of recursion

avoids the use of enumeration in the implementation of the distinguished functions.

Theorem 4 Assuming that every defining type of D has a decidable **equal** predicate, and is expressively complete, if D is expressively rich, then D is also expressively complete.

Proof We show that the **equal** predicate for D can be defined in terms of the distinguished functions of D, and the **equal** predicates on the defining types of D. Then the required result follows from **Theorem 2**.

Let us suppose the following: $\Omega_c = \{ \sigma_1, \ldots, \sigma_N \}$; **n(i)** is the arity of σ_i ; $\sigma_1, \ldots, \sigma_K$ are the basic constructors (i.e., constructors that do not take arguments of type D); $\sigma_{K+1}, \ldots, \sigma_N$ are the non-basic constructors.

For convenience, in the following implementation, we use = to denote the **equal** predicate on all types inside the body of the code. The first k main clauses constitute the basis condition. The remaining clauses form the recursive step, since they involve recursive invocations of equal on D.

equal(v1, v2) ≙

if $p_1(v1)$ then $p_1(v2) \;\&\; d_1^1(v1) = d_1^1(v2) \;\&\; \ldots \;\&\; d_1^{n(1)}(v1) = d_1^{n(1)}(v2)$

else if $p_2(v1)$ then $p_2(v2) \;\&\; d_2^1(v1) = d_2^1(v2) \;\&\; \ldots \;\&\; d_2^{n(2)}(v1) = d_2^{n(2)}(v2)$

.

.

else if $p_k(v1)$ then $p_k(v2) \;\&\; d_k^1(v1) = d_k^1(v2) \;\&\; \ldots \;\&\; d_k^{n(k)}(v1) = d_k^{n(k)}(v2)$

else if $p_{k+1}(v1)$ then $p_{k+1}(v2) \;\&\; d_{k+1}^1(v1) = d_{k+1}^1(v2) \;\&\ldots\; \&\; d_{k+1}^{n(k+1)}(v1) = d_{k+1}^{n(k+1)}(v2)$

.

.

else $p_N(v2) \;\&\; d_N^1(v1) = d_N^1(v2) \;\&\; \ldots \;\&\; d_N^{n(N)}(v1) = d_N^{n(N)}(v2)$

Q.E.D.

4.3 Examples

In this section we investigate the expressive richness of the designs of the data types we have discussed so far. The method we use is to design a set of distinguished functions for the type, and then see if the functions are implementable in terms of the operations of the type without the use of recursion.

(1) stack [N] is null, push, pop, top, empty

(2) file [t] is null, insert, rewind, skip, delete, read, pos, empty, eof, length

The above two designs are expressively rich; for each of them we designed a family of distinguished functions in the preceding sections which were demonstrated to be implementable as simple compositions of the operations the type.

(3) set [N] is null, insert, remove, has, empty, equal

Ω_c = {null, insert}

One possible set of distinguished functions is :

$d^2_{insert}(s) \triangleq max(s)$
$d^1_{insert}(s) \triangleq remove(s, max(s))$,

where max returns the maximum element in the set.

$p_{null}(s) \triangleq empty(s)$
$p_{insert}(s) \triangleq not(empty(s))$

Another possible set of distinguished functions is :

$d^2_{insert}(s) \triangleq min(s)$,

where min returns the least element of the set.

$d^1_{insert}(s) \triangleq remove(s,min(s))$

p_{null} and p_{insert} are the same as before.

set [N] is not expressively rich as it is, because it is not possible to implement without using recursion a function to extract some element from a set. However, if we include **max** or **min** as an additional operation of **set [N]**, then the modified design is expressively rich.

5. Concluding Remarks

In a strongly typed system with abstract data types, the designer of a type ought to be careful in choosing the operations for the type. If the operation set chosen is not rich enough, it might be impossible or inconvenient to discover certain useful properties about the values of the type. In this paper, we have provided a formal characterization of the expressive power of the operation set of a data type. We defined two properties related to the expressiveness of a data type - expressive completeness and expressive richness. We believe that such a characterization can help one gain a better insight into the intuitive aspects of the design of a data type. In the following, we discuss a few situations in which requiring a data type to be expressively rich proves to be beneficial.

An important advantage of a data type is that it delineates the use of the type from its implementation. Among other things, this enables one to provide several versions (implementations) of the same data types; each version can be made suitable for a particular class of applications. In such a context, the user might often want to convert among the values belonging to different versions. Having the data type expressively rich (or, at least expressively complete) can be very helpful for the user in such a situation. The user will be able to easily write conversion routines himself, since the operations of an expressively rich type can be conveniently used to extract all the information needed to reconstruct any given value; there is no need to divulge any information about the representation for this purpose. The property of expressive richness of a data type also enables its user to convert any of its values to an external representation of his choice. This makes it possible to write output routines to either store a value efficiently in a backup store or to display the value in a suitable format on a peripheral device.

An elegant way to incorporate protection [6] for data objects in a system that supports abstract data types is to control the set of operations that is made available to the user. Based on the kind of information that needs to be protected from the user some operations are made inaccessible to him. In such a context, comparison of

expressive power of different sets of operations of a data type becomes essential. The expressiveness properties discussed in the paper are of help here. For instance, one should make sure that the subset of operations that is accessible to the user is not expressively complete. The distinguished functions can be used as a guide to determine which operations of the data type ought to be made inaccessible to the user.

Another interesting situation where the property of expressive richness plays a useful role is in automatic synthesis of an implementation of a data type in terms of another data type (representation type) from their algebraic specifications. The synthesis procedure under investigation in [15] derives implementations for the operations by transforming the axioms of the type in two stages. In the first stage, the axioms are transformed into a form in which they are expressed as formulas on the values of representation type. In the second stage, the axioms are converted into implementations involving the operations of the representation type. The concept of distinguished functions are helpful in the second stage. The axioms in the intermediate form can be systematically converted into implementations involving the distinguished functions of the representation type. This is because the distinguished functions provide the information that is implicit in the pattern matching mechanism used by algebraic axioms. So the job of the synthesis procedure is reduced to one of finding implementations for the distinguished functions of the representation type. Although this is a non-trivial task, it turns out that the task can be automated with relative ease for expressively rich data types.

In this paper, we have only concentrated on immutable data types. It would be interesting to conduct a similar analysis for mutable data types also. We imagine that the requirement in the mutable case has to be stronger, since the ability to distinguish object identity also ought to be taken into consideration.

Acknowledgements

We are thankful to Carl Seaquist for his insightful suggestions throughout the development of this paper, and to Craig Schaffert for bringing to our notice a mistake in an earlier draft of the paper. We are also thankful to Valdis Berzins, John Guttag, Barbara Liskov, Eliot Moss, and N. Natarajan for their helpful comments.

REFERENCES

1. Greif, I.G., *Induction in Proofs about Programs.* MAC TR-93, M.I.T., Cambridge, MA, Feb., 1972.

2. Goguen, J.A., Thatcher, J.W., Wagner, E.G., "Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types," *Current Trends in Programming Methodology,* Vol. IV, Data Structuring, (Ed. Yeh, R.T.), Prentice Hall (Automatic Computation Series), Englewood Cliffs, New Jersey, 1978.

3. Guttag, J.V., *The specification and Application to Programming of Abstract Data Types.* Ph. D. Thesis, University of Toronto, CSRG-59, 1975.

4. Guttag, J.V., Horowitz, E., Musser, D.R., "Abstract Data Types and Software Validation," *Comm. ACM* Vol. 21 No. 12, 1048-1064, Dec. 1978.

5. Hoare, C.A.R., "Proof of Correctness of Data Representations," *Acta Informatica* Vol. 1, No. 4, pp 271-281, 1972.

6. Jones, A.K., Liskov, B.H. "A Language Extension for Controlling Access to Shared Data," *IEEE Tran. on Software Engg Vol. SE-2 No. 4,* pp. 277-285, Dec. 1976.

7. Kapur, D., Srivas, M.K., *Expressiveness of the Operation Set of A Data Abstraction.* Computation Structures Group Memo 179-1, Lab. for Computer Science, M.I.T., Cambridge, MA, June, 1979, Revised Nov., 1979.

8. Kapur, D., *Towards a Theory for Abstract Data Types* Forthcoming Ph.D. Thesis, Dept. of E.E. & C.S., M.I.T., Cambridge, Mass., Jan., 1980.

9. Kleene, S.C., "General Recursive Functions of Natural Numbers," *Mathematical Annals 112,* pp. 727-742 (1936).

10. Liskov, B.H., Snyder, A., Atkinson, R., Schaffert, C., "Abstraction Mechanisms in CLU," *CACM* Vol. 20 No. 8, pp. 564-576, 1977.

11. Liskov, B.L., Snyder, A.S., *Exception Handling In CLU.* Computation Structures Group Memo 155-2, Lab. for Computer Science, M.I.T., Cambridge, MA, Dec., 1977, Revised March 1979. To appear in *IEEE Trans. on Software Engineering.*

12. McCarthy, J., "A Basis for a Mathematical Theory of Computation" in *Computer Programming and Formal Systems,* (Eds. Braffort and Hirschberg),

North Holland Publishing Co., Amsterdam -London, pp 33-70, 1963.

13. Morris, J.H., Jr., "Towards More Flexible Systems," *Lecture Notes in Computer Science 19*, Springer-Verlag, pp. 377-383, 1974.

14. Rogers, H., Jr., *Theory of Recursive Functions and Effective Computability.* McGraw-Hill Series in Higher Mathematics, McGraw-Hill, Inc., 1967.

15. Srivas, M.K., "Draft of a Thesis Proposal on Automatic Synthesis of Abstract Data Types," LCS, M.I.T., December 1978.

16. Weihrauch, K., "A Generalized Computability Thesis," *Lecture Notes in Computer Science 56*, Springer-verlag, pp. 538-542, 1977.

17. Wulf, W., London, R.L., and Shaw, M., *Abstraction and Verification in ALPHARD: Introduction to Language and Methodology.* Carnegie-Mellon University Technical Report, also USC Information Sciences Institute Research Report, 1976.

18. Zilles, S.N., *An Introduction to Data Algebra.* Draft Working Paper, IBM San Jose Research Lab., Sept. 1975.

Appendix I - Definitions of Data Types

We present the definitions of various data types discussed in the body of the paper. We first give the syntactic specifications of the operations of a data type, and then we give a model of the data type. We usually use the first two letters of an operation name to stand for its interpretation in the model. The definitions of the functions are presented in any convenient mathematical notation. A data type is the set of all algebras behaviorally equivalent to the given model.

set [N] is null, insert, remove, has, empty, max, equal

null : --> set [N]
insert : set [N] X N --> set [N]
remove : set [N] X N --> set [N]
has : set [N] X N --> **bool**
empty : set [N] --> **bool**
max : set [N] --> N
equal : set [N] X set [N] --> **bool**

The model $\mathfrak{A}_{sn}$ is a natural model of **set [N]** in the sense that its value set is the set of all finite sets of natural numbers, and the interpretations of its operations are defined in terms of the standard set operations.

$\mathfrak{A}_{sn}$ = [{ StN, N, B }; { nu, in, re, ha, em, ma, eq }],
where B = { **true**, **false** }, a value set of **bool**,
N = { 0, 1, 2, 3, . . . }, a value set of N, and
StN = { Φ, {0}, {1}, {2}, {0, 1}, {0, 2}, {1, 2}, {3}, {0, 3},
{1, 3}, {2, 3}, {0, 1, 2}, . . . }, a value set of **set [N]**.

nu = Φ
in(s, i) = s ∪ {i}
re(s, i) = s - {i} ; - is the difference operator
ha(s, i) = i ∈ s
em(s) = **true** if s is the empty set
false otherwise
ma(s) = 0 if s is the empty set
n | n ∈ s ∧ (∀ i) i ∈ s ==> i ≤ n otherwise
eq(s1, s2) = **true** if s1 and s2 are the same set
false otherwise

stack [N] is null, push, pop, top, empty, equal

```
null    :  --> stack [N]
push    : stack [N] X N  --> stack [N]
pop     : stack [N]  --> stack [N]
top     : stack [N]  --> N
empty   : stack [N]  --> bool
equal   : stack [N] X stack [N]  --> bool
```

The model $\mathfrak{A}_{stk}$ of **stack [N]** has sequences of natural numbers as the values of **stack [N]**.

$\mathfrak{A}_{stk}$ = [{ SqN, N, B }; { nu, pu, po, to, em, eq }],

where SqN is the set of sequences of natural numbers.

SqN = { ⟨⟩, ⟨0⟩, ⟨1⟩, ⟨2⟩, ⟨0, 0⟩, ⟨0, 1⟩, ⟨0, 2⟩,
⟨1, 0⟩, ⟨1, 1,⟩, ⟨1, 2⟩, ⟨2, 0⟩, ⟨2, 1⟩,
⟨2, 2⟩, ⟨3⟩, ⟨0, 3⟩, . . . }.

The interpretations of the operation names are defined as follows ($\langle e_i, \ldots, e_j \rangle$ is the empty sequence if $j < i$) :

$nu = \langle\rangle$

$pu(\langle e_1, \ldots, e_m \rangle, e) = \langle e_1, \ldots, e_m, e \rangle$

$po(\langle\rangle) = \langle\rangle$

$po(\langle e_1, \ldots, e_m \rangle) = \langle e_1, \ldots, e_{m-1} \rangle$ if $m > 0$.

$to(\langle\rangle) = 0$

$to(\langle e_1, \ldots, e_m \rangle) = e_m$ if $m > 0$.

$em(\langle\rangle) = T$

$em(\langle e_1, \ldots, e_m \rangle) = F$ if $m > 0$

$eq(\langle e_{11}, \ldots, e_{1m} \rangle, \langle e_{21}, \ldots, e_{2n} \rangle) = T$ if $m = n$ and for each i, $e_{1i} = e_{2i}$

F otherwise.

file [t] is null, insert, rewind, skip, delete, read, pos, empty, eof, length

null : --> **file [t]**
insert : **file [t] X t** --> **file [t]**
rewind : **file [t]** --> **file [t]**
skip : **file [t] X N** --> **file [t]**
delete : **file [t]** --> **file [t]**
read : **file [t]** --> **t**
pos : **file [t]** --> **N**
empty : **file [t]** --> **bool**
eof : **file [t]** --> **bool**
length : **file [t]** --> **N**

The model $\mathfrak{A}_{fl}$ has a set of 2-tuples whose first element is a sequence of records and the second element is a natural number, as the value set. We assume that the function corresponding to **read** returns the element r_0 when the pointer is at the end of file. The functions corresponding to **rewind** and **read** are denoted as **re** and **rd** respectively.

$\mathfrak{A}_{fl}$ = [{ **Fl, N, B** };
{ **nu, in, re, sk, de, rd, po, em, eo, le** }]

where **Fl** is a subset of **<SqT, N>**, where **SqT** is the set of sequences of elements of type **t**, such that every member **f** = $\langle\langle r_1, \ldots, r_n\rangle, k\rangle$ of **Fl** has $1 \leq k \leq n+1$. Let **s** stand for a sequence $\langle r_1, \ldots, r_n\rangle$ of n records.

nu = <>
in(f, r) = $\langle\langle r_1, \ldots, r_{k-1}, r, r_k, \ldots, r_n\rangle, k+1\rangle$
sk(f, m) = $\langle s, k+m\rangle$ if $k+m \leq n+1$
$\langle s, n+1\rangle$ otherwise
rw(f) = $\langle s, 1\rangle$
de(f) = $\langle\langle r_1, \ldots, r_{k-1}, r_{k+1}, \ldots, r_n\rangle, k\rangle$ if $k \leq n$
f otherwise
re(f) = r_k if $k \leq n$
r_0 otherwise
po(f) = **k**
em(f) = **(n = 0)**
eo(f) = **(k = n+1)**
le(f) = **n**

Appendix II - Specifications of Data Types

This appendix gives the algebraic specifications of the data types that were defined in Appendix I.

set [N] is null, insert, remove, has, empty, max, equal

Operations

```
null    : --> set [N]
insert  : set [N] X N --> set [N]
remove  : set [N] X N --> set [N]
has     : set [N] X N --> bool
empty   : set [N] --> bool
max     : set [N] --> N
equal   : set [N] X set [N] --> bool
```

Axioms

```
insert(insert(s, e1), e2) ≡ if e1 = e2 then s else insert(insert(s, e2), e1)

remove(null, e) ≡ null
remove(insert(s, e1), e2) ≡ if e1 = e2 then remove(s, e2)
                            else insert(remove(s, e2), e1)

has(null, e) ≡ false
has(insert(s, e1), e2) ≡ if e1 = e2 then true else has(s, e2)

empty(null) ≡ true
empty(insert(s, e)) ≡ false

max(null) ≡ 0
max(insert(null, e) ≡ e
max(insert(insert(s,e1),e2)) ≡ if e2 > max(insert(s, e1)) then e2
                               else max(insert(s, e1))

equal(null, null) ≡ true
equal(null, insert(s, e)) ≡ false
equal(insert(s, e), null) ≡ true
equal(insert(s1, e1), insert(s2, e2)) ≡ has(insert(s1, e1), e2)
                                      ∧ has(insert(s2, e2), e1)
                                      ∧ equal(remove(remove(s1, e1), e2),
                                              remove(remove(s2, e2), e1))
```

stack [N] is null, push, pop, top, empty, equal

Operations

```
null    :  --> stack [N]
push    : stack [N] X N  --> stack [N]
pop     : stack [N]  --> stack [N]
top     : stack [N]  --> N
empty   : stack [N]  --> bool
equal   : stack [N] X stack [N] --> bool
```

Axioms

pop(null) ≡ null
pop(push(s, e)) ≡ s

top(null) ≡ 0
top(push(s, e)) ≡ e

empty(null) ≡ true
empty(push(s, e)) ≡ false

equal(null, null) ≡ true
equal(null, push(s, e)) ≡ false
equal(push(s, e), null) ≡ false
equal(push(s1, e1), push(s2, e2)) ≡ equal(e1, e2) ∧ equal(s1, s2)

---

file [t] is null, insert, skip, rewind, read, pos, empty, eof, length

Operations

```
null   :  --> file [t]                    as  N
insert :  file [t] X t  --> file [t]      as  In
skip   :  file [t] X N  --> file [t]      as  S
rewind :  file [t]  --> file [t]          as  Rw
delete :  file [t]  --> file [t]          as  D
read   :  file [t]  --> t                 as  Re
pos    :  file [t]  --> N
empty  :  file [t]  --> bool              as  E
eof    :  file [t]  --> bool
length :  file [t]  --> N                 as  Ln
```

Axioms

```
S(f, 0)  ≡  f
S(N, n)  ≡  N
S(S(f, n1), n2)  ≡  S(f, n1+n2)

Rw(N)  ≡  N
Rw(Rw(f))  ≡  Rw(f)
Rw(S(f, n))  ≡ Rw(f)

f  ≡  S(f, pos(f) -1)

In(S(Rw(In(f, r1)), n), r2)  ≡  if n = pos(f) -1 then S(Rw(In(In(f, r2), r1)), n+1)
  else if n < pos(f) -1 then S(Rw(In(S(Rw(In(S(Rw(f), n), r2)), pos(f)), r1)), n+1)
  else if n > pos(f) -1 then S(Rw(In(S(Rw(In(S(Rw(f), n-1), r2)), pos(f)-1), r1)), n+1)

D(N)  ≡  N
D(S(Rw(In(f, r)), n))  ≡ if n > Ln(f) then S(Rw(In(f, r)), n)
         else if n = pos(f) -1 then S(Rw(f), n)
         else if n < pos(f) -1 then S(Rw(In(S(Rw(D(S(Rw(f), n))), pos(f)-2), r)), n)
         else if n > pos(f) -1 then S(Rw(In(S(Rw(D(S(Rw(f), n-1))), pos(f)-1), r)), n)

Re(N)  ≡  r0
Re(S(Rw(In(f, r)), n)  ≡  if n = pos(f) -1 then r
                          else if n < pos(f) -1 then Re(S(Rw(f), n))
                          else  Re(S(Rw(f), n-1))

pos(N)  ≡  1
pos(In(f, r))  ≡  pos(f) +1
pos(S(f, n))  ≡  min(pos(f) + n, Ln(f)+1)
pos(Rw(f))  ≡  1

E(f)  ≡  Ln(f) = 0

Ln(N)  ≡  0
Ln(In(f, r))  ≡  Ln(f) +1
Ln(S(f, n))  ≡  Ln(f)
Ln(Rw(f))  ≡  Ln(f)

eof(f)  ≡  if pos(f) > Ln(f) then true else false
```

Appendix III - Proof of Theorem 3

Theorem 3 Assuming that (i) for each of the defining types of D the **equal** predicate is decidable, and (ii) every defining type of D is expressively complete, D is expressively complete if the operation set of D includes the **equal** predicate on D.

Proof A function f in **Comp(D)** is a function on W preserving E; as explained in chapter 2, f can also be viewed as a function on V, i.e., W/E. In the following we take this latter view of f, so that we do not have to continually consider the equivalence relation E.

We already know that for any D it is possible to construct an encoding function η that can be used to map every function of D to a function on natural numbers. Because of our change of view of functions of D, it is convenient to come up with a similar scheme that encodes values as numbers. The two premises of the theorem guarantee the existence of a computable bijective function $\delta' : N \to V$. We call δ' a *numbering* for D; let η' be the inverse of δ'. We show later how one such numbering scheme can be constructed from a pair of encoding-decoding functions (η and δ) for D. We use $\eta' \cdot f \cdot \delta'$ as the image of f. For convenience, we define the *inverse image* of a function $\overline{g}$ on N as the function $\delta' \cdot \overline{g} \cdot \eta'$. Note that if f is computable (i.e., $f \in$ **Comp(D)**) then so is its image (via $\langle \eta', \delta' \rangle$), and if $\overline{g}$ on N is computable, so is its inverse image.

To prove that D is expressively complete, we have to show that **Comp(D)** = $\mathfrak{C}(\Omega)$. We do this in two parts:

Part I $\mathfrak{C}(\Omega) \subseteq$ **Comp(D)**

Every operation in Ω is computable and the mechanisms of composition, recursion, and conditional expression preserve computability; so it is obvious that $\mathfrak{C}(\Omega) \subseteq$ **Comp(D)**.

Part II **Comp(D)** $\subseteq \mathfrak{C}(\Omega)$

Let the functions 0_D , S_D , and $=_D$ on D be the inverse images of the natural number functions 0, S, and =, respectively. The proof of **Comp(D)** $\subseteq \mathfrak{C}(\Omega)$ follows from the following two claims:

(i) **Comp(D)** $\subseteq \mathfrak{C}(\{ 0_D , S_D , =_D \})$

(ii) $\mathfrak{C}(\{ 0_D , S_D , =_D \}) \subseteq \mathfrak{C}(\Omega)$

To prove (i), let $f \in$ **Comp(D)**. So there exists a computable function $\overline{f}$ on N such that $f = \delta' \cdot \overline{f} \cdot \eta'$; $\overline{f} \in \mathfrak{C}(\{ 0 , S, = \})$. Let $\overline{f}(x) \triangleq$ **Def**$(\overline{f})(x)$, where **Def**$(\overline{f})(x)$ is the implementation of $\overline{f}$, expressed using composition, recursion, and conditional expression, possibly using a set of auxiliary functions. Now consider the function $f' : D \to D$ whose implementation is obtained by replacing every occurrence of $\overline{f}$, 0, S, =, and the auxiliary functions in **Def**$(\overline{f})$ by their corresponding inverse images. f' obviously belongs to $\mathfrak{C}(\{ 0_D , S_D , =_D \})$. Lemma 1 below shows that f' is indeed the inverse image of f,

and hence is functionally equivalent to **f**.

Lemma 1 $f'(x) = \delta \cdot \overline{f} \cdot \eta(x)$.[5]

Proof We prove the equivalence of the two recursively defined functions using Morris's truncation induction rule [1]. We augment the value set of every data abstraction **D** by the *undefined* element, denoted by $\perp_D$, and order the augmented domain such that $\perp_D < v$, where v is any non-$\perp_D$ value, and the non-$\perp_D$ values are non-comparable. The functions on **D** are assumed to be defined on the augmented domains. The numbering function δ' is also extended so that $\delta'(\perp_N) = \perp_D$. Let $\perp^D$ denote the constant function on **D** that returns $\perp_D$, and $\perp^N$ denote the constant function on **N** that returns $\perp_N$. Then we have $\perp^D = \delta' \cdot \perp^N \cdot \eta'$.

Let g_i stand for the i^{th} truncation function of **g**. We can show the lemma by proving that

(i) $f'_0(x) = \delta' \cdot \overline{f}_0 \cdot \eta'(x)$,

(ii) if $(\forall\ i < j)\ f'_i(x) = \delta' \cdot \overline{f}_i \cdot \eta'(x)$,
then $f'_j = \delta' \cdot \overline{f}_j \cdot \eta'(x)$,

where $f'_0(x) = \perp^D(x)$, $\overline{f}_0(x') = \perp^N(x')$,
and for $i > 0$, $f'_i(x) = \mathbf{Def}'(f'_{i-1})(x)$, and $\overline{f}_i(x') = \mathbf{Def}(\overline{f}_{i-1})(x')$.

The proof of (i) is trivial. (ii) can be proved by induction on the structure of the definition **Def**, where **Def** being 0, S, or = serves as the basis. The inductive step involves two cases - (a) when **Def** is a composition, and (b) when **Def** is a conditional expression.

Q.E.D.

We prove claim (ii) constructively by giving implementations for 0_D, S_D, and $=_D$ in terms of the **equal** operation of **D** in conjunction with a minimal set of constructors of **D**. Note that the latter is the least we can assume about Ω.

Let Ω_c denote a minimal subset of constructors of **D**; so $\Omega_c \cup \{ \mathbf{equal} \} \subseteq \Omega$. Without any loss of generality, we assume that every constructor in Ω_c takes

(i) at most one argument from **D**, and

(ii) at most one argument from the value set of a data type other than **D**.

We classify Ω_c into the following four subsets.

$\Omega_c^{0,0} = \langle c_i , \ldots , c_j \rangle$, where c_i : -> **D**.

---

5. In order to simplify the proof, we have not considered the case when **f** is defined mutually recursively using a system of recursive definitions. For this case also, the proof can be worked out along the similar lines.

$\Omega_c^{0,1} = \{ q_1, \ldots, q_k \}$, where $q_i : D_i \rightarrow D$ and D_i different from D.
$\Omega_c^{1,0} = \{ r_1, \ldots, r_l \}$, where $r_i : D \rightarrow D$.
$\Omega_c^{1,1} = \{ s_1, \ldots, s_m \}$, where $s_i : D \times D_i \rightarrow D$, where D_i is different from D.

We define a numbering scheme that induces a total ordering on the value set of D. The least element of this ordering is chosen to be 0_D. The function S_D is defined so that when S_D is passed a value v of D, it returns the value whose number is one greater than that of v. We implement[6] S_D such that it has the following behavior: $(\forall v \in D)[S_D(v) = \delta'(\eta'(v) + 1)]$. The predicate $=_D$ is the same as the predicate **equal** of D. Below we sketch the numbering on which the implementations of the functions are based, and then give an implementation for S_D. Later, for the sake of completeness, we also give the obvious implementations for η' and δ' in terms of S_D.

Q.E.D.

(1) A Numbering Scheme for D

In order to simplify the presentation, we assume that the cardinality of each of the sets $\Omega_c^{0,0}$, $\Omega_c^{0,1}$, $\Omega_c^{1,0}$, $\Omega_c^{1,1}$ is 1; the approach discussed below easily extends to the general case where the cardinalities are arbitrary.

$\Omega_c^{0,0} = \{ c \}$, c: $\rightarrow$ D
$\Omega_c^{0,1} = \{ q \}$, q: $D_1 \rightarrow$ D
$\Omega_c^{1,0} = \{ r \}$, r: D $\rightarrow$ D
$\Omega_c^{1,1} = \{ s \}$, s: D X $D_2 \rightarrow$ D

Let $\langle \eta_1, \delta_1 \rangle$ and $\langle \eta_2, \delta_2 \rangle$ denote the encoding-decoding pairs for D_1 and D_2, respectively. The following table depicts the encoding function for D. An implementation of η maintains a counter; it increments the counter while enumerating the words formed out of the constructors in Ω_c in the order indicated below until it hits the word which is to be encoded; the value of the counter at this point is the code for the given word.

---

6. In a condensed version of this paper that appeared in the proceedings of the 7th POPL Conference, we mention (by mistake) that S_D is implemented (in this paper) in terms of η' and δ'. Instead, here we provide a direct implementation for S_D which is based on a numbering scheme since such an implementation is more interesting.

v	$\eta(v)$
c	0
$q(\delta_1(0))$	1
$r(\delta(0))$	2
$s(\delta(0), \delta_2(0))$	3
$q(\delta_1(1))$	4
$r(\delta(1))$	5
$s(\delta(1), \delta_2(0))$	6
$s(\delta(0), \delta_2(1))$	7
$s(\delta(1), \delta_2(1))$	8
$q(\delta_1(2))$	9
$r(\delta(2))$	10

A numbering for D can be derived from the above table by grouping together all words that yield equivalent values. For example, let us suppose that among the first 10 words, the following set of words represent the same value.

$c, q(\delta_1(2)), s(\delta(0), \delta_2(1))$

$q(\delta_1(0)), r(\delta(0)), s(\delta(0), \delta_2(0))$

$q(\delta_1(1)), s(\delta(1), \delta_2(0))$

$r(\delta(1)), r(\delta(2)), s(\delta(1), \delta_2(1))$

Then the first 4 values are encoded as depicted in the following table. An implementation of η' is also based on the enumeration of all the words formed out of the constructors in Ω_c. It enumerates the words in the same order as an implementation of η would, but it increments the counter selectively; it increments the counter if and only if the next word generated in the enumeration is the first word to be generated from the equivalence class to which the word belongs.

v	$\eta'(v)$
$[c]$	0
$[q(\delta_1(0))]$	1
$[q(\delta_1(1))]$	2
$[r(\delta(1))]$	3

(2) Implementation of S_D

The strategy employed in the implementation is the following. We generate the values of D in the order specified in the second table shown before until we hit the value v, the argument to S_D. Then the required result is the value generated next in the sequence. Note that the ordering on values is induced by an ordering on the words that construct those values. The algorithm used in the implementation enumerates the values

by generating all the words in the order depicted in the first table shown before. But an enumeration of words can result in repitition of values of D. The algorithm avoids this repitition by checking every time it generates a new word if the value constructed by the word was already constructed by a word already generated. The function **seen** does this checking in the implementation.

The enumeration inside S_D can be considered to be conducted in two distinct states - the pre-terminal state and the terminal state. Enumeration begins in the pre-terminal state, and remains in that state until a word that constructs a value equal to v is generated; so it is not necessary to bother about repititions of values in the pre-terminal state. Enumeration in the terminal state continues until a word that constructs a "new" value (i.e., a value that was never constructed by any word generated before) is generated; so the enumeration in the terminal state uses the function **seen**.

The definition of S_D uses several auxiliary functions. Two sets of functions are used to do the enumeration in the two states. **q_preterm, r_preterm, s_preterm1,** and **s_preterm2** do the enumeration in the pre-terminal state. **q_term, r_term, s_term1,** and **s_term2** do the enumeration in the terminal state. The enumerating functions in each set are setup in such a way that they call each other cyclically to generate the words in the desired order. The auxiliary functions **D1_to_D, D2_to_D,** and **D2_to_D1** are used to convert values that have the same encodings from a source domain to a destination domain.

S_D : D -> D
$S_D(v)$ ≜ if v = c then q_term(v, 0_{D_1}) else q_preterm(v, 0_{D_1})

q_preterm : D X D_1 -> D
q_preterm(v, d1) ≜ if v = q(d1) then r_term(v, D1_to_D(d1))
else r_preterm(v, D1_to_D(d1))

r_preterm : D X D -> D
r_preterm(v, v') ≜ if v = r(v') then s_term1(v, v', 0_{D_2}) else s_preterm1(v, v', 0_{D_2})

s_preterm1 : D X D X D_2 -> D
s_preterm1(v, v', d2) ≜
if v = s(v', d2)
then if v' = S_D(D2_to_D(d2))
then s_term2(v, 0_D , S_{D_2}(d2))
else s_term1(v, v', S_{D_2}(d2))
else if v' = S_D(D2_to_D(d2))
then s_preterm2(v, 0_D , S_{D_2}(d2))
else s_preterm1(v, v', S_{D_2}(d2))

```
s_preterm2 : D X D X D2 -> D
s_preterm2(v, v', d2) ≙
                  if v = s(v', d2)
                  then if v' = D2_to_D(d2)
                       then q_term(v, S_D1(D2_to_D1(d2)))
                       else s_term2(v, S_D(v'), d2)
                  else if v' = D2_to_D(d2)
                       then q_preterm(v, S_D1(D2_to_D1(d2)))
                       else s_preterm2(v, S_D(v'), d2)

q_term : D X D1 -> D
q_term(v, d1) ≙ if seen(v, q(d1)) then r_term(v, D1_to_D(d1)) else q(d1)

r_term : D X D -> D
r_term(v, v') ≙ if seen(v, r(v')) then s_term1(v, v', 0_D2) else r(v')

s_term1 : D X D X D2 -> D
s_term1(v, v', d2) ≙
                  if not(seen(v, s(v', d2))) then s(v', d2)
                  else if v' = S_D(D2_to_D(d2))
                       then s_term2(v, 0_D , S_D2(d2))
                       else s_term1(v, v', S_D2(d2))

s_term2 : D X D X D2 -> D
s_term1(v, v', d2) ≙
                  if not(seen(v, s(v', d2))) then s(v', d2)
                  else if v' = D2_to_D(d2))
                       then q_term(v, S_D1(D2_to_D1)))
                       else s_term2(v, S_D(v'), d2)
```

In the following definitions P_{D1} and P_{D2} denote the predecessor functions on D_1 and D_2, respectively.

```
D1_to_D : D1 -> D
D1_to_D(d1) ≙ if d1 = 0_D1 then 0_D else S_D(D1_to_D(P_D1(d1)))

D2_to_D : D2 -> D
D2_to_D(d2) ≙ if d2 = 0_D2 then 0_D else S_D(D2_to_D(P_D2(d2)))

D2_to_D1 : D2 -> D1
D2_to_D1(d2) ≙ if d2 = 0_D2 then 0_D1 else S_D1(D2_to_D1(P_D2(d2)))
```

(3) Implementation of seen

seen is a predicate that takes two arguments v and v' of type D. It finds out if v' is ever generated before v is generated for the first time in the enumeration sequence. The strategy is to enumerate the values of D in order until we hit either v or v'. If we hit v' before v then **seen** returns **true**; otherwise, it returns **false**. It uses a similar set of functions to enumerate the domain as before except that the enumeration is done only in one state. We use the suffix **gen** for the auxiliary functions which do the enumeration.

```
seen(v, v') ≜ if v= v' then true
                  else if v = c then false else q_gen(v, v', 0_D1)

q_gen : D X D X D1 -> bool
q_gen(v, v', d1) ≜ if v = q(d1) then false else r_gen(v, v', D1_to_D(d1))

r_gen : D X D X D -> bool
r_gen(v, v', v1) ≜ if v = r(v1) then false
                       else if v' = r(v1) then true else s_gen1(v, v', v1, 0_D2)

s_gen1 : D X D X D X D2 -> bool
s_gen1(v, v', v1, d2) ≜
                  if v = s(v1, d2) then false
                  else if v' = s(v1, d2) then true
                      else if v1 = S_D(D2_to_D(d2)) then s_gen2(v, v', 0_D , S_D2(d2))
                           else s_gen1(v, v', v1, S_D2(d2))

s_gen2 : D X D X D X D2 -> bool
s_gen2(v, v', v1, d2) ≜
                  if v = s(v1, d2) then false
                  else if v' = s(v1, d2) then true
                      else if v1 = D2_to_D(d2) then q_gen(v, v', S_D1(D2_to_D1(d2)))
                           else s_gen2(v, v', S_D(v1), d2)
```

(4) Implementations of δ' and η'

Given below are definitions of δ' and η'. They make use of S_D to do the required enumeration of the values of D. They maintain a counter **count** to halt the enumeration.

```
δ' : N -> D
δ'(n) ≜ δ'_aux(n, 0_D , 0)
```

$\delta'\_aux : N \times D \times N \rightarrow D$
$\delta'\_aux(n, v, count) \triangleq$ if $n = count$ then v else $\delta'\_aux(n, S_D(v), count + 1)$

$\eta' : D \rightarrow N$
$\eta'(v) \triangleq \eta'\_aux(v, 0_D, 0)$

$\eta'\_aux : D \times D \times N \rightarrow N$
$\eta'\_aux(v, v', count) \triangleq$ if $v = v'$ then $count$ else $\eta'\_aux(v, S_D(v'), count + 1)$