A CONSTRUCTIVE ALTERNATIVE TO AXIOMATIC DATA TYPE DEFINITIONS

Robert Cartwright

TR 80-427

June 1980

Department of Computer Science
Cornell University
Ithaca, New York 14853

A Constructive Alternative to Axiomatic Data Type Definitions†

Robert Cartwright

Department of Computer Science
Cornell University
Ithaca, N.Y. 14853

ABSTRACT

Many computer scientists advocate using axiomatic methods (such as algebraic specification) to specify a program data domain -- the universe of abstract data objects and operations manipulated by a program. Unfortunately, correct axiomatizations are difficult to write and to understand. Furthermore, their non-constructive nature precludes automatic implementation by a language processor.

In this paper, we present a more disciplined, purely constructive alternative to axiomatic data domain specification. Instead of axiomatizing the program data domain, the programmer explicitly constructs it by using four type construction mechanisms: constructor generation, union generation, subset generation, and quotient generation. These mechanisms are rich enough to define all of the abstract data objects that programmers commonly use: integers, sequences, trees, sets, arrays, functions, etc. In contrast to axiomatic definitions, constructive definitions are easy to write and to understand. An unexpected advantage of the constructive approach is a limited capacity to support non-deterministic operations. As an illustration, we define a non-deterministic "choose" operation on sets.

1. Introduction

One of the most difficult and poorly understood tasks in programming is defining the program data domain -- the universe of abstract data objects and operations manipulated by the program. In traditional programming practice, the data domain is never precisely defined. Instead, the program manipulates concrete machine data objects (such as pointers and records) and operations (such as pointer dereferencing and record field extraction) representing abstract data objects and operations. If the program is well-documented, the data domain is informally specified in the documentation. Otherwise, it is left to the imagination of the reader.

The main defect in the traditional approach is that it makes reasoning about programs (both formally and informally) extremely difficult. Anyone who attempts to understand the program (including the programmer) must continually decipher machine representations for abstract data objects and operations that are never precisely described.

In an attempt to solve this problem, a number of computer scientists have proposed using axiomatic methods to define the abstract data domain for a program. Some ([Hoare 72], [von Henke and Luckham 74]) advocate expressing data domain[1] axiomatizations in standard first order predicate calculus (with equality). Others, particularly Guttag and the ADJ group [Guttag and Horning 78, ADJ 76, ADJ 77] advocate restricting data domain axiomatizations to finite sets of equations they call "algebraic specifications."

Despite their apparent popularity, axiomatic methods for defining program data domains suffer from three serious weaknesses.

◇ First, correct axiomatic definitions are hard to write. An axiomatic definition must specify the "critical" properties of the data domain. However, it is very difficult to identify which properties are critical and correctly state them in formal terms. Moreover, there is no viable methodology for detecting mistakes in axiomatic definitions.

◇ Second, axiomatic definitions are hard to understand. Instead of explicitly describing the objects composing the domain, they merely specify some properties of the data domain and leave it

†Research supported by NSF grant MCS78-05850.

[1]We are consciously avoiding the term data type since it has radically different meanings in different programming languages. We will give our own definition for the term later in the paper.

to the reader to infer its structure. Since attempted data domain axiomatizations are often inconsistent, the reader has no assurance that the axioms describe any data domain at all.

◇ Third, the axiomatic approach forces the programmer to implement the data domain that he defines. Data domain implementations are complex, time-consuming programming tasks which should ideally be performed by a language processor (interpreter or compiler) supporting abstract data domain definitions. Axiomatic definitions preclude this possibility.

With the objective of developing better methods for defining and reasoning about program data domains, this paper presents a disciplined, purely constructive approach to data domain definitions that builds on the concept of recursive data domain definitions developed by [McCarthy 63] and [Hoare 73]. Instead of axiomatizing the data domain, the programmer explicitly constructs it by using four domain construction operations: constructor definition, union definition, subset definition, and quotient definition.

With these four mechanisms, a programmer can clearly and concisely define all of the abstract types that commonly occur in programs: integers, sequences, trees, sets, arrays, functions, etc. Despite the constructive character of the definitions, they are no less "abstract" than first order axiomatizations or algebraic specifications. In fact, it is easy to generate either a simple first order axiomatic definition (similar in character to Peano's first order axiomatization of the natural numbers) or an algebraic specification (assuming all the operations are computable) for the defined data domain. Hence, constructive data domain definitions constitute a highly disciplined approach to data domain axiomatization; the method guarantees that an axiomatization has a simple symbolic model.

The constructive character of the method makes it feasible for a language processor to assume the burden of actually implementing the defined data domain. A very high level recursive language similar to LISP could support constructive data domain definitions just as PASCAL supports low level, machine-oriented data domain definitions. In this language, LISP S-expressions would merely be one of many definable domains.

A surprising consequence of our constructive viewpoint is a limited, but useful capacity to handle non-deterministic abstract operations. For example it is possible to define an abstract operation to choose an "arbitrary" element of a finite set. To our knowledge, no other data domain definition method yet proposed has this capability. The key idea in our approach is to provide the illusion of an extensional treatment[2] of types (such as finite sets) that are defined as equivalence classes over other types (such as sequences), while retaining a strictly intensional viewpoint in the formal semantic definition. As a result, the programmer can informally define abstract, non-deterministic operations on equivalence classes (such as choosing an arbitrary element of a set) by defining deterministic operations on the corresponding intensional objects.

2. A Critique of Algebraic Specification

An algebraic specification is a first order axiomatization consisting of a finite set of equations, i.e. quantifier-free formulas of the form t1=t2 where t1 and t2 are terms (expressions) constructed from free variables and domain operations. The domain defined by an algebraic specification is a particular term (syntactic) model selected from the lattice of term models satisfying the axiomatization (although Guttag and the ADJ group disagree on the exact choice). Each element of the term model is an equivalence class of ground terms (variable-free expressions) in the language of the data domain. In any interesting domain, every equivalence class contains an infinite number of terms. For example, consider the abstract data domain consisting of the integers with primitive operations {0, suc, pred}. In a correct algebraic specification of this domain, the number 1 is the infinite set of terms

{suc(0), suc(pred(suc(0)), pred(suc(suc(0), ...} .

Although algebraic specifications are syntactically simpler than conventional first order axiomatizations, they share the weaknesses of axiomatic methods enumerated above. Moreover, algebraic specification suffers from a peculiar problem of its own. Although the method is supposed to provide a simple formal framework for reasoning about abstract data domains, it actually imposes an awkward deductive system on the programmer. The source of the problem is the blatant incompleteness of algebraic specifications as conventional first order axiomatizations. It is easy to verify that the trivial model containing a single object (in a sorted system, a single object of each sort) satisfies any

[2] Let E be a equivalence relation on the set S. For any $s \in S$, let $\langle s \rangle$ denote the equivalence class (under E) containing s. An extensional treatment of an equivalence class $\langle s \rangle$ ignores the particular element s used to denote the class. If $(s_1, s_2) \in E$ then $\langle s_1 \rangle$ and $\langle s_2 \rangle$ have the same meaning. In contrast, an intensional treatment of equivalence classes distinguishes between different descriptions for the same class.

axiomatization consisting solely of equations. Consequently, it is impossible to prove anything interesting about an abstract data domain defined by an algebraic specification by using ordinary first order deduction (since any such deduction must hold for the trivial model). Similarly, it is impossible to express mathematical induction within an algebraic specification (because neither quantification nor axiom schemes are allowed).

As a result, reasoning about data domains defined by algebraic specification is a more complex process that simply applying first order deduction to the equations comprising the specification. The most attractive solution to the problem is to augment all algebraic specifications with mechanically generated implicit axioms. The axioms must include some statement that excludes the trivial model (such an inequation asserting that the 0-ary operations **true** and **false** are distinct)[3] and a special form of induction schema commonly called data type or generator induction [Spitzen and Wegbreit 75]. The most interesting (and annoying) characteristic of the resulting formal system is the awkward style of induction proof that it forces. In contrast to ordinary structural induction, generator induction inducts on the syntactic structure of object descriptions (ground terms) rather than the structure of abstract data objects themselves. As a result, the generator induction schemes for defined domains invariably include unnecessary premises (corresponding to the multitude of ways to describe the same object). For example, the generator induction axiom schema for the natural numbers with primitive operations {0, pred, suc} has the unusual form

$$\{P(0) \;\&\; \forall x[P(x)\Rightarrow P(suc(x))] \;\&\; \forall x[P(x)\Rightarrow P(pred(x))]\} \Rightarrow \forall x\, P(x)$$

instead of the conventional

$$P(0) \;\&\; \forall x[P(x)\Rightarrow P(suc(x))] \Rightarrow \forall x\, P(x)$$

with one fewer premise. While the two axioms schemes are formally equivalent (if the algebraic specification correctly defines the natural numbers), proofs utilizing the generator induction axiom schema are much longer and more tedious to construct.

[3]Without this inequation, an algebraic specification is guaranteed to be consistent since the trivial model satisfies any algebraic specification. On the other hand, it is useless as a first order axiomatization of the data domain for the same reason.

3. A Constructive Alternative

The constructive approach to defining data domains is based on the premise that a data domain is a set of of symbolic objects and associated operations satisfying the following three constraints:

[1](Finite constructibility). Every data object is constructible in a finite number of steps by composing a finite collection of primitive operations called constructors. Each constructor c_i accepts a fixed number n_i of objects $o_1,\ldots,o_{n_i}$ as arguments and creates the object $c_i(o_1,\ldots,o_{n_i})$. 0-ary constructors function as constants.

[2](Unique constructibility). Every data object is constructible in exactly one way. Hence, two objects $c_i(o_1, \ldots, o_{n_i})$ and $c_j(p_1, \ldots, p_{n_j})$ are equal if and only if $i = j$ and $o_k = p_k$ for $k = 1,\ldots,n_i$.

[3](Explicit definability). Every operation -- excluding a small set of primitive functions and predicates serving as building blocks -- is explicitly defined by either a recursive function definition or an explicit predicate definition (in the usual sense in first order logic [Enderton 72]). Unlike arbitrary axiomatic extensions which may be contradictory, explicit definitions always create well-defined operations.

A simple example of a constructive data domain definition is the construction of the natural numbers from the 0-ary constructor 0 and the unary constructor suc:

0, suc(0), suc(suc(0)), ...

A constructive definition of the natural numbers may also include other operations such as predecessor, addition, multiplication. However, none of these operations can be designated as constructors, since that would violate the property of unique constructibility. For example, if pred (the unary predecessor operation) were included in the set of constructors, then there would be infinitely many different ways to construct 0:

0, pred(suc(0)), pred(pred(suc(suc(0)))), ...

In contrast, algebraic specification does not distinguish constructors from other operations in the data domain; the more complicated induction rule required for algebraically specified domains reflects this fact.

The three constraints have some desirable conse-

quences. First, they ensure that the data domain has a simple, relatively complete (in the sense described in [Cartwright and McCarthy 79]) first order axiomatization comparable to Peano's first order axioms for the natural numbers.[4] As a result, simple verification methods relying on structural induction (such as those employed in the Boyer-Moore LISP Verifier [Boyer and Moore 75, 79] and the Stanford TYPED LISP Verifier [Cartwright 76]) are applicable to the data domain.[5] Second, they guarantee the existence of a simple, tangible symbolic representation for the data domain, because each data object is uniquely denoted by the ground term that constructs it. Third, they assure that every ground term composed from functions in the domain can be evaluated.

4. Primitive Operations Defined by Constructive Data Domain Definitions

For a constructively defined data domain to support the recursive definition of arbitrary computable functions over the domain, they must include a small set of basic operations for manipulating the constructed objects. Without additional primitive operations, it is impossible to recursively define arbitrary computable functions on the data domain. Consequently, we adopt the convention that constructive data domain definitions implicitly define a small collection of operations that form a basis for computing arbitrary partial recursive functions.

The following set of primitive operations forms a universal basis set of operations (in the terminology of [McCarthy 63]) for the constructive data domain D formed using the constructors c_i, i=1,...,n with arities p_i, i=1,...,n:

a) The constructor functions c_i, i=1,...,n.

b) The implicitly declared 0-ary Boolean constructor functions **true** and **false**.

c) The set of selector functions s_i^j, j=1,...,p_i; i=1,...,n. The selector function s_i^j extracts the jth component u_j from an object of the form $c_i(u_1, \ldots, u_{p_i})$. If a selector is applied to the wrong form of object (i.e. an object whose outermost constructor does not correspond to the selector), then the result is unspecified.

d) The standard Boolean function **equal**. **equal**(x,y) returns **true** if x and y are identical objects (are constructed in the same way) and **false** otherwise.

e) The standard conditional function **if-then-else**. **if-then-else**(x,y,z) returns y if x is the object **true** and z if x is the object **false**. Otherwise, the result is unspecified.

Since recursive definitions over a constructive domain can diverge on particular inputs, constructive data domain definitions must accomodate divergence. Following the approach developed in [Cartwright and McCarthy 1979], we augment the defined domain with the special object ⊥ denoting divergence (any non-terminating evaluation). Unlike every other object in the domain, the divergent object ⊥ is not a constructible object. The non-constructibility of ⊥ is consistent with its intuitive meaning: there is no way to compute divergence by composing a finite number of operations. Unlike other constants, ⊥ is not considered a 0-ary constructor.

To complete our description of the basic operations created by a constructive data domain definition, we must specify how each primitive operation behaves on the input ⊥. For each of the primitive operations except if-then-else, we naturally extend (in the terminology of [Manna 74]) the operation to D^+ by defining the result to be ⊥ if any argument has the value ⊥. Natural extensions correspond to a call-by-value computation rule for evaluating operation arguments. However, to support arbitrary recursive definitions, we must extend **if-then-else** in a less trivial way (naturally extending **if-then-else** would force every recursive definition to diverge everywhere); we define

$$\textbf{if-then-else}(x,y,z) = \begin{cases} \perp & \text{if } x \text{ is } \perp \\ y & \text{if } x \text{ is } \textbf{true} \\ z & \text{if } x \text{ is } \textbf{false} \end{cases}$$

This definition corresponds to the standard protocol for evaluating conditional expressions.

Augmenting the data domain by ⊥ also allows us to define the meaning of error producing computations. Any erroneous term evaluates to ⊥. Hence, we can extend the meaning of selector functions to the entire domain by adopting the convention that applying a selector to the wrong kind of object produces ⊥. Similarly, applying if-then-else to a triple (x,y,z) where x is not a Boolean value yields ⊥.

Given a constructive data domain definition, it is possible to mechanically generate a Peano-like first order axiomatization for the defined domain

---

[4] For a precise statement of this theorem and a sketch of its proof, see [Cartwright and McCarthy 79].

[5] [Cartwright 76] contains formal proofs of several interesting theorems about constructively defined data domains.

augmented by ⊥. The construction is described in detail in the Appendix.

The major technical obstacle in creating the first order axiomatization is formalizing Boolean operations. Many programming logics (e.g. naive forms of Hoare's logic) attempt to formalize Boolean operations as predicates within the logic. This approach works only if primitive program operations never diverge[5] and the logic bans the recursive definition of non-total (sometimes divergent) operations.[6] Otherwise, it is possible to write contradictory recursive definitions (because no least fixed point exists for the corresponding non-monotonic functional). The following recursive function definition, for example,

```
loop(x) = if loop(x) equal 1 then 0
          else 1
```

is contradictory if **equal** is treated as a predicate.[7]

A related technical obstacle to formalizing **equal** as a predicate is the fact that the equality predicate is not computable on a data domain that includes divergent operations. It is impossible to decide whether two possibly divergent terms denote the same object.

The solution that we advocate is to treat computable Boolean operations in the data domain as functions in the logic. Boolean operations in the data domain are distinct from the standard connectives (e.g. ∧, ∨) and predicate symbols (e.g. =) of first order logic. Consequently, Boolean expressions are terms rather than formulas in the logic. As a result, the meaning of a divergent Boolean expression is ⊥, just like any other divergent expression, preventing contradictory definitions like the equation defining loop above. When **equal** is interpreted as the natural extension of the equality function on the natural numbers, the function loop simply diverges everywhere; no contradiction arises.

Of course, logical predicates and connectives still must appear in the first order axiomatization of the data domain. Furthermore, if programs are annotated with assertions expressed in the first order language of the data domain, then logical predicates will naturally appear in program documentation as well.

For the sake of notational convenience, we let any Boolean expression <expr> abbreviate the corresponding logical formula <expr> = **true**. Since expressions and atomic formulas appear in distinct contexts within logical formulas, this abbreviation does not introduce any ambiguity.

An interesting data domain normally includes primitive operations besides constructors and basis operations -- although their presence does not enlarge the class of computable functions or definable predicates over the domain. The additional operations are standard functions or predicates that the programmer expects to use repeatedly in programs or program documentation. From a practical standpoint, the expanded set of operations constitutes a natural rather than minimal set of building blocks for describing computations on the data domain. For example, most definitions of the data domain consisting of the natural numbers include standard operations like addition, subtraction, multiplication, division, and <.

In the constructive approach to defining data domains, these additional operations are explicitly defined rather than axiomatized. Explicitly defined operations fall into two categories:

1) recursively definable partial functions; and

2) first order definable predicates.

Partial function operations are explicitly defined by recursive definitions expressed in terms of the primitive functions. Kleene's first recursion theorem guarantees the existence of functions defined in this fashion. Furthermore, since the primitive functions are all manifestly computable, all defined partial functions are also computable.

In contrast, predicate operations are defined according to the standard practice in first order logic: a n-ary predicate $P(x_1,\ldots,x_n)$ is introduced as an abbreviation for a formula with no free variables other than $x_1, \ldots, x_n$, and no predicate symbols other than = and previously defined predicates. Predicate definitions may not be recursive; otherwise, contradictory definitions would be possible. Since predicate definitions are merely abbreviations for formulas, they are obviously well-defined. Predicate operations are included in data domains solely for the purpose of expressing properties of the data domain (such as the equality of two -- possibly undefined -- terms). Since they are not computable, they cannot be employed as primitive operations in a program manipulating the data domain.

In contrast to the architects of algebraic specification, we believe that a data domain definition may legitimately include non-computable operations as well as computable ones. While non-

[6] PL/CV [Constable and O'Donnell 78] rigorously develops this approach.

[7] In this context, we must interpret **if-then-else** as a logical connective. See [Manna 74].

computable operations obviously cannot appear in actual programs, they are useful in writing program specifications.

As an illustration, assume that we are defining the data domain consisting of partial recursive functions over the natural numbers. We certainly would like for the data domain to include an (extensional) equality operation even though the operation is not computable. As we demonstrate in the last section of the paper, we can easily accomplish this objective within the framework of constructive data domain definitions.

5. Dividing the Data Domain into Types

A typical program data domain includes several different kinds of objects, e.g. integers, Booleans, strings of characters. If no constraints are placed on the arguments accepted by constructors, the resulting "type-less" data domain will inevitably contain a multitude of irrelevant objects which have no intuitive meaning to the programmer. For example, a data domain including constructors both for the natural numbers and for LISP S-expressions would include objects like suc(cons(A,B)).

We can solve this problem by imposing a type structure on the data domain and constraining the arguments of each constructor to belong to a particular type. In our view, a type is simply a "meaningful" subset of the data domain.[8] To exclude irrelevant objects, we let each constructor designate a distinct elementary type and assign every object with outermost constructor c to the elementary type c. For example, every natural number except 0 (i.e. suc(0), suc(suc(0)), ...) belongs to the elementary type suc since the outermost constructor is suc; the natural number 0, on the other hand, belongs to the elementary type 0. Given this primitive type facility, we can obviously eliminate objects like suc(cons(A,B)) from the data domain by constraining the argument to suc to be of type suc or type 0.

Besides eliminating irrelevant objects, a data type definition facility gives the programmer a formal way to identify intuitively meaningful subsets of the data domain. Even in typeless languages like PURE LISP, virtually all functions and variables appearing in programs have intended domains which are "small" subsets of the entire data domain (e.g. lists of atoms, lists of dotted pairs, non-repeating lists of atoms). If a programming language includes a rich data type definition facility, then the domain of virtually every function and variable appearing in a program will be definable as a data type. As a result, much of the information traditionally stated informally in the program documentation can be formally expressed by type declarations.

To achieve this objective, we need a richer collection of types than the elementary constructor types (all objects formed using a particular constructor) described above.

6. Data Type Definitions

In addition to the constructor type definitions described in the previous section, we propose three other schemes for defining types: union definition, subset definition, and quotient definition. Unlike most programming languages, we do not force types to be disjoint, because that convention forces the data domain to include multiple versions of the same abstract object in cases where the object naturally belongs to several different types (e.g. union types, subrange types). Abstract data type definition facilities are supposed to avoid introducing semantic complications of this form.

6.1 Constructor Definition

A constructor type definition has syntax

constructor <id> (<selector-list>)

where <id> is the name of the constructor and <selector-list> is a (possibly empty) sequence of selector declarations separated by commas. Each selector declaration has the form

<selector-id> : <type-id>

where <selector-id> denotes the name of the selector function (which must unique within the <selector-list>) and <type-id> is the name of any type (including forward references). If the <selector-list> is empty the parentheses may be omitted. For example the type definitions

constructor 0;
constructor suc(pred: natnum)

define the 0 and suc constructors for the natural numbers (assuming we subsequently define natnum as the union of 0 and suc).

6.2 Union Definition

A union type definition has the form

union <id> = <type-list>

where <id> is the name assigned to the defined union

[8] Scott develops a similar point of view in his formalization of data types as lattices [Scott 76].

type and <type-list> is a sequence (containing at least two elements) of previously defined type names separated by the ∪ symbol. The defined type is literally the set union of the types appearing in <type-list>. For example, the type definition

union natnum = 0 ∪ suc

defines natnum as the union of the types 0 and suc. Observe that the data object suc(0) belongs to both the constructor type suc and the union type natnum.

For more readable notation, we allow constructor definitions to appear within the <type-list> defining a union. For example, the types natnum, 0, and suc are defined in the single line:

union natnum = 0() ∪ suc(pred: natnum)

The union type Bool and the 0-ary constructor types **true** and **false** defined by

union Bool = **true**() ∪ **false**()

are pre-defined in every data domain.

6.3 Subset Definition

The data types definable by constructor and union definitions are nearly identical to the recursive types described in [McCarthy 63] and [Hoare 73]. The only difference is that our scheme accomodates overlapping (non-disjoint) types and divergent operations. The most significant limitation of recursive type definitions is that they can only define "context-free" types -- types where any object belonging to the declared type of a particular constructor argument is a valid argument to the constructor regardless of the constructor's context. Consequently, data domains like height-balanced trees or non-repeating sequences of integers are not definable as recursive types.

To accommodate data type definitions that are not context free, we allow arbitrary first order definable subsets of previously defined types to be defined as types. This form of type definition, called subset definition, has the following syntax:

subset <id> = { <var-id> ⁻ <type-id> | <formula> }

where <id> is the name of the new type, <var-id> is a variable name local to the type definition, <type-id> is the name of a previously defined type, and <formula> is a logical formula[9] (with no free variables other than <var-id>) composed from primitive and programmer-defined functions and predicates. The defined type <id> consists of all objects in the the parent type <type-id> satisfying the predicate denoted by <formula>.

While it is clearly impossible for a language processor to enforce subset type declarations by performing run-time checks, a sophisticated processor could easily generate at parse time a collection of lemmas asserting that no run-time type errors occur. Since these lemmas are simple statements in a first order programming logic, it is feasible to actually try to prove them using an interactive program verifier similar in spirit to that described in [Cartwright 76].

We define the type non-repeating sequence of natural numbers (nonrep) as follows:

```
union nat_seq = nil() ∪
   cons(first: natnum, rest: nat_seq);
subset nonrep = {x ⁻ nat_seq | nodup(x)}
```

where the function nodup is defined by the recursive definitions:

```
function nodup(x: nat_seq): Bool ≡
  if x equal nil then true
  else if member(first(x), rest(x)) then false
  else nodup(rest(x))

function member(e: natnum, s: nat_seq): Bool ≡
  if s equal nil then false
  else if e equal first(s) then true
  else member(e, rest(s))
```

6.4 Quotient Definitions

While the ability to designate an arbitrary recursive subset of a recursive data type as a type greatly expands the collection of definable types, it still does not give us the power to define types containing objects that are not uniquely constructible -- such as finite sets, computable functions (viewed extensionally) and finite maps. In the case of finite sets, for example, it is possible to construct a set by accumulating it elements in any order. Although we can obviously represent finite sets by the constructive type consisting of non-repeating sequences, the inability to define a bona fide set type is a serious deficiency in our data domain definition facility.

At first glance, we appear to be in a quandary. Our constructive approach to data domain definition relies on the principle of unique constructibility, yet some important abstract types -- such as finite sets -- contain objects that violate this principle.

[9] Recall that we allow an expression <expr> to abbreviate the logical formula <expr> = **true**

The solution to the problem is to add a type construction mechanism that generates types whose members are equivalence classes of conventional constructible objects. An abstract object that is not uniquely constructible is treated as the equivalence class of different constructions corresponding to the object. For example, each distinct way to construct a finite set is naturally represented by a distinct sequence of elements. The finite set $\{x_1, x_2, \ldots, x_n\}$ is formally defined as the equivalence class of sequences containing precisely the elements $x_1, x_2, \ldots, x_n$.

To avoid violating the principle of unique constructibility, we treat equivalence classes intensionally in the formal logic and symbolic model for data domain. The model assigns distinct interpretations to distinct descriptions of the same equivalence class.

From the programmer's viewpoint, equivalence class data objects behave extensionally -- unless he specifically chooses to exploit the underlying intensional semantics.

The syntax for a quotient type definition is:

quotient <id> = <type-id> **under** <equiv-id>

where <id> is the name of the defined quotient type, <type-id> is the type on which the equivalence relation is being defined, and <equiv-id> is either the name of a binary function (mapping <type-id> × <type-id> into Bool) or a binary predicate (over <type-id> × <type-id>) defining the equivalence relation on <type-id>.

In the formal semantic definition, the quotient type name <id> is a unary constructor with domain <type-id>. Applying the constructor <id> to an object x of type <type-id> constructs an object denoting the equivalence class containing x under the equivalence relation <equiv-id>. The reserved identifier **intension** is the name of the selector corresponding to all quotient constructors. Given an equivalence class object <id>(y), **intension** extracts the particular element y denoting the class.

In order to support equivalence classes as legitimate objects, we modify the definition of the primitive equality predicate = and function **equal** as follows. If t_1 and t_2 are both objects of type <id>, then the atomic formula $t_1 = t_2$ is interpreted as an abbreviation for t_1 <equiv-id> t_2.[10] Furthermore, if <equiv-id> is a function name, then t_1 **equal** t_2 is interpreted as an abbreviation for t_1 <equiv-id> t_2. If <equiv-id> is a predicate name, then **equal** diverges when applied to two objects of type <id>.

To guarantee the integrity of the informal interpretation of quotient type definitions, the programmer must prove that <equiv-id> defines an equivalence relation on the parent type <type-id>. As in the case of subset type definitions, a sophisticated language implementation could automatically generate the necessary first order lemma and interactively help the programmer prove it.

As an illustration of quotient definition, let us define the quotient type consisting of finite sets of natural numbers.

```
quotient nat_set = nonrep under set_equal;
```

where the function set_equal is defined by the recursive definitions:

```
function set_equal(x:nat_set, y:nat_set):Bool ≡
  seq_equal(intension(x), intension(y))

function seq_equal(u:nat_seq, v:nat_seq):Bool ≡
  if u equal nil then v equal nil
  else if member(first(u), v) then
    seq_equal(rest(u), delete(first(u), v))
  else false

function delete(e: natnum, s: nat_seq): Bool ≡
  if s equal nil then nil
  else if e equal first(s) then rest(s)
  else cons(first(s), delete(e, rest(s)))
```

The function set_equal is formally defined as a function on the constructor type nat_set. However, since the answer does not depend on which intension (sequence) is used to denote a set, it is also a function on the extensional set objects denoted by the intensional sequence descriptions. In the next section, we explore how to interpret operations on quotient types that depend on the particular intension used to describe the quotient object.

7. Explicitly Defined Operations

As we have already illustrated in preceding examples, explicit function definitions have the following form:

function <id> (<parameters>):<type-id> ≡ <expr>

where <id> is the name of the explicitly defined function, <parameters> is a (possibly empty) list of parameter declarations separated by commas, <type-

[10] Recall that we allow an expression <expr> to abbreviate the logical formula <expr> = **true** .

id> is is the range (output) type of the function, and <expr> is an expression containing no free variables others than parameters. Each parameter declaration has the form

<parm-id>: <type-id>

where <parm-id> is the name of the parameter and <type-id> is the declared type for the parameter.

Explicitly defined predicates have the following analogous format:

predicate <id> (<parameters>) ⇔ <formula>

where <id> is the name of the defined predicate, <parameters> is a list of parameter declarations separated by commas, and <formula> is a first order formula with no free variables other than parameters.

Explicit predicate definitions are much less common in data domain definitions than explicit function definitions because they denote non-computable operations. They are useful in type definitions and program documentation. For example, assume that we have already defined the type func consisting of abstract syntax representations of functions defined by lambda expressions over the integers. In addition, assume that we have explicitly defined the function

function apply(f:func, x:integer):int-or-func ≡ ...

that applies functions to integers. Then the following predicate definition captures extensional equality between function representations.

predicate func-equal(f: func, g:func) ⇔
∀ x isinteger(x) ⇒ apply(f,x) = apply(f,y)

In order for the programmer to define interesting basic operations on a quotient type q, he must explicitly access the intensional descriptions embedded in the objects belonging to type q. Consequently, if the programmer defines a function on a quotient type and uses the primitive operation intension within the function body, he must prove that the function is well-defined on the quotient type.

If he makes a mistake and defines a function which is not well-defined from an extensional viewpoint, the function has a clear intensional meaning which the language implementation will compute.

In many cases, an operation which is not extensionally well-defined has an intuitively clear extensional interpretation as a non-deterministic operation. For example, the primitive operation intension selects an arbitrary element of the equivalence class denoted by the input object. A more useful example occurs in the context of the quotient type nat_set. In this case, the operation

λ q . first(**intension**(q))

selects an arbitrary element from the nat_set q -- the first element of the particular sequence describing q.

To accommodate non-deterministic operations, we allow programmers to define non-deterministic operations on quotient types as well as functions. For the sake of clarity, however, we force programmers syntactically to distinguish non-deterministic function definitions from standard deterministic ones. Non-deterministic function definitions must use the keyword **multifunction** instead of **function**. For example, the definition of the choose operation described above has the following form:

multifunction choose(s:nat_set):natnum ≡
first(**intension**(s))

In the formal semantic definition, there is absolutely no distinction between functions and multifunctions, since they are both ordinary functions at the intensional level. Consequently, an interpreter or compiler supporting quotient type definitions would treat multifunctions and functions identically. The difference between non-deterministic operations and functions lies solely in what they implicitly assert. In contrast to a convention interpreter or compiler, a sophisticated language processor/verifier would treat functions and multifunctions quite differently. A function definition would automatically generate a well-definedness lemma; a multifunction would not. Similarly, a verifier could apply special derived rules (concerning extensional equality) to terms containing no non-deterministic functions.[11]

8. An Interesting Example

In this section, we present a non-trivial data type definition -- priority queues -- that illustrates the constructive approach to data type definition. In the following definition, a priority queue is a data object containing entries that are pairs consisting of an integer item and an integer priority. Duplicate items and entries may appear in the queue.

[11] For example, given the statement u = v for objects u and v of the same quotient type, the verifier could deduce that s(u) = s(t) for any term s(x) free of non-deterministic functions.

We assume that the type integer has already been defined including the usual Boolean function <.

```
constructor
    entry(item: integer, priority: integer)

union seq = nil ∪ cons(hd: entry; tl: seq)

subset pri-queue = {s ⁻ seq | ordered(s)}

function ordered(s: seq): Bool ≡
  if s equal nil then true
  else if tl(s) equal nil then true
  else if priority(hd(p)) ≥ priority(hd(tl(p)))
    then ordered(tl(p))
  else false

function
  insert(e: entry; p: pri-queue): pri-queue ≡
  if p equal nil then cons(e,nil)
  else if priority(e) > priority(hd(p))
    then cons(e,p)
  else cons(hd(p), insert(e,tl(p)))

function remove(p: pri-queue): pri-queue ≡ tl(p)

function front (p: pri-queue): entry ≡ hd(p)
```

The definition formalizes a priority queue as a sequence of entries sorted in order of descending priority. Other formalizations are obviously possible. For example, we could formalize a priority queue as an unordered sequence of entries and force front to search the queue for the highest priority item. Still another formalization would treat a priority queue as a multi-set of entries.

9. Ambiguity in Data Domain Definitions

At first glance, the existence of several fundamentally different formalizations for an abstract data type like priority queues is disconcerting. Abstract data type definitions are supposed to avoid introducing unnecessary detail; yet the constructive definition of the type priority queue clearly involves arbitrary design choices. Is it possible that axiomatic methods are superior to the constructive approach in this regard?

The answer to this question is surprisingly subtle. For algebraic specifications, as defined by both the ADJ group and Guttag, the answer is no. For more general axiomatic methods, however, the answer is yes.

The three different formalizations of priority queues described above correspond to different equality relations on ground terms denoting priority queues. Any axiomatic method -- such as algebraic specification -- that implicitly defines a specific term model[12] unambiguously specifies an equality relation on ground terms. Consequently, the same arbitrary design decisions that arise in devising a constructive definition also arise in developing a comparable algebraic specification.

On the other hand, axiomatic methods -- such as ordinary first order axiomatizations -- that do not identify a unique term model can define an ambiguous equality relation on terms denoting objects of the defined type. Different models for the axiomatization may incorporate different equality relations. However, writing axiomatic definitions with exactly the right degree of ambiguity is a very subtle problem. The interested reader is encouraged to try axiomatizing priority queues as an example. We believe that the advantages of constructive definitions far outweigh their inability to accomodate this form of ambiguity.

10. Directions for Further Work

A major weakness in the constructive data domain definition facility described in this paper is the absence of parameterized types. For example, to define several distinct sequence types (with different element types) a data domain definition must either

1) separately define each sequence type; or

2) define a general sequence type that places no restrictions on the element type (by defining a universal type that is the union of all constructor types in the domain) and define each particular sequence type as a subset type.

In either case, the definition is less convenient and perspicuous than the corresponding parameterized definition. Extending constructive data domain definitions to support parameterized types is a major focus of our current research.

Another promising extension to constructive data domain definitions is accommodating lazy (evaluation) constructors. Lazy constructors have radically different semantics than their call-by-value counterparts. With lazy evaluation, recursively enumerable sequences and sets can be treated as genuine data objects. Partial recursive functions, for example, can be represented by recursively enumerable graphs instead of program text (abstract

[12] If algebraic specifications are interpreted as specifying any model in the lattice of term models satisfying the specification instead of one particular model (e.g the initial algebra designated by ADJ group), then algebraic specifications accomodate ambiguity like general first order axiomatizations.

syntax). The major obstacle to including this extension is developing a simple programming logic to handle infinite objects.

References

ADJ (J. Goguen, J. Thatcher, E. Wagner) (1976) An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types, IBM Research Report RC 6487.

ADJ (J. Goguen, J. Thatcher, E. Wagner, J. Wright) (1977) Initial Algebra Semantics and Continuous Algebras, JACM 24, 68-95.

Boyer, R. and J Moore. (1975): Proving Theorems about LISP Functions, J. ACM 22(1), 129-144.

Cartwright, R. (1976): User-Defined Data Types as Aid to Verifying LISP Programs, in S. Michaelson and R. Milner (eds.), Automata Languages, and Programming, pp. 228-256, Edinburgh Press, Edinburgh.

Cartwright, R. and J. McCarthy (1979): First Order Programming Logic, Proc. Sixth Annual Symposium on Principles of Programming Languages, January 1979, pp. 68-80.

Constable, R. and M. O'Donnell (1978): A Programming Logic, Winthrop Publishers, Cambridge, Massachusetts.

Enderton, H. B. (1972): A Mathematical Introduction to Logic, Academic Press, New York.

Guttag, J. V. et. al. (1976): Abstract Data Types and Software Validation, USC Information Sciences Institute Technical Report ISI/RR-76-48.

Guttag, J. V. (1977): Abstract Data Types and the Development of of Data Types, Acta Informatica 10, 27-52.

Guttag, J. V. and J. J. Horning (1978): The Algebraic Specification of Data Structures, Comm. ACM 20, 396-404.

von Henke, F. and D. C. Luckham (1974): Automatic Program Verification III: A Methodology for Verifying Programs, Stanford Artificial Intelligence Project Memo AIM-256.

Hoare, C. A. R. (1972): Proofs of Correctness of Data Representation, Acta Informatica 1, 271-281.

Hoare, C. A. R. (1973): Recursive Data Types, Stanford Artificial Intelligence Project Memo AIM-223.

McCarthy, J. (1963): A Basis for a Mathematical Theory of Computation, in P. Braffort and D. Hirschberg (eds.), Computer Programming and Formal Systems), pp. 33-70. North-Holland Publishing Company, Amsterdam.

Musser, D. R. (1980): On Proving Inductive Properties of Abstract Data Types, Proc. Seventh Annual Symposium on Principles of Programming Languages, January 1980, pp. 154-162.

Scott, D. (1976): Data Types as Lattices, SIAM J. Comput. 5, 522-586.

Spitzen, J. and B. Wegbreit (1975): The Verification and Synthesis of Data Structures, Acta Informatica 4, 127-144.

APPENDIX

Constructive Data Domain Definitions as First Order Axiomatizations

Let D be an arbitrary data domain definition. We will construct a first order axiomatization for D. For the sake of simplicity, we will not present a relatively complete axiomatization. The weakness in our axiomatization concerns divergent computations, so it is not very important from a practical viewpoint. In general, we have omitted axioms that specify how functions and predicates behave on $\perp$.

A. Primitive Functions and Predicates

Every data domain includes the constants $\perp$, **true**, and **false**; the unary functions isBool, **istrue**, **isfalse**; the binary infix functions **eq**, **equal**, and $\subset$; the ternary function **if-then-else**; and the binary predicate operations $\equiv$ and $=$.

The function **eq** is the computable intensional equality function; two objects are intensionally equal if and only if they are constructed in exactly the same way. The function **equal**, on the other hand, is the computable extensional equality function. In comparisons between objects that do not contain quotients, the two functions **eq** and **equal** behave identically. On quotient objects, however, the definition of **equal** depends on the equivalence relation defining the quotient.

The function $\subset$ is the Boolean structural containment function; $x \subset y$ if and only if x is a proper structural component of y. More precisely, for any constructed object y of the form $c(x_1,\ldots,x_n)$, $x_i \subset y$. Furthermore, $\subset$ is closed under transitivity. The function $\subset$ is included as a primitive function, because it is used in the formal statement of course-of-values structural induction.

The predicate symbol $\equiv$ denotes the standard equality predicate from first order logic. Since our data

domain model is intensional, $\equiv$ corresponds to intensional equality. In contrast, the predicate symbol = denotes extensional equality.

The following axioms specify the primitive operations:

1. Intensional equality.[13] For any formula $\theta(x)$,

$$x \equiv y \Rightarrow [\theta(x) \Leftrightarrow \theta(y)]$$

2. Distinctness of Primitive Objects

$$\textbf{true} \not\equiv \textbf{false}, \quad \perp \not\equiv \textbf{true}, \quad \perp \not\equiv \textbf{false}$$

3. Definition of **eq** in terms of $\equiv$.

$$x \equiv y \wedge x \not\equiv \perp \Rightarrow x \textbf{ eq } y = \textbf{true}$$
$$x \not\equiv y \wedge x \not\equiv \perp \wedge y \not\equiv \perp \Rightarrow x \textbf{ eq } y = \textbf{false}$$
$$\perp \textbf{ eq } y = \perp, \quad x \textbf{ eq } \perp = \perp$$

4. Definition of **if-then-else.**

$$\textbf{if true then } y \textbf{ else } z \equiv y$$
$$\textbf{if false then } y \textbf{ else } z \equiv z$$

5. Definition of characteristic functions.

$$\text{isBool}(x) \equiv \textbf{if } x \textbf{ equal true then true else } x \textbf{ equal false}$$
$$\textbf{istrue}(x) \equiv x \textbf{ equal true}$$
$$\textbf{isfalse}(x) \equiv x \textbf{ equal false}$$

6. Partial definition of **equal** in terms of **eq.**

$$\text{isBool}(x) \vee \text{isBool}(y) \Rightarrow x \textbf{ equal } y \equiv x \textbf{ eq } y$$

7. Partial definition of = in terms of $\equiv$.

$$\text{isBool}(x) \vee \text{isBool}(y) \vee x=\perp \vee y=\perp \Rightarrow x=y \Leftrightarrow x \equiv y$$

8. Partial definition of $\subset$.

$$x \not\equiv \perp \wedge \text{isBool}(y) \Rightarrow x \subset y \equiv \textbf{false}$$
$$x \subset y \wedge y \subset z \Rightarrow x \subset z$$

9. Structural induction. For any formula $\theta(x)$

$$\forall x\, [\forall y\, (y \subset x \Rightarrow \theta(y)) \Rightarrow \theta(x)] \Rightarrow \forall x\, \theta(x)$$

[13] In first order predicate calculus with equality, this axiom scheme is a built-in "logical" scheme.

B. Domain Dependent Objects and Operations

Every definition in D extends the domain by adding new data objects or operations. For definitions that create new objects, we must describe how primitive operations behave on the new objects. For definitions that create new operations, we must specify the new operations in terms of primitive ones.

The axioms generated by a constructor definition

constructor $c(s_1:t_1, \ldots, s_n:t_n)$

(excluding quotients) appear below:

10. Definition of selectors. For i=1,...,n:

$$\text{ist}_1(x) \wedge \ldots \wedge \text{ist}_n(x) \Rightarrow s_i(c(x_1,\ldots,x_n)) \equiv x_i$$

11. Definition of characteristic function.

$$\text{ist}_1(x) \wedge \ldots \wedge \text{ist}_n(x) \Rightarrow \text{isc}(c(x_1,\ldots,x_n))$$
$$\text{isc}(\perp) \equiv \perp, \quad \text{isc}(x) \Rightarrow c(s_1(x),\ldots,s_n(x)) \equiv x$$

12. Disjointness. For any other constructor q:

$$\text{isq}(x) \Rightarrow \text{isc}(x) \equiv \textbf{false}$$

13. Extension of primitive operations.

$$\text{isc}(x) \vee \text{isc}(y) \Rightarrow [x = y \Leftrightarrow x \equiv y]$$
$$\text{isc}(x) \vee \text{isc}(y) \Rightarrow x \textbf{ equal } y \equiv x \textbf{ eq } y$$
$$\text{isc}(y) \Rightarrow [x \subset y \Leftrightarrow (x \textbf{ eq } s_1(y) \vee x \subset s_1(y)) \vee \ldots \vee (x \textbf{ eq } s_n(y) \vee x \subset s_n(y))]$$

Each union definition

union $t = t_1 \cup \ldots \cup t_n$

generates the following axiom:

14. Definition of characteristic function.

$$\text{ist}(x) \equiv \textbf{if ist}_1(x) \textbf{ then true else if } \ldots \textbf{ else ist}_n(x)$$

Each quotient definition

quotient $q = t$ **under** r

generates the same axioms (10-13) as the definition

constructor q(**intension**: t)

except for the axioms extending the operations = and **equal**. For a quotient under a function r, the following axiom extends the definition of **equal**:

15. Definition of **equal** on quotient type q.

$$\text{isq}(x) \wedge \text{isq}(y) \Rightarrow x \textbf{ equal } y \equiv r(x,y)$$

For any quotient, the following axiom extends = :

16. Definition of = on quotient type q.

$$\text{isq}(x) \wedge \text{isq}(y) \Rightarrow [x = y \Leftrightarrow r(x,y)]$$

Explicit definitions of operations are interpreted directly as axioms. A (multi)function definition

(multi)function $f(x_1: t_1, \ldots, x_n: t_n) = t$

generates the following axiom:

17. Axiom for a **(multi)function** definition.

$$\text{ist}_1(x_1) \wedge \ldots \wedge \text{ist}_n(x_n) \Rightarrow f(x_1,\ldots,x_n) \equiv t$$

Similarly a predicate definition

predicate $p(x_1: t_1, \ldots, x_n: t_n) \Leftrightarrow \theta$

translates into the following axiom:

18. Axiom for a **predicate** definition.

$$\text{ist}_1(x_1) \wedge \ldots \wedge \text{ist}_n(x_n) \Rightarrow [p(x_1,\ldots,x_n) \Leftrightarrow \theta]$$