MIT/LCS/TR-237

TOWARDS A THEORY FOR ABSTRACT DATA TYPES

Deepak Kapur

This blank page was inserted to preserve pagination.

TOWARDS A THEORY FOR ABSTRACT DATA TYPES

DEEPAK KAPUR

Copyright Massachusetts Institute of Technology 1980

May 1980

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-75-C-0661, and in part by the National Science Foundation under grant MCS'74-21892'A01.

Massachusetts Institute of Technology
Laboratory for Computer Science

Cambridge **Massachusetts 02139**

This empty page was substituted for a blank page in the original document.

Towards a Theory for Abstract Data Types

Abstract

A rigorous framework for studying immutable data types having nondeterministic operations and operations exhibiting exceptional behavior is developed. The framework embodies the view of a data type taken in programming languages, and supports hierarchical and modular structure among data types.

The central notion in this framework is the definition of a data type. An algebraic and behavioral approach for defining a data type is developed which focuses on the input-output behavior of a data type as observed through its operations. The definition of a data type abstracts from the representational structure of its values as well as from the multiple representations of the values for any representational structure.

A hierarchical specification language for data types is proposed. The semantics of a specification is a set of related data types whose operations have the behavior captured by the specification. A clear distinction is made between a data type and its specification(s). The normal behavior and the exceptional behavior of the operations are specified separately. The specification language provides mechanisms to specify (i) a precondition for an operation thus stating its intended inputs, (ii) the exceptions which must be signalled by the operations, and (iii) the exceptions which the operations can optionally signal. Two properties of a specification, consistency and behavioral completeness, are defined. A consistent specification is guaranteed to specify at least one data type. A behaviorally complete specification 'completely' specifies the observable behavior of the operations on their intended inputs.

A deductive system based on first order multi-sorted predicate calculus with identity is developed for abstract data types. It embodies the general properties of data types, which are not explicitly stated in a specification. The theory of a data type, which consists of a subset of the first order properties of the data type, is constructed from its specification. The theory is used in verifying programs and designs expressed using the data type. Two properties of a specification, well definedness and completeness, are defined based on what can be proved from it using different fragments of the deductive system. The sufficient completeness property of Guttag and Horning is also formalized and related to the behavioral completeness property. The well definedness property is stronger than the consistency property, because the well definedness property not only requires that the specification specifies at least one data type, but also captures the intuition that it preserves other specifications used in it thus ensuring modular structure among specifications. The

completeness property is stronger than the sufficient completeness property, since in addition to the requirement that the behavior of the observers can be deduced on any intended input by equational reasoning, it also requires that the equivalence of the observable effect of the constructors can be deduced from the specification by equational reasoning.

A correctness criterion is proposed for an implementation coded in a programming language with respect to a specification. It is defined as a relation between the semantics of an implementation and the semantics of a specification. It does not require a correct implementation to have the maximum amount of nondeterminism specified by a specification. A methodology for proving correctness of an implementation is developed which embodies the correctness criterion.

Name and Title of Thesis Supervisor: Barbara H. Liskov
Associate Professor of Electrical Engineering
and Computer Science

Key Words and Phrases: Abstract Data Type, Data Type, Data Abstraction, Type Algebras, Nondeterminism, Exceptions, Specification Language, Semantics, Consistency, Behavioral Completeness, Deductive System, Verification, Proof Technique, Sufficient Completeness, Completeness, Well Definedness, Implementation Correctness

---

This report is a minor revision of a thesis of the same title submitted to the Department of Electrical Engineering and Computer Science in March, '80 in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Acknowledgments

I am thankful to my thesis supervisor, Professor Barbara Liskov, for her patience and encouragement during the thesis research and especially during the later stages; to Professor John Guttag for posing many challenges and for many suggestions leading to improvements in the presentation of the thesis; to Professor Carl Hewitt for helping me organize and present my ideas in the early stage of the research; and to Professor Hal Abelson for diligently reading the final draft and making many helpful comments.

My officemates, Valdis Berzins, Srivas Mandayam, and Carl Seaquist have helped me in many ways during the thesis research. They gave me an audience whenever I needed, helped me organize my ideas, and found time to read my work whenever I asked them irrespective of their other important responsibilities. Carl and Srivas provided a very stimulating and encouraging atmosphere during the last year. I am also thankful to Russ Atkinson, Moms Krishnamurthy, Dave Musser, Gene Stark, and Jeannette Wing for their helpful comments. Eliot Moss is to be thanked for producing and maintaining the software necessary for the production of this document.

The graduate study at MIT has provided me a unique opportunity to live outside of my own country which has been a tremendous learning experience. Besides computer science, I have learnt a great deal about life, this country, my country, and myself, which has fundamentally changed my attitude and outlook towards life. For this, I am indebted to the students and staff of the Seminar on International Students and Their Participation in Development, and my friends, especially Arvind, Ashok, Carl, Kanchan, Krishna, Mukundan, Nagu, Ravi, Rashid, Sekhar, Srivas, Vaqar, and Vinod. Without their encouragement and interest, continuing the thesis research would not have been possible. Roli has contributed to the completion of the thesis in her own unique way; in no way can I adequately express my gratitude to her.

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-75-C-0661, and in part by the National Science Foundation under grant MCS'74-21892'A01.

This empty page was substituted for a blank page in the original document.

Table of Contents

1. Introduction ..... 9.

1. Scope and Approach of the Thesis ..... 11.

1. Scope and Assumptions ..... 11.
2. Definition of a Data Type ..... 11.
3. Specification Method ..... 13.
4. Deductive System ..... 17.
5. Correctness of Implementation ..... 18.

2. Related Work ..... 19.
3. Outline of the Thesis ..... 22.

2. Definition of an Abstract Data Type ..... 23.

1. Informal Description of a Data Type ..... 26.

1. Terminology ..... 26.
2. Hierarchical Structure ..... 28.
3. Minimality Property ..... 29.

2. Formalism ..... 31.

1. Type Algebras ..... 32.
2. Examples of Type Algebras ..... 35.
3. Interpretation of Terms ..... 37.
4. Observable Behavior ..... 39.
    1. Definitions of Observable Equivalence and Distinguishability ..... 41.
    2. Reduced Algebras ..... 45.
5. Behavioral Equivalence of Type Algebras ..... 45.
6. Definition of a Data Type ..... 49.
7. Observable Equivalence and Distinguishability of Terms ..... 51.

3. Exceptional Behavior of a Data Type ..... 53.

1. Assumptions about Exception Handling Mechanism ..... 53.
2. Formalism ..... 56.
    1. Terms, Exception Terms, and Interpretations ..... 57.
    2. Examples of Modified Type Algebras ..... 58.
    3. Observable Behavior and Distinguishability ..... 59.
    4. Comparison with Goguen's Approach ..... 62.
3. A Simpler Approach ..... 63.

4. Mutually Recursive Data Types ..... 66.

3. Specification of an Abstract Data Type ..... 68.

1. Specification Language ..... 72.
1. Operations ..... 73.
2. Auxiliary Functions ..... 74.
3. Restrictions ..... 77.
1. Preconditions ..... 77.
2. Exception Conditions ..... 79.
3. Discussion ..... 80.
4. Axioms ..... 81.
5. Specifying Nondeterministic Operations ..... 83.
6. Specification of Mutually Recursive Data Types ..... 85.

2. Semantics of Specification Language ..... 86.
1. Specifications without Auxiliary Functions ..... 87.
1. Restrictions ..... 88.
2. Axioms ..... 89.
2. Specifications with Auxiliary Functions ..... 91.
3. Semantics of a Specification ..... 92.

3. Specification of a Data Type and Equivalence of Specifications ..... 94.
4. Specification of Bool ..... 98.
5. Properties of a Specification ..... 99.
1. Consistency ..... 99.
2. Behavioral Completeness ..... 102.
1. Partial Isomorphic Equivalence ..... 103.
2. Isomorphic Embeddability ..... 104.
3. Partial Isomorphic Embeddability ..... 106.
4. Definition of Behavioral Completeness ..... 106.

6. Comparison With Related Works ..... 109.

4. Deductive System ..... 112.

1. Preliminaries ..... 115.
2. Theory of Data Types without Nondeterminism and without Exceptional Behavior ..... 119.
1. Derivation of Nonlogical Axioms ..... 121.
2. Equational Subtheory ..... 122.
3. Distinguishability Subtheory ..... 123.
4. Inductive Subtheory ..... 124.
1. Infinite Induction Rule ..... 125.
2. Rationale for an Infinite Induction Rule ..... 126.
3. Use of the Induction Rule ..... 128.
4. Specifications with Nontrivial Preconditions for Constructors ..... 131.
5. The Full Theory ..... 134.
6. Properties of a Specification ..... 136.

1. Sufficient Completeness ..... 138.
2. Completeness ..... 141.
3. Well Definedness ..... 142.

7. Automation of IND(S) ..... 143.

3. Theory of Exceptions Without Nondeterminism ..... 144.

1. Derivation of Nonlogical Axioms ..... 145.
1. Restrictions Component ..... 145.
2. Axioms Component ..... 146.
3. Definition of $N?_D$ ..... 147.

2. Equational Subtheory ..... 149.
3. Distinguishability Subtheory ..... 151.
4. Inductive Subtheory ..... 152.
5. The Full Theory ..... 153.
6. Properties of a Specification ..... 157.
1. Sufficient Completeness ..... 158.
2. Completeness and Well Definedness ..... 160.

4. Theory of Nondeterminism ..... 161.

1. Transformation Procedure TR ..... 163.
2. Th(S) ..... 167.
3. Data Types with Exceptional Behavior ..... 168.
4. Properties of a Specification ..... 173.

5. Strong Equivalence of Specifications ..... 175.

5. Correctness of Implementation ..... 176.

1. Correctness Criterion and Overview of Correctness Method ..... 178.
1. Semantics of an Implementation ..... 179.
2. Correctness Method ..... 181.
1. Nondeterminism ..... 182.
2. Definition of Correctness ..... 185.

2. Implementation Structure and Semantics ..... 187.

1. Procedures - Approach I ..... 188.
2. Procedures - Approach II ..... 188.
3. Properties of the Encapsulation Mechanism ..... 191.
4. Semantics of an Implementation ..... 195.

3. Correctness Method ..... 196.

1. Auxiliary Functions in a Specification ..... 196.
2. Preservation of Inv ..... 196.
3. Termination of Procedures ..... 197.
4. Proving Restrictions and Axioms ..... 197.
1. Preservation of Equivalence Relation ..... 198.
2. Restrictions ..... 199.
3. Axioms ..... 201.

5. Nondeterministic Procedures ..... 202.

6. Pseudo-Nondeterministic Procedures ..... 203.

4. Recursive and Mutually Recursive Implementations .. 205.

1. Recursive Implementations ..... 205.
2. Mutually Recursive Implementations ..... 209.

6. Conclusions ..... 210.

1. Summary of Contributions ..... 210.
2. Directions for Further Research ..... 212.

References ..... 216.

Appendix I. Elaboration of Scope and Assumptions 224.

1. Immutable and Mutable Data Types ..... 224.
2. Exceptional Behavior ..... 225.
3. Nondeterminism ..... 225.

Appendix II. Definitions of Algebraic Concepts and Proofs of Theorems in Chapter 2 ..... 227.

1. Congruence, Homomorphism, and Isomorphism ..... 227.
2. Proof of Theorem 2.2 ..... 229.
3. Elaboration of the Definition of Behavioral Equivalence and Proofs of Theorems 2.5 and 2.6 ..... 230.

Appendix III. Proofs of Theorems in Chapter 4 ..... 236.

Appendix IV. Specifications of Data Types used in Chapter 5 ..... 246.

1. Introduction

The role of abstraction, modularity and hierarchical structure has been well recognized in the literature on program design and construction [12, 66, 73]. Data abstraction, in particular, has been found to be a useful abstraction mechanism in the design and construction of well structured programs [51].[1] Most of the recent programming languages encourage the use of abstract data types by providing an encapsulation mechanism for implementing them [65, 49, 52, 75, 45, 1]. It is necessary to develop a rigorous foundation for abstract data types so that the informal concept of an abstract data type can be placed on a firm and sound basis, and various aspects of this concept can be studied and analyzed.[2]

In this thesis, we develop a framework for abstract data types. The central notion in this framework is the *definition* of an abstract data type. We develop a *behavioral* method for defining a class of abstract data types, called *immutable* data types [49, 52]. An immutable data type is defined as a set of *behaviorally equivalent algebras* having interpretations for the values and the operations of the data type. Behaviorally equivalent algebras have the same behavior as observed through their operations. We propose a specification language for abstract data types. The semantics of a specification is a set of related data types sharing the common behavior captured by the specification. We make a clear distinction between a data type and its specification(s). We develop a deductive system for abstract data types embodying their general properties which are not explicitly stated in a specification. We use the deductive system to prove properties of an abstract data type from its specification. We propose a correctness criterion for an implementation of an abstract data type with respect to its specification, and develop a methodology for proving correctness of an implementation with respect to a specification which embodies the proposed criterion.

---

1. The terms abstract data type, data type, data abstraction, and type are used synonymously in this thesis.
2. Liskov and Zilles [47] emphasize the need for rigorously developing the mathematical foundation of the specification methods for abstract data types.

The main contribution of this research is a framework for abstract data types that is rigorous and that brings together various aspects of abstract data types in a unified and coherent way. Our approach is better than other similar attempts, in particular the initial algebra formalism of the ADJ group [23] and the category theory formalism of Goguen [20, 7, 30], because it is more in tune with the way programming languages support the mechanism of abstract data type. The framework incorporates important and useful features such as hierarchical structure and modularity. It is also broader in scope as it handles data types with nondeterministic operations and with operations exhibiting exceptional behavior. We had originally developed the framework without considering nondeterminism and exceptional behavior; however, we did not encounter any major difficulties in extending it to incorporate nondeterminism and exceptional behavior. This makes us believe that our framework is robust and extensible for studying other aspects of data type behavior not discussed in this thesis.

Our framework will be useful to a designer of a specification language for abstract data types as it provides a semantic basis for studying and comparing such specification languages. It can be used to define the semantics of a specification language. It also provides a formal basis of automatic deductive systems for abstract data types, such as AFFIRM [60]. It suggests an approach for studying and extending the method of reasoning about data types developed in the thesis. Other methods of reasoning can also be developed using it. Furthermore, this research clarifies our intuitions about data type behavior and provides a formal basis for them; as examples, the notions of consistency and sufficient completeness advocated by Guttag and Horning [28], and the correctness criterion for an implementation [29, 40] can be stated formally and analyzed.

Our research has been highly influenced by Peano's method of defining natural numbers and McCarthy's method of defining S-expressions [57]. We are intellectually indebted to Zilles [77] and the ADJ group [23], for their work on the algebraic approach for abstract data types, and to Guttag et al. [25, 28, 29] for their work on specification technique for abstract data types which emphasizes programmers' intuitions about data types. We cite other related works in Section 1.2, and state how we plan to compare these works with that discussed in the thesis.

1.1 Scope and Approach of the Thesis

We first state the scope of the thesis and the assumptions made about the data type behavior. The scope and assumptions are further discussed in Appendix I. Later, we give an overview of the approach taken in studying four issues, namely, definition, specification, deductive system, and implementation correctness.

1.1.1 Scope and Assumptions

In our research, we have considered immutable data types having nondeterministic operations and operations exhibiting exceptional behavior. Every operation is assumed to be total and computable; see [42] for a precise characterization of computability on the values of a data type. It terminates on every input in its domain either normally by returning a value of its range type or by signalling an exception. A nondeterministic operation has only finitely many choices on an input. If a nondeterministic operation signals on an input, it is assumed to behave deterministically on that input. So, it does not have a choice between signalling and terminating normally on a particular input. Henceforth, by a data type, we mean an immutable data type with the above behavior, and by an object, we mean an immutable object or a value.

1.1.2 Definition of a Data Type

Our formalism for defining a data type is algebraic in the style of Zilles [77] and the ADJ group [23]. Algebras are a natural and elegant way to define an immutable data type, because an immutable data type is informally a set of values and a set of operations. In a programming language supporting data types, the most important aspect of a data type to its designer as well as its user is the input output behavior of its operations [37, 47, 25]. The values of a data type are manipulated only by its operations. Outside its implementation module(s), the values are viewed abstractly as sequences of operations. The details about the representations of values and the operations of a data type are of no

relevance.[3] To a user, two distinct representations are behaviorally identical if they cannot be distinguished by the operations of the data type. We call this view the *behavioral* view of a data type. The behavioral view abstracts from the representational structure of the values as well as from the multiple representations of a value for any representational structure. It is a further abstraction on the view of a data type adopted by ADJ [23] and Zilles [77] which abstracts only from the representational structure of the values.

In a programming language supporting modularity and hierarchical structure such as CLU, EUCLID, etc., data types are implemented hierarchically one at a time except that mutually recursive data types are implemented together as a group; data types other than those being implemented are assumed to be implemented elsewhere.[4] We take the same approach in defining a data type. Our definitional method is hierarchical. We distinguish between the data type(s) being defined and other data types used in the definition. We call the data type(s) being defined the *defined* type(s) and other data types in the definition the *defining* types. The distinction between the defined type and defining types is significant because the behavior of the values of the defined type is observed by the operations which return the values of the defining types. This was first pointed out by Guttag [25], and is the basis of his definition of the sufficient completeness property. We use the data type *boolean*, which is self-contained and does not have any defining types, as the basis of our definitional method. We assume its definition and that all boolean values are distinguishable. In fact, any data type whose values can be distinguished a priori (outside the formalism) can be used as the basis. For example, any data type directly supported in a programming language whose values are distinguishable using the literal (constant naming) mechanism in the programming language is a suitable candidate.

We classify the operations of a data type into two categories - the *constructors*, which construct the values of the data type, and the *observers*, which return the values of

---

3. We will not be concerned about other issues, such as efficiency of the operations, etc., relevant to a user of a data type. Our formalism is limited in this sense.

4. Mutually recursive data types are different from mutually recursive implementations; see Chapter 5 for a detailed discussion.

the defining types. A value of a data type manifests its behavior through the observers with the help of constructors.

Our approach for modeling the exceptional behavior embodies a practical view of exceptions. Each exception is named, and can have arguments that carry information to its handler from the place where it is signalled. The exceptional behavior of the operations can also be used to distinguish among different values. An operation can distinguish between two values by signalling on one value and terminating normally on the other value, or by signalling different exceptions on different values.

The model used for nondeterminism is simple. If a nondeterministic operation behaves nondeterministically on an input (i.e., it has a choice to return one of the many possible results), we expect it to return every possible result. We do not consider how these results are scheduled by an implementation of the operation. Two operations having different amounts of nondeterminism are considered to have different observable behavior because for some input, they can always return distinguishable results. Data types with operations having different amounts of nondeterminism are thus considered different. For example, consider a data type *finite set of integers* with a nondeterministic operation **Choose** which nondeterministically picks an arbitrary element from a nonempty finite set of integers given as an argument. This data type is different from another similar data type with the same set of operations which also have the same behavior with the exception of **Choose** which is deterministic and returns the maximum integer of a nonempty set. Furthermore, both data types are different from yet a third data type with the same set of operations as the other two types except that **Choose** has a limited amount of nondeterminism: **Choose** nondeterministically picks between the maximum and minimum integers from a nonempty set.

1.1.3 Specification Method

A specification is mainly used, among other things, for reasoning about a data type. So, our specification method is axiomatic in the style of Standish [69], Hoare [38, 39], Guttag [26, 29], Nakajima et al. [62], etc. A specification embodies information hiding [66], i.e., it only specifies the behavior of a data type. Our specification method is hierarchical.

Data types are specified incrementally, one at a time; a specification uses the specifications of other data types. We believe that specifications should be modular and well structured just like programs; otherwise, specifications of large problems become unmanageable and difficult to understand.[5]

A specification expresses the properties particular to the data type(s) being specified. It specifies (i) the domain, range, and the exceptions with the types of their arguments, if any, signalled by every operation, (ii) the normal behavior as well as the exceptional behavior of the operations. The general properties of data types which hold for every data type, for example, the *minimality property* which requires that every value of a data type is constructed by finitely many applications of its constructors, are not included in a specification.

The normal behavior of the operations is specified as a restricted set of formulas of first order multi-sorted predicate calculus with identity. A typical formula is a conditional equation relating different sequences of operations under a condition. A specification can use a finite set of auxiliary functions so that any data type with a finite set of total deterministic computable operations can be specified in this way [43]. A nondeterministic operation is specified like a deterministic operation by expressing the properties of its possible results on an input, rather than by explicitly specifying its relation which holds for all possible results of the operation and the input and does not hold for any other value and the input. For example, in case of the data type *finite set of integers*, the nondeterministic operation **Choose** is specified by relating its possible results to its set argument, instead of explicitly specifying its relation **Choose_p : Set-Int x Int --> Bool** which holds for a set and an integer if and only if **Choose** can return the integer when applied on the set.

The exceptional behavior of the operations is specified as a separate layer on top of the normal behavior. Following Guttag [31], if an operation signals an exception, we

---

5. Burstall and Goguen [7] and Nakajima et al. [62] also emphasize the need for structured specifications.

specify the condition on its input under which the exception is signalled.[6] The specification language provides mechanisms to specify the exceptions which must be signalled by the operations as well as the exceptions which the operations can optionally signal. The specification also allows a precondition on an operation to be specified, stating that the behavior of the operation on inputs not satisfying the precondition is not of any interest. A formula expressing the normal behavior of the operations holds only if the input to the operations in the formula satisfy the specified preconditions and if the operations do not signal; it thus has a restricted interpretation. A formula specifying the normal behavior is called an *axiom*. The preconditions and the exceptional behavior of the operations is specified using *restrictions*.

Our approach of specifying data types is thus different from those of Zilles [77] and the ADJ group [23]. In their approaches, a specification of a data type is a finite set of identities (or conditional identities) presenting the set of algebras serving as the definition of a data type. These identities are interpreted exactly the same way as in Universal Algebra [4, 10]. We are also not constrained to employ only "equational" reasoning; instead, our reasoning method embodies the general properties of data types as is discussed later.

The semantics of a properly designed specification is a set of related data types which differ in the behavior intentionally not captured by the specification. If an operation is specified to be nondeterministic, the semantics of a specification includes data types in which that operation can have as much nondeterminism as desired insofar as the operation behavior satisfies the axioms and restrictions expressed in the specification. We define equivalence among specifications. We also state when a data type can be (precisely) specified in the proposed specification language. We define two important properties of a specification: The *consistency* property, which states whether a specification specifies any data type; the *behavioral completeness* property, which guarantees that the observable behavior of the operations is not left unintentionally unspecified. These properties ensure

---

6. However, this way of specifying the exceptional behavior of the operations may be overly restrictive, as for an operation, the subset of inputs on which it signals a particular exception may be very complex to specify.

that various components of a specification have the desired structure. Checking for these properties is a step towards ensuring that the specification captures the intuition of a designer.

In our research, a clear distinction is made between a data type and its specification. In most of the literature on specification techniques for data types [47, 25, 28, 29, 61, 77, 48, 37], this distinction is either not made or blurred if it is implied. Most of the literature does not explicitly define what a data type is. The ADJ group [23] was the first to our knowledge to explicitly state in their formalism a definition of a data type and make this distinction. We believe the distinction between a data type and its specification is useful and necessary in a formal treatment of data types. Given a definition of a data type, different specification techniques can be developed to serve different purposes, if needed, and their semantics can be given in terms of data types. Different methods of reasoning about a data type can be developed incorporating the general properties of data types with the definition of a data type serving as their basis. The question of whether a given data type can be specified using a particular specification technique can arise only when this distinction is made; only then can different specification techniques be compared in their expressive power. Only then it is meaningful to discuss the properties of a specification technique such as the ease of expression, comprehensibility, minimality, etc., [47]. (See [34] for a similar discussion for programs.)

A specification plays an important role in our research. It is used as a standard for checking the correctness of an implementation as well as for deriving properties of the data types specified as is discussed in the next two subsections. It is an interface between the programs using the data type and the program(s) implementing the data type. The specifications of abstract data types are a major component of a program verification system. Our specification method can be used to specify the behavior of the data component of software designs; questions and inquiries about the data in a design can be expressed and analyzed using the deductive system discussed in the next subsection. (See the two survey papers on specification methods [47, 48], where the need for writing formal specifications is discussed. Guttag and Horning [32] discuss the importance of formal specifications as a design tool.)

1.1.4 Deductive System

As was stated earlier, one of the main reasons for designing a specification is to have an implementation independent description of the data type that can be used to reason about the data type as well as to reason about the designs and programs using the data type. We propose a deductive system based on first order multisorted predicate calculus with identity for deriving properties of a data type from its specification. The deductive system embodies the general properties of data types which are not explicitly stated in a specification but assumed in its semantics. These properties are derived from the syntactic structure of the operations.

The deductive system has an infinite rule which captures the minimality property of data types. The deductive system is powerful enough to prove inequalities. We axiomatize the general properties of the exceptional behavior of the operations. Properties expressed using nondeterministic operations can be proved. We construct a theory of a data type, which is a large subset of its first order properties, from its specification. If a specification specifies a set of related data types, every theorem in the theory constructed from the specification holds for each data type in the set.

We define three other structural properties of a specification, namely, *sufficient completeness*, *well definedness*, and *completeness*, based on what properties of a data type can be deduced from its specification using different fragments of the deductive system. We precisely state the sufficient completeness property defined by Guttag and Horning [28] for a restricted set of specifications and extend it to specifications in our specification language. This property requires that the behavior of the observers on their intended inputs can be completely determined from the specification by purely equational reasoning. We relate this property to the behavioral completeness property stated in the previous subsection, which is model theoretic and which requires that the specification completely specify the behavior of the observers on intended inputs. Recall that the behavioral completeness property does not say anything about what can be deduced from the specification. In this sense, the relation between behavioral completeness and sufficient completeness reflects the power of the equational fragment of the deductive system.

The well definedness property is stronger than the consistency property, because

the well definedness property not only requires that a specification specifies at least one data type, but also that it (specification) is modular in the sense that it preserves the specifications of other data types used in it.

The completeness property is stronger than the sufficient completeness property, since in addition to the requirement that the behavior of the observers can be deduced on any intended input by equational reasoning, it also requires that the equivalence of the observable effect of the constructors on intended inputs can be deduced from the specification by equational reasoning.

1.1.5 Correctness of Implementation

We state the correctness criterion for an implementation coded in a programming language with respect to a specification as a relation between the semantics of the implementation and the semantics of the specification. Roughly speaking, a correct implementation implements one of the data types in the semantics of a specification. Our correctness criterion is weak as it does not require a correct implementation to have the maximum amount of nondeterminism specified by a specification.

We develop a method for proving correctness of an implementation with respect to a specification which embodies the correctness criterion. The method requires, among other things, that the procedures implementing the operations satisfy the axioms and restrictions in the specification when appropriately interpreted. We thus provide the formal basis of the correctness method proposed by Guttag et al. [29] and extend it to specifications specifying nondeterministic operations and operations exhibiting exceptional behavior.

We distinguish among different procedures implementing an operation specified to be nondeterministic, since the nondeterministic behavior of an operation on abstract values can be implemented by a deterministic procedure on the representation of these abstract values that returns different results on different but equivalent representations. We call a procedure *nondeterministic* (respectively, *deterministic*) if it is nondeterministic (respectively, deterministic) and it returns equivalent results on equivalent representations. Otherwise, if a procedure returns different results on equivalent representations, then it is

called *pseudo-nondeterministic* irrespective of whether it is deterministic or nondeterministic on the representations. We discuss the correctness method for these three kinds of procedures implementing an operation specified to be nondeterministic.

1.2 Related Work

In this section, we discuss different definitional and specification methods for data types, briefly stating the major differences as well as the main thrust of these works. The detailed comparison of these works with ours is contained in the rest of the thesis where we discuss various topics.

The definitional methods for data types can be broadly classified as (i) the *algebraic* or *model* approach, and (ii) the *axiomatic* approach. In the model approach, a data type is defined as an algebra satisfying certain properties, or as a set of such algebras. ADJ [23] defines a data type in this way. Though Hoare [37], Zilles [77], Guttag [28], and Berzins [3] do not explicitly define what a data type is, their approaches suggest that a data type is defined using the model approach. Our approach is also the model approach.

Nakajima et al. [62] take the axiomatic approach; they define a data type as a first order multi-sorted theory. Recently Nourani [63] has also discussed the use of a first order theory for defining a data type. Though this view of a data type is useful in program verification, there is no explicit model of a data type to match with the intuition of a designer of the data type. If a first order theory is interpreted as in Logic [16] and its models are taken as the models of a data type being defined, then there are nonstandard models for a data type, which are of no relevance to its designer. A nonstandard model does not satisfy the minimality property of data types discussed in the next chapter. Hoare [38, 39] has also used the axiomatic approach for defining a class of data types.

A survey of specification techniques for data types can be found in [47] and [48]. The specification techniques can be broadly classified into three categories based on their approach: (i) the *model* approach, (ii) the *algebraic* approach, and (ii) the *axiomatic* approach. The model approach is used only in case a data type is defined using the model approach. A data type is specified by presenting one of its models. Berzins [3] has formalized and extended the model approach originally proposed by Hoare [37]. He has

also related his research to other works following the model approach. We discuss here the algebraic and axiomatic approaches.

The algebraic approach has been proposed by Zilles [77] and the ADJ group [23]; in this approach, a set of algebras defining a data type is presented as a finite set of identities or conditional identities. Burstall and Goguen [7] and Goguen [20] specify a data type as an algebraic theory.

The axiomatic approach for specifying a data type can be used for either of the two definitional approaches discussed above. If a data type is defined using the model approach, a specification using the axiomatic approach consists of the properties of the models of a data type. Otherwise, a specification consists of a subset of the theory serving as the definition of the data type. The axiomatic approach followed by Nakajima et al., Hoare [38, 39], and Standish [69] uses the full first order predicate calculus to specify data types. The approach advocated by Guttag et al. uses a restricted set of formulas, namely equations and conditional equations.

Our approach is also axiomatic. A specification expresses the normal behavior of a data type(s) (which is a set of algebras) as equations and conditional equations, and its exceptional behavior as restrictions. As is stated in the previous section, these formulas are interpreted using the restrictions in a different way than in the algebraic approach. In contrast to the specification methods proposed by Nakajima et al., Hoare, and Standish, the general properties of data types are not explicitly stated in our method. A specification provides an incomplete (in the sense of Logic) first order axiomatization of the data types being specified. From a properly designed specification, it is possible to derive most of the interesting properties of a data type needed in program verification.

The major focus of Zilles' work and the ADJ group's work has been to extend the theory of heterogeneous algebras to capture the meaning of data types. They have not investigated how to use the definition of a data type for proving properties of programs using data types. Zilles [76] has suggested an ad hoc method for establishing correctness of an implementation of a data type; however, the method as well as its foundation have not been fully developed. The ADJ group and Ehrig et al. [15] have proposed an algebraic approach for establishing the correctness of an implementation of a data type in which they

have attempted to incorporate the algebraic semantics of the control structures of the programming language used for the implementation. Although the ADJ group's work is rigorous, there are two main problems with it:

(i) it has not embodied the view of data types taken in programming languages, and is thus useful only for a small set of data types, and

(ii) it is complex.

The approach taken by Burstall and Goguen [7] seems more promising than the ADJ group's approach from the viewpoint of program verification, but, we have been told, its category theoretic semantics again seems to introduce unnecessary complexity [30].

Guttag et al. have focused on using specifications for proving properties of data types and programs using data types. The nice aspect of their approach is that it captures the view of a data type taken in programming languages. Our research formalizes, provides a mathematical basis for, and extends their approach.

The ADJ group [23] has been the first to investigate rigorously the exceptional behavior of a data type. In their method, the set of values of every data type is extended to include a distinguished value, called *error*. Using special auxiliary functions which test whether an arbitrary value is an error, they specify the exceptional and normal behavior of a data type. Goguen [20] has enriched and structured their approach. Our approach is based on Guttag's recent suggestions for separating the exceptional behavior of a data type from its normal behavior [31].

1.3 Outline of the Thesis

The second chapter introduces a formalism for defining a data type. We first discuss the formalism for data types assuming that the operations do not signal exceptions. Later, we extend the formalism to incorporate the exceptional behavior of the operations.

The third chapter describes the specification language, gives its semantics, and defines the consistency and behavioral completeness properties of a specification.

The fourth chapter discusses the deductive system. We discuss how a theory of a set of data types serving as the semantics of a specification can be constructed from the specification. We first describe the deductive system for specifications specifying neither nondeterministic operations nor the exceptional behavior of the operations; later, we discuss specifications specifying the exceptional behavior of the operations, and finally, we incorporate nondeterminism. We discuss the deductive system incrementally introducing its various components; we first discuss the equational theory, then the distinguishability theory, later the inductive theory, and finally, the full theory.

The fifth chapter discusses a correctness criterion for an implementation with respect to a specification and a methodology embodying the criterion. The correctness of recursive and mutually recursive implementations is also briefly discussed.

The sixth chapter presents conclusions and directions for future research.

2. Definition of an Abstract Data Type

In this chapter, we develop a formalism to define an abstract data type. We take a *behavioral* view for defining a data type in which every value of the data type is constructed by finitely many applications of its constructors and these values are distinguishable only by means of its operations. We adopt the model approach: A data type is defined to be a set of behaviorally equivalent type algebras, where a type algebra is an extended heterogeneous algebra with additional properties needed to model data types. The syntactic structure of a data type determines the structure of type algebras in the set. Every type algebra in the set is called a model of the data type. A model provides an explicit meaning (interpretation) for the values and the operations of a data type; in this way, it captures concretely the informal description of a data type in our mind. The model approach for defining a data type is closer to the intuition of a programmer than the axiomatic approach as in [62, 63], where a data type is defined as a first order theory.

The crucial concept in the definition of a data type is that of behavioral equivalence of type algebras. The definition of behavioral equivalence captures the informal notion that two behaviorally equivalent type algebras have the same behavior as observed through their operations. We are interested in how the interpretations of the values and the operations of a data type in a model behave, and not in how they are represented. We have decided not to pick a particular model to be the definition of a data type because we do not want the irrelevant details of the model to be associated with the data type. We have only considered the input-output behavior of the operations of a data type.

Behavioral equivalence abstracts from (i) multiple representations of a value for a representational structure as well as from (ii) the representational structure of the values in an algebra. Thus type algebras differing only in the representational structure of their values are behaviorally equivalent; furthermore, type algebras using the same representational structure but differing in the number of representations a value has are also behaviorally equivalent. The property (i) above is achieved by defining a congruence, called the observable equivalence relation, on a type algebra, and the property (ii) is

achieved by the standard algebraic concept of isomorphism. The distinguishability relation, which is the complement of the observable equivalence relation, on the representations of the values of the data type is defined inductively in terms of the distinguishability of the representations of the values of the defining types of the data type. (The basis of this induction is any data type with no defining types, and in particular, the data type *boolean* whose two values *true* and *false* are assumed to be distinguishable.)