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

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

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 Göguen [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.) Two representations are distinguishable if and only if there is a sequence of operations having an observer as the outermost operation, that produces distinguishable results when applied separately on the representations.

If the operations of a data type signal exceptions, then two representations can also be distinguished due to the exceptional behavior of the operations. If a sequence of operations signals on a representation and does not signal on the other, or if it signals different exceptions on the two representations, then they are distinguishable.

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. The definition of distinguishability relation on representations of the values of a data type incorporates this view of nondeterminism.

In the first section, we introduce terminology, define hierarchically structured data types, and informally discuss the minimality property of a data type. We assume data types to be hierarchically structured and defined one at a time. There are however no technical problems in our formalism in handling mutually recursive data types which are not defined separately. We outline the simple extensions of the formalism to such data types in the last section of the chapter. Until the point where we define a data type, we have used the notion of a data type in an informal way to motivate the formalism developed.

In the second section, we first introduce the formalism for defining a data type

assuming that its operations do not signal exceptions. Our definitional method is hierarchical; we assume that the definitions of the defining types are given. We motivate and discuss in detail the distinguishability relation on the representations of the values. We then precisely define the behavioral equivalence relation on type algebras.

In the third section, we incorporate the exceptional behavior of a data type and discuss extensions to the formalism introduced in the second section. We extend a type algebra and the behavioral equivalence relation on type algebras to capture the normal as well as the exceptional behavior of the operations. We compare our approach with Goguen's approach of modeling the exceptional behavior [20, 21]. We also formalize a simpler approach for modeling the exceptional behavior which has been generally assumed in the literature on algebraic specification of data types [25, 27, 77, 23]. We compare our definition of a data type with the definition used by the ADJ group [23] which abstracts only from the representation structure of the values in a type algebra.

2.1 Informal Description of a Data Type

We use the data type *finite set of integers* for illustration; let **Set-Int** be its name. **Set-Int** has been widely discussed in the literature [37, 76, 74, 31]. It has the following operations:

Null	a constant (or 0-ary operation) returning the empty set of integers;
Insert	constructs a finite set of integers by adding a given integer to a given finite set of integers;
Remove	constructs a finite set of integers by deleting a given integer from a given finite set of integers;
Has	checks whether a given integer is an element of a given finite set of integers;
Size	results in an integer giving the size of a given finite set of integers

In addition, we assume that **Set-Int** has an additional operation **Choose**, which has non-deterministic behavior. **Choose** returns an arbitrary element of a given non-empty set of integers; for the time being, we arbitrarily assume that **Choose** returns the integer '0' for the empty set. This behavior of **Choose** for the empty set may not be adequate for some applications. In Section 2.3, we modify **Choose** so that it signals an exception for the empty set.

2.1.1 Terminology

To simplify the mathematics, we assume that an operation has a cartesian product (possibly empty) of data types as its domain and a single data type as its range. An operation having a cartesian product of n data types ($n > 1$) as its range can be viewed in one of the following two ways depending on whichever is more convenient: (i) The operation is modeled as a family of n operations, each having the same domain as the original operation and a different type in the cartesian product as the range, or (ii) the cartesian product is viewed as a single type. We use the first method in the thesis.

Let D be the name of a *new* data type being defined, and Ω be the finite set of symbols naming its operations. Let Δ' stand for the set of names of data types appearing either as a component of the domain or as the range of an operation in Ω. Let Δ be

$\Delta' - \{ D \}$.[1] D is the defined type and every data type in Δ is a defining type of D.

In order to include the syntactic specification (i.e., the domain and range specifications) of the operations, we index every operation σ in Ω by a pair (d, r), where d is a string made from the alphabet Δ' and r is an element of Δ'. d specifies the domain of σ and r specifies its range.

Let **Int** stand for the data type *integers* and **Bool** stand for the data type *boolean*. For **Set-Int**, $\Delta = \{$ **Int, Bool** $\}$, $\Delta' = \{$ **Int, Bool, Set-Int** $\}$ and $\Omega = \{$ **Null, Insert, Remove, Has, Size, Choose** $\}$. The index of **Insert** for example is **(Set-Int · Int, Int)**.

As is discussed in the first chapter, the operations of D can be classified as constructors and observers. Let Ω_c be the subset of Ω consisting of all constructors of D (recall that a constructor is an operation having D as its range). For example, **Null**, **Insert**, and **Remove** are the constructors of **Set-Int**. The constructors construct all the values of D. Some constructors construct a value of D using only the values of the defining types of D. We call such a constructor a *basic constructor*. For example, **Null** is a basic constructor of **Set-Int**. Every data type is required to have at least one basic constructor; otherwise, D will not have any values.

Let Ω_o be the subset of Ω consisting of all observers of D. An observer examines the values of D; it takes at least one argument of type D, and returns a value of a defining type of D. For example, **Has**, **Size**, and **Choose** are the observers of **Set-Int**. Every interesting data type must have at least one observer, otherwise there is no way to distinguish among different values of D [25] other than by the operations signalling on the values. An observer is also called an *inquiry operation* [77].

We thus assume that every operation of D either results in a value of D, or takes an argument of type D, or both. We consider a data type having an operation not satisfying this requirement to be not properly designed, because the behavior of such an operation does not depend on the data type.

---

1. Henceforth we will not distinguish between a data type and its name, and between an operation and its name, unless needed.

Let Ω_{nd} stand for the set of nondeterministic operations of D. We allow any kind of operation, an observer or a constructor, to be nondeterministic. In our experience, however, we have found that a nondeterministic operation is often an observer. [2]

2.1.2 Hierarchical Structure

We define the following two relations on a set of data types for capturing the dependency structure among the data types:

Def. 2.1 D *directly depends on* every $D' \in \Delta$, and does not directly depend on any other data type. ∎

Def. 2.2 D *depends on* D′ if (i) D directly depends on D′, or (ii) there is a D″ such that D directly depends on D″ and D″ depends on D′. ∎

The *direct dependency relation* captures one level of hierarchical dependency. The *dependency relation* is the *transitive closure* of the direct dependency relation. We define

$(D)^+ = \{ D' \mid D \text{ depends on } D' \}$, and

$(D)^* = (D)^+ \cup \{ D \}$.

If data types are designed so that every data type on which D depends is assumed to be designed independently of D, then the dependency relation on $(D)^+$ will not have any cycles and is a strict partial order on data types. In such a case, data types are said to be *hierarchically structured*, and they can be defined incrementally one at a time. Data types on which D depends do not have to be designed in any particular order relative to D; any approach, for example top-down, bottom-up, etc., is compatible. Unless stated otherwise, we assume in the thesis that data types are hierarchically structured.

We assume that the partial order induced by the dependency relation on the set of hierarchically structured data types has finite descending chains. The bottom of every

---

2. In case a constructor σ is nondeterministic, σ is usually *derived* with respect to a subset Ω_g of deterministic constructors ($\Omega_g \subseteq \Omega_c$) in the sense that σ does not return any value that cannot be constructed using the constructors in Ω_g.

chain is a data type having no defining type. Throughout this thesis, we assume that the data type *boolean* does not have any defining type; **Bool** serves as the bottom element of the chains in the partial ordering for all interesting data types as will be clear from the discussion in Section 2.2. (The definition of **Bool** is given in Section 2.2.) We will often use the structure induced by the dependency relation on the set of data types for inductively defining properties of data types, as well as for proving properties about data types. **Bool** will often serve as the basis step of such definitions and proofs (in general, data types having no defining type serve as the basis).

2.1.3 Minimality Property

The requirement on a data type behavior imposed because of the modularity and good program design considerations that its values be manipulated only by its operations translates to requiring that its values be constructed only by its constructors, possibly using abstractly the values of its defining types. Furthermore in a computer the values can be constructed only by a finite sequence of operations, so the values of a data type constitute the smallest set closed under finitely many applications of its constructors. We call this property of a data type the *minimality property.*

We require that every data type under consideration satisfy the minimality property. This requirement constrains the implementations of a data type to be *protected* in Morris's sense [59]. An implementation of a data type defined in a strongly typed language that hides the representation of its (data type) values from its users by providing an encapsulation mechanism, as in CLU, ALPHARD, etc., is protected. The minimality requirement does not rule out data types defining 'infinite' values, insofar as these values can be finitely described. [3]

---

3. For example, we can define a data type *infinite sequence of squares*, whose values are infinite sequences of consecutive squares starting from n^2, for every $n \geq 0$. It has a constructor, **Cons**, which takes a natural number as an argument and returns an infinite sequence. In addition, it has three observers - **First**, which gives the first element in the sequence; **Rest**, which gives the remaining sequence after stripping the first sequence; and, **Equal**, which checks whether two infinite sequences are equal or not.

The minimality property serves as the basis of a powerful *induction rule* for a data type D: To prove that a property P holds for D, i.e., for all values of D, we need to show that P is preserved by every constructor of D. Wegbreit and Spitzen [72] called this *generator induction*; Guttag et al. [27] called it *data type induction*. We discuss this induction rule in detail in Chapter 4 on the deductive system for data types.

Since every operation of D is assumed to be computable, it can be easily shown by induction on data types, that the set of values of D is recursively enumerable.[4] This is based on the fact that the set of sequences of constructions is recursive. This thesis considers data types with a recursively enumerable set of values and a finite set of total computable operations.

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

2.2 Formalism

In this section, we describe the formalism to state precisely what a data type is. To simplify the presentation, we assume that data types do not have any exceptional behavior, i.e., their operations do not signal any exceptions. Every operation terminates normally on every input in its domain.

This section is organized as follows. We first extend the notion of a heterogeneous algebra as defined in [4] to model nondeterminism; then we define a *type algebra* to be an extended heterogeneous algebra with additional properties. The domain corresponding to the defined type **D** consists of the representations of the values of **D** and is called the *principal* domain of the type algebra. To extract the behavior of a type algebra as observed through its operations, we must

(i) abstract from the multiple representations of a value, assuming a particular representational structure, and

(ii) abstract from the representation structure of the values and operations in a type algebra.

To do the first, we define an interpretation of a term in a type algebra, where a term expresses a sequence of operations. Terms are used to observe the behavior of the representations of the values of the defined type in a type algebra in terms of the representations of the values of the defining types. We define the *observable equivalence* and *distinguishability* relations on the principal domain of a type algebra. These relations are defined inductively using the corresponding relations on the domains corresponding to the defining types in the type algebra. Observable equivalence is an equivalence relation and is preserved by the functions in a type algebra; it relates two values having the same behavior. We then define the *behavioral equivalence* relation on type algebras which relates two type algebras having the same observable behavior. A data type is an equivalence class defined by the behavioral equivalence relation, and every type algebra in the equivalence class is a *model* of the data type. A model of a data type concretely defines the *value set*, which is the principal domain of the model, and the operations of the data type.

Most of the definitions throughout this section are inductive; they make use of

the dependency relation, which is a strict partial order with finite descending chains, on hierarchically structured data types. An inductive definition of a concept has three parts:

(i) *Basis* part, which deals with the case of a data type D having no defining type, i.e., its Δ is the null set,

(ii) *inductive* part, which deals with the case of a data type having defining types, and

(iii) *closure* part, which states that the above two ways are the only ways of defining a concept.

To avoid repetition, we omit the closure part, and if the basis part can be derived from the inductive part by assuming Δ to be the null set, we give only the inductive part of the definition. Some of the definitions - the definitions of type algebra (Def. 2.3), distinguishability and observable equivalence relations (Defs. 2.6 and 2.7) and data type (Def. 2.14) are mutually recursive. The definitions 2.3, 2.6, and 2.7 assume the definitions of the defining types in Δ in their inductive part.

We would like to motivate various concepts and definitions introduced on type algebras. So for exposition purposes, we may refer to a type algebra as though it is a model of a data type being discussed.[5]

2.2.1 Type Algebras

A heterogeneous algebra as defined by Birkhoff and Lipson [4] is a finite indexed set of sets (called *domains* in the thesis) and a finite indexed set of total functions. We extend this definition to model the nondeterministic operations of a data type. An *extended* heterogeneous algebra can have either a total (deterministic) function or a total *nondeterministic* function.

A nondeterministic function $f: X \rightarrow Y$ is similar to a function in mathematics with the exception that it has a choice among a subset of possible results when applied on an input $x \in X$. Let $f(x)$ stand for an arbitrary result of applying f on x. f can be characterized

5. We are technically justified to do so as almost every type algebra is a model of some data type.

using a relation $\mathbf{R} \subseteq \mathbf{X} \times \mathbf{Y}$ such that $\mathbf{f}(x) \in \mathbf{R}(x)$.[6] If $\mathbf{R}(x)$ is a singleton set for some input x, then **f** is said to be deterministic on x. By $\{ \mathbf{f}(x) \}$ we will mean the set $\mathbf{R}(x)$; in this way we do not have to refer to **R**. Since we assume every nondeterministic operation to have finitely many choices on a particular input, $\{ \mathbf{f}(x) \}$ is always a finite set. We admit that calling **f** a nondeterministic function is an abuse of the term function; however, we feel this term conveys the behavior of **f** well. Henceforth, by the term function we mean either a mathematical (deterministic) function or a nondeterministic function, unless qualified. We have chosen a nondeterministic function over the corresponding deterministic relation for modeling a nondeterministic operation because (i) in contrast to the nondeterministic function, the relation models the nondeterministic operation indirectly, and (ii) it is inconvenient and unnatural to express the behavior of a computation scheme involving nondeterministic operations by means of the relations corresponding to the nondeterministic operations.

The definitions of concepts such as congruence, homomorphism, isomorphism on heterogeneous algebras [4] are revised for extended heterogeneous algebras in Appendix II. Henceforth, we use the term heterogeneous algebra to mean an extended heterogeneous algebra.

A *type* algebra is a heterogeneous algebra with additional properties. For a data type **D**, we are interested in type algebras having a particular structure, which is determined by Δ' and Ω of **D**. The sets Δ' and Ω serve as the index sets of the type algebras of interest for **D**. We call such an algebra as an *algebra of type* **D** or simply a type algebra when **D** is evident from the context. The triple $(\mathbf{D}, \Delta, \Omega)$ is called the (*similarity*) *type* of such an algebra. An algebra **A** of type **D** consists of a domain corresponding to every type name $\mathbf{D}' \in \Delta'$ and a function of the appropriate arity corresponding to every operation name in Ω. The domain corresponding to **D** is the principal domain of **A**. The function corresponding to σ is called the *interpretation* of the operation symbol σ in **A**. The domain corresponding to a defining type $\mathbf{D}' \in \Delta$ is the *interpretation* of $\mathbf{D}'$.

---

6. For a relation **R**, a subset of $\mathbf{X} \times \mathbf{Y}$, $\mathbf{R}(x)$ stands for the subset $\{ y \mid \langle x, y \rangle \in \mathbf{R} \}$ of **Y** for an $x \in \mathbf{X}$, and $\mathbf{R}(\Lambda)$ stands for $\{ y \mid \langle x, y \rangle \in \mathbf{R}, x \in \Lambda \}$, where $\Lambda \subseteq \mathbf{X}$.

We assume that every defining type D' in Δ of D is defined elsewhere and we are *given* the models of D' (see Subsection 2.2.6 for the definition of a data type and a model of a data type). The interpretation of a data type $D' \in \Delta$ in an algebra of type D is fixed. We use the models of each $D' \in \Delta$ to define type algebras of D. The domain corresponding to $D' \in \Delta$ in a type algebra $\mathbf{A}$ of D is the value set of D' defined by some model $\mathbf{A}'$ of D'. A type algebra $\mathbf{A}$ of D explicitly includes only the interpretations of the operation names of D, and does not include the interpretations of the operation names of any defining type D'. We assume that every operation name of a defining type D' has the same interpretation in $\mathbf{A}$ of D as its interpretation in the model $\mathbf{A}'$ of D'. In this way, we define the interpretation of every operation name of a data type $D'' \in (D)^*$ in a type algebra $\mathbf{A}$ of D.[7] An algebra $\mathbf{A}$ of type D is thus really a huge structure having interpretations for every data type in $(D)^*$.

Def. 2.3 An algebra $\mathbf{A}$ of type D is a heterogeneous algebra

$$[\{ V_{D'} \mid D' \in \Delta' \};\ \{ f_\sigma \mid \sigma \in \Omega \}],$$

such that

(i) for every defining type $D' \in \Delta$, $V_{D'}$ is the value set of D' defined by a model $\mathbf{A}'$ of D',

(ii) for every $\sigma \in \Omega$, f_σ is a total function of the appropriate arity, i.e., if σ has $D_1 \times \ldots \times D_n$ as its domain and D' as its range,[8] then f_σ has $V_{D_1} \times \ldots \times V_{D_n}$ as its domain and $V_{D'}$ as its range, and

(iii) V_D is the smallest set closed under finitely many applications of the functions corresponding to the constructors of D, i.e.,

$$V_D = \bigcup_{j=0}^{\infty} V_D^j, \text{ where } V_D^0 = \emptyset \text{ and}$$

$$V_D^{j+1} = \{ f_\sigma(v_1, \ldots, v_n) \mid \text{for each } \sigma \in \Omega_C \text{ such that}$$

$$\sigma : D_1 \times \ldots \times D_n \to D,\ v_i \in \bigcup_{k=0}^{j} V_D^k \text{ if } D_i = D, \text{ and } v_i \in V_{D_i} \text{ if } D_i \neq D \}.$$

∎

---

7. Recall that $(D)^*$ is the set consisting of D and all data types on which D depends.
8. I.e., $(D_1 \cdot \ldots \cdot D_n, D')$ is the index of σ.

So, V_D is the principal domain of **A**, f_σ is the interpretation in **A** of the operation name $\sigma \in \Omega$. We do not require the interpretation f_σ of σ to be a deterministic function if σ is deterministic and f_σ to be a nondeterministic function when σ is nondeterministic; the reason for this will become clear in Subsections 2.2.5 and 2.2.6 on the behavioral equivalence of type algebras and the definition of a data type respectively.

If any f_σ in **A** is a nondeterministic function, then **A** is called a *nondeterministic* type algebra; otherwise, if every f_σ is deterministic, then **A** is called a *deterministic* type algebra. Henceforth, in the context of an algebra **A** of type D, by an operation σ we mean its interpretation f_σ and by a value of D we mean an element of V_D.

The property (iii) above is due to the requirement that D satisfies the minimality property. For a constructor σ, if f_σ is nondeterministic, then V_D is closed under f_σ assuming f_σ could return any possible result for an input. Once the value set corresponding to each defining type D' is fixed, then obviously V_D is uniquely determined by $\{ f_\sigma \mid \sigma \in \Omega_C \}$, and is nonempty, because Ω_e is nonempty and has at least one basic constructor (see Section 2.1).

2.2.2 Examples of Type Algebras

We discuss below a type algebra A_{si} of **Set-Int**. A_{si} is a natural model of **Set-Int** in the sense that its principal domain is the set of all finite sets of integers, and the interpretations of its operations are defined in terms of the standard set operations [16].

$$A_{si} = [\{ S, Z, B \}; \{ Nu, In, Re, Ha, Si, Ch \}],$$

where $B = \{$ **true**, **false** $\}$, a value set of **Bool**,

$Z = \{ 0, 1, -1, 2, -2, \ldots \}$, a value set of **Int**, and

$S = \{ \varnothing, \{0\}, \{1\}, \{-1\}, \{2\}, \{-2\}, \{0, 1\}, \{0, -1\}, \{0, 2\}, \{0, -2\}, \{1, -1\}, \{1, 2\}, \ldots \}$, the domain corresponding to **Set-Int**.

The domains Z and B are defined elsewhere by the models of **Int** and **Bool**, respectively.

The first two letters of an operation name are used to denote in A_{si} the total function corresponding to the operation. These functions are defined below. We will use any convenient mathematical formalism to give the definitions of the functions. We use

the symbol ' $\triangleq$ ' as the definition symbol; the symbol ';' marks the beginning of a comment in a definition, running until the end of the line.

$\mathbf{Nu} \triangleq \varnothing$

$\mathbf{In}(s, i) \triangleq s \cup \{i\}$

$\mathbf{Re}(s, i) \triangleq s - \{i\}$; - is the difference operator

$\mathbf{Ha}(s, i) \triangleq i \in s$

$\mathbf{Si}(s) \triangleq \#(s)$; the cardinality of the set

$$\mathbf{Ch}(s) \triangleq \begin{cases} 0 & \text{if } s = \varnothing \\ i & \text{such that } i \in s, \text{ otherwise.} \end{cases}$$

Ch is a nondeterministic total function; if s is not ø, then $\{\mathbf{Ch}(s)\} = s$.

We discuss another type algebra $\mathbf{A}^1_{si}$ of **Set-Int** in which the set values are represented as finite sequences of nonrepeating integers.

$$\mathbf{A}^1_{si} = [\{\mathbf{SQ}^1, \mathbf{Z}, \mathbf{B}\}; \{\mathbf{Nu}^1, \mathbf{In}^1, \mathbf{Re}^1, \mathbf{Ha}^1, \mathbf{Si}^1, \mathbf{Ch}^1\}],$$

where $\mathbf{SQ}^1 = \{\langle\rangle, \langle 0\rangle, \langle 1\rangle, \langle -1\rangle, \langle 2\rangle, \langle -2\rangle, \langle 0, 1\rangle, \langle 0, -1\rangle, \langle 0, 2\rangle, \langle 0, -2\rangle,$
$\langle 1, 0\rangle, \langle 1, -1\rangle, \langle 1, 2\rangle, \langle 1, -2\rangle, \langle -1, 0\rangle, \langle -1, 1\rangle, \langle -1, 2\rangle, \langle -1, -2\rangle,$
$\langle 2, 0\rangle, \langle 2, 1\rangle, \ldots, \}$, the domain corresponding to **Set-Int**.

The set $\mathbf{SQ}^1$ contains all finite sequences of integers not having multiple occurrences of the same integer, for example, $\langle 0, 0\rangle$, $\langle 0, 1, -1, 1\rangle$ are not in $\mathbf{SQ}^1$. Let s stand for an element of $\mathbf{SQ}^1$. So, $s = \langle i_1, \ldots, i_m\rangle$, $m \geq 0$; if $m = 0$, then $s = \langle\rangle$.

$\mathbf{Nu}^1 \triangleq \langle\rangle$

$$\mathbf{In}^1(\langle i_1, \ldots, i_m\rangle, i) \triangleq \begin{cases} \langle i_1, \ldots, i_m\rangle & \exists\, 1 \leq j \leq m,\ i_j = i \\ \langle i_1, \ldots, i_m, i\rangle & \text{otherwise} \end{cases}$$

$$\mathbf{Re}^1(\langle i_1, \ldots, i_m\rangle, i) \triangleq \begin{cases} \langle i_1, \ldots, i_{j-1}, i_{j+1}, \ldots, i_m\rangle & \exists\, 1 \leq j \leq m,\ i_j = i \\ \langle i_1, \ldots, i_m\rangle & \text{otherwise} \end{cases}$$

$$\mathbf{Ha}^1(s, i) \triangleq \begin{cases} \text{true} & \exists\, 1 \leq j \leq m,\ i_j = i \\ \text{false} & \text{otherwise} \end{cases}$$

$\mathbf{Si}^1(s) \triangleq m$

$$\mathbf{Ch}^1(\langle i_1, \ldots, i_m\rangle) \triangleq \begin{cases} 0 & m = 0 \\ i_j & 1 \leq j \leq m > 0. \end{cases}$$

Ch^1 is a nondeterministic function; $\{ Ch^1(\langle i_1, \ldots, i_m \rangle) \} = \{ i_1, \ldots, i_m \}$ for $m > 0$.

2.2.3 Interpretation of Terms

A term is constructed using the operation names of types in $(D)^*$ and the typed variables. It expresses a sequence of operations, so it forms a straight line program. The interpretation of a term in a type algebra is like the execution of such a program. The interpretation of all terms characterizes the behavior of the algebra.

We assume that we have as many variables (possibly infinite) of every type $D' \in (D)^*$ as needed.

Def. 2.4 A *term* of type $D' \in (D)^*$ is defined inductively as follows:

(i) A variable x of type D' is a term of type D',

(ii) if σ is an operation of some type $D'' \in (D)^*$ such that its domain is $D_1 \times \ldots \times D_n$ and its range is D', then '$\sigma(e_1, \ldots, e_n)$' is a term of type D' if and only if each e_i is a term of type $D_i \in (D)^*$. ∎

If a term has no variables, it is called a *ground* term. A term of type **Bool** is called a *boolean* term. When we wish to refer to the variables of e, we write e as $e(x_1, \ldots, x_n)$ (or $e(X)$), where the set $\{ x_1, \ldots, x_n \}$ (or X) consists of all variables in e. A *subterm* of a term that is a variable is the term itself. The subterms of a term of the form '$\sigma(e_1, \ldots, e_n)$' are (i) the term '$\sigma(e_1, \ldots, e_n)$' itself, (ii) all subterms of $e_1, \ldots, e_n$, and nothing else.

An *interpretation* of a ground term e in an algebra **A** of type **D** is obtained by performing the sequence of operations expressed by e. A ground term e of type D' is *interpreted* in **A** as follows: If e is a 0-ary operation name σ, an interpretation of e is the result of applying the interpretation of σ in **A**. If e is '$\sigma(e_1, \ldots, e_n)$,' an interpretation of e is the result of applying the interpretation of σ in **A** on the interpretations of $e_1, \ldots, e_n$ in **A**. An interpretation of e is an element of $V_{D'}$. Since e may be constructed using nondeterministic operation names, e can have many interpretations. Let $e|_{\mathbf{A}}$ stand for an arbitrary interpretation of e in **A**.

For example, let us assume that the defining type **Int** of **Set-Int** has the

constructors 0, 1, 2, and 3, and that they have the standard interpretation in a model of **Int**. Then e_1 = **Insert(Insert(Null, 0), 1)** and e_2 = **Choose**(e_1) are ground terms of types **Set-Int** and **Int** respectively. We have,

$e_1|_{\mathbf{A}_{si}}$ = {0, 1}, and

$e_2|_{\mathbf{A}_{si}}$ = 0 or 1.

Since every operation name of a data type $D' \in (D)^*$ has a total function as its interpretation in an algebra **A** of type D, we have

Prop. 2.1 Every ground term of type $D' \in (D)^*$ has an interpretation in **A**. ∎

Furthermore, since every data type under consideration has the minimality property, we have

Prop. 2.2 Every value in V_D is an interpretation of some ground term of type D.

Proof Straightforward, by induction on type algebras using the dependency relation. ∎

For a term e of type D' having variables, its interpretation is a function, which is denoted by f_e. If e has nondeterministic operation names, then f_e is in general a nondeterministic function. Let $\{x_1, \ldots, x_n\}$ be the only variables in e and D_i be the type of x_i. Then f_e has $V_{D_1} \times \ldots \times V_{D_n}$ as its domain and $V_{D'}$ its range. If the variables $x_1, \ldots, x_n$ in e are *instantiated* in **A** to be the values $v_1, \ldots, v_n$ respectively, from the appropriate domains in **A**, then $e(x_1, \ldots, x_n)$ is said to be *instantiated* in **A** as $e[x_1/v_1, \ldots, x_n/v_n]$, and can be interpreted in **A**. The *assignment* $[x_1/v_1, \ldots, x_n/v_n]$ is called an **A**-*instance* of $x_1, \ldots, x_n$, and each v_i is called an *instance* of x_i. (We will abbreviate the assignment as $[X/V]$, where V stands for $(v_1, \ldots, v_n)$.) An interpretation of $e[X/V]$ in **A**, written as $e[X/V]\|_{\mathbf{A}}$, is defined as follows:

(i) If e is a variable x_i, then $e[X/V]\|_{\mathbf{A}} = v_i$, and

(ii) if e is of the form '$\sigma(e_1, \ldots, e_m)$', $m \geq 0$,

then $e[X/V]\|_{\mathbf{A}} = f_\sigma(e_1[X/V]\|_{\mathbf{A}}, \ldots, e_m[X/V]\|_{\mathbf{A}})$.

$f_e(V)$ is $e[X/V]\|_{\mathbf{A}}$.

Interpreting a ground term or an instantiated term in **A** is thus like performing a

computation; an interpretation is the result of the computation.

2.2.4 Observable Behavior

The behavior of a sequence of operations of a data type D, strictly speaking, becomes externally observable if the sequence has an effect on the outside world, for example, the sequence of operations ultimately results in some output on an I/O device, such as a line printer, CRT, etc. In this sense, the distinction between two values of **D** is observable if and only if there exists a sequence of operations such that when applied on the values separately, it returns distinguishable outputs on an I/O device. An output on an I/O device can be considered as a sequence of characters, and we can have a predicate on the outputs, resulting in the boolean constants **T** and **F** depending upon whether the two given outputs are distinguishable or not. In this way, we can define the distinguishability of the values of **D** using the distinguishability of the boolean constants. We stop at **Bool**. As was stated earlier, we use the definition of **Bool** as the basis of our formalism. In fact, any data type (or a collection of data types) whose values can be distinguished a priori (outside the formalism) can be used as the basis. For instance, a data type directly supported in a programming language whose values are distinguishable using the literal mechanism in the programming language can be used.

We structure the above informal definition of distinguishability using the dependency relation on data types. Instead of defining the distinguishability of the values of **D** in terms of the distinguishability of boolean values in a single step, we do it incrementally. We assume that the distinguishability relation is defined on the values of every defining type $D' \in \Delta$, if any; in this way, the behavior of the values of **D** can be incrementally observed through its observers. Except for **Bool**, if **D** does not have any observers, i.e., its Ω_0 is the empty set, then the values of **D** are not distinguishable, as there is no way to tell whether any two values are different. That is why we remarked earlier that every interesting data type must have at least one observer.

For a **D** with a nonempty set of observers, it is generally not sufficient to examine the values of **D** directly by the observers due to the possible delayed effects of the constructors. The distinguishability of the values may not manifest itself until some

constructors are applied on them. For example, two different nonempty stacks of the data type *stack of integers* may have the same integer as their *top* element, so they cannot be distinguished directly by the observer **Top**. But if we apply the **Pop** operation first on the two stacks, then the resulting stacks may be directly distinguishable by the observer **Top** thus exhibiting that the original stacks are also distinguishable. There is generally a need to perform a sequence of operations with an observer of **D** as the last operation in the sequence, to distinguish two values of **D**.

Informally, two values of **D** are *distinguishable* if and only if either

(i) there is a sequence of deterministic operations of **D** such that when it is applied on the two values assuming every other argument of the sequence fixed, it results in distinguishable values of some defining type $D' \in \Delta$, or

(ii) there is a sequence including nondeterministic operations such that the result of applying it on a value for some choice made by the nondeterministic operations is distinguishable from the result of applying it on the other value no matter what choice is made by the nondeterministic operations.

If two values are not distinguishable, they are called *observably equivalent.* For better exposition, we have deliberately structured the definition of distinguishability into two cases, though the second case can be modified to cover the first case. The second case may appear to be a very strong requirement, but a small amount of thinking should convince the reader that such is not the case, as we definitely do not want a value to be distinguishable from itself. Furthermore, observable equivalence should be an equivalence relation and it must be preserved by the operations of the data type. We precisely state below these requirements in the context of a type algebra and illustrate them using examples.

The operations of a data type must also preserve the observable equivalence relation on the values of every defining type in the sense that the operations cannot distinguish among the observably equivalent values of a defining type. This requirement on the operation behavior is necessary because of the modular structure of data types. A new data type should not impose any additional structure on the values of any of its defining data types. This property of a data type is guaranteed in all programming

languages supporting an abstract data type mechanism in which an implementation of a data type is hierarchically structured and the representation is hidden from the users of a data type.

We would like the type algebras to have the above properties. Definition 2.3 of a type algebra does not guarantee them, so we put an additional constraint on a type algebra. We first define the observable equivalence relation E_D on the principal domain V_D of a type algebra **A**; we will assume that the observable equivalence relation $E_{D'}$ on $V_{D'}$ in **A** is defined for each $D' \in \Delta$ by a model **A'** of **D'** having $V_{D'}$ as its principal domain. We show that E_D as defined below is an equivalence relation. Later we define a well formed type algebra whose functions preserve the set $E = \{ E_{D'} \mid D' \in \Delta' \}$ of observable equivalence relations. Only the well formed type algebras are of interest for defining a data type.

In the above discussion, we have only considered the input-output behavior of the operations for distinguishing different values. We have not considered the efficiency of the operations. In case of nondeterministic operations, we have not considered how possible values that a nondeterministic operation can return on a particular input are scheduled. Our formalism is limited in this sense.

2.2.4.1 Definitions of Observable Equivalence and Distinguishability

We give the basis and the inductive parts of the inductive definition of the distinguishability relation. The basis part is the case when **D** does not have any defining type and the inductive part is the case when **D** has defining types. In the basis part, there are two subcases: (i) **D** is **Bool**, and (ii) **D** is different from **Bool**. We first define the data type **Bool** and then define the distinguishability relation on the models of **Bool**.

The data type **Bool** does not have any defining types and is self-contained. We present below a model of **Bool** and call it **B**.

B = ({ { **true**, **false** } }; { T, F, ∨, ~, ∧, ⇒, ⇔ }), where

$T \triangleq$ **true**

$F \triangleq$ **false**

~ **true** $\triangleq$ **false**

$$\sim \textbf{false} \triangleq \textbf{true}$$
$$\textbf{true} \vee \textbf{true} \triangleq \textbf{true}$$
$$\textbf{true} \vee \textbf{false} \triangleq \textbf{true}$$
$$\textbf{false} \vee \textbf{true} \triangleq \textbf{true}$$
$$\textbf{false} \vee \textbf{false} \triangleq \textbf{false}$$
$$x \wedge y \triangleq \sim((\sim x) \vee (\sim y))$$
$$x \Rightarrow y \triangleq (\sim x) \vee y$$
$$x \Leftrightarrow y \triangleq (\sim(x \vee y)) \vee (x \wedge y)$$

The interpretation of T is the logical value **true** and the interpretation of F is the logical value **false.**

Def. 2.5 The data type **Bool** is the set of all type algebras isomorphic to **B.** ∎

We will often use **B** as if it is the only model of **Bool**, and interchange between T and its interpretation **true** in **B** as well as between F and its interpretation **false**. We assume that the boolean constants T and F are distinguishable from each other a priori, meaning that their interpretation in every model of **Bool** is distinguishable. Each boolean constant is observably equivalent to itself.

Def. 2.6.1 Let **A** be a model of **Bool** and V_{Bool} be the value set of **Bool** defined by **A**. The *observable equivalence* relation on V_{Bool} is defined to be the identity relation on V_{Bool}. The *distinguishability* relation on V_{Bool} is defined to be the complement of the observable equivalence relation with respect to the universal relation on V_{Bool} (i.e., $V_{Bool} \times V_{Bool}$). ∎

The other component of the basis part of the definition of distinguishability is now given.

Def. 2.6.2 For any data type D other than **Bool** not having any defining type, no value in V_D of an algebra **A** of type D is distinguishable from any other value in V_D. ∎

The inductive part is as follows:

Def. 2.6.3 Two values v_1 and v_2 in V_D of a type algebra **A** are *distinguishable* iff there is a term of type D' with exactly one variable of type **D**, expressed as $c(x)$, such that the instantiation $c[x/v_1]$ interprets in **A** to a value of a type $D' \in \Delta$ (an element of $V_{D'}$) that is distinguishable from every possible value to which the instantiation $c[x/v_2]$ interprets, or vice versa. ∎

The case 2.6.2 above can be derived from the case 2.6.3.

Def. 2.7 v_1 and v_2 are *observably equivalent*, i.e., $(v_1, v_2) \in E_D$ iff v_1 and v_2 are not distinguishable. ∎

It should also be obvious from the above definitions that if **D** does not have any observers and **D** is different from **Bool**, then all members of V_D are observably equivalent. The following definitions are useful in dealing with data types having nondeterministic operations.

Def. 2.8 Given two subsets A_1 and A_2 of V_D, A_1 is *observably equivalent* to A_2 and vice versa, iff $(\forall\, v_1 \in A_1)(\exists\, v_2 \in A_2)[\langle v_1, v_2\rangle \in E_D]$, and vice versa ∎

Def. 2.9 A_1 and A_2 are *distinguishable* iff A_1 and A_2 are not observably equivalent. ∎

Then the case 2.6.3 can be rephrased as:

v_1 and v_2 are distinguishable iff there is a term $c(x)$ such that $\{\ c[x/v_1]\|_{\mathbf{A}}\ \}$ is distinguishable from $\{\ c[x/v_2]\|_{\mathbf{A}}\ \}$.

Consider the type algebra $\mathbf{A}_{si}$ of **Set-Int** (see Subsection 2.2.2). It can be proved using the definition of **Int** that the observable equivalence relation on **Z**, the value set of **Int** used in $\mathbf{A}_{si}$, is the identity relation. Then the sets {} and {0} are distinguishable since the term Size(x) distinguishes them. The sets {0, 1} and {1, 2} are also distinguishable since the term **Choose**(x) distinguishes them: An interpretation of **Choose**({0, 1}) is either 0 or 1, and if 0 is chosen as an interpretation, there is no interpretation of **Choose**({1, 2}) returning 0. By similar reasoning, {0, 1} is also distinguishable from {0}. {0, 1} is observably equivalent to itself. The observable equivalence relation on the principal domain of $\mathbf{A}_{si}$ is the identity relation. However, it can be shown that the observable equivalence relation on

the principal domain of $\mathbf{A}^1_{si}$ is not the identity relation, because for example, $\langle 1, 2\rangle$ is observably equivalent to $\langle 2, 1\rangle$. In fact, any two sequences having the same set of integers are observably equivalent. In $\mathbf{A}^1_{si}$,

$\mathbf{E}^1_{Set\text{-}Int} = \{ \langle s1, s2\rangle \mid s1 \text{ is a permutation of } s2 \}$.

Thm. 2.1 The observable equivalence relation $\mathbf{E}_D$ is an equivalence relation.

Proof That $\mathbf{E}_D$ is reflexive and symmetric is obvious from the definition. The transitivity of $\mathbf{E}_D$ can be shown by induction on type algebras using the dependency relation. ∎

The requirement that the functions in a well formed type algebra **A** preserve the observable equivalence relation $\mathbf{E}_{D'}$ for each $D' \in \Delta'$ is equivalent to requiring that $\mathbf{E} = \{ \mathbf{E}_{D'} \mid D' \in \Delta' \}$ be a congruence on **A**., where a congruence on a heterogeneous algebra is defined in Appendix II.

Def. 2.10 A type algebra **A** is *well formed* if and only if **E** is a congruence on **A**. ∎

Since we are interested only in well formed type algebras, by a type algebra we henceforth mean a well formed type algebra unless stated otherwise.

For example, both $\mathbf{A}_{si}$ and $\mathbf{A}^1_{si}$ are well formed. $\mathbf{E}^1 = \{ \mathbf{E}^1_{Set\text{-}Int}, \mathbf{E}_{Int}, \mathbf{E}_{Bool} \}$ in case of $\mathbf{A}^1_{si}$, where $\mathbf{E}_{Int}$ and $\mathbf{E}_{Bool}$ are the identity relation, is a congruence on $\mathbf{A}^1_{si}$.

Thm. 2.2 Assuming that $\mathbf{E}_{Bool}$ is the largest congruence on a model of **Bool**, **E** is the largest congruence on **A**.

Proof See Appendix II. ∎

The above theorem implies that the observable equivalence relations on the domains in **A** completely extract its observable behavior in the sense that in the quotient algebra **A/E** induced by **E** on **A**, every value is distinguishable from each other.

2.2.4.2 Reduced Algebras

It is technically cumbersome to deal with a type algebra having distinct but observably equivalent values, so we introduce the notion of a reduced algebra.

Def. 2.11 An algebra **A** of type **D** is called *reduced* if and only if for each $D' \in \Delta'$, $E_{D'}$ is the identity relation. ∎

So all members in every domain of a reduced type algebra are distinguishable. For example, $\mathbf{A_{si}}$ is reduced, whereas $\mathbf{A^1_{si}}$ is not. **B**, the model of **Bool**, is also reduced.

Given an algebra **A**, we can get *its reduced algebra* by taking the quotient of **A** w.r.t. $E = \{ E_{D'} \mid D' \in \Delta' \}$, since **E** is a congruence on **A**. The reduced algebra corresponding to **A** is

$$\mathbf{A/E} = [\{ V_{D'}/E_{D'} \mid D' \in \Delta' \} ; \{ g_\sigma \mid \sigma \in \Omega \}], \text{ where}$$

$$g_\sigma([v_1], \ldots, [v_n]) = [f_\sigma(v_1, \ldots, v_n)].^9$$

The principal domain of the reduced algebra corresponding to an algebra of **D** having no observers, where **D** is not **Bool**, will have a single element. The reduced algebra corresponding to $\mathbf{A^1_{si}}$ has as its principal domain

$$SQ^1/E_{\text{Set-Int}} = \{ \{ \langle\rangle \}, \{ \langle 1 \rangle \}, \{ \langle -1 \rangle \}, \{ \langle 2 \rangle \}, \{ \langle -2 \rangle \},$$
$$\{ \langle 0, 1 \rangle, \langle 1, 0 \rangle \}, \{ \langle 0, -1 \rangle, \langle -1, 0 \rangle \}, \ldots \}$$

2.2.5 Behavioral Equivalence of Type Algebras

As was stated at the beginning of this section, in order to abstract the observable behavior of a type algebra, we must abstract from (i) multiple representations of the values of a data type in the type algebra as well as from (ii) different representational structures used for the values in different type algebras. The observable equivalence relation discussed above does the first task. It identifies representations having the same observable behavior. For the second task, we employ the standard algebraic concept of isomorphism.

---

9. It can be easily shown that **A/E** is also a type algebra.

By combining the two, we define the behavioral equivalence relation on type algebras as follows:

Def. 2.12 Type algebras $\mathbf{A}_1$ and $\mathbf{A}_2$ are *behaviorally equivalent* if and only if the reduced algebra corresponding to $\mathbf{A}_1$ is isomorphically equivalent to the reduced algebra corresponding to $\mathbf{A}_2$. ∎

We later show that the above definition indeed captures the desired intuition that two behaviorally equivalent algebras have the same observable behavior. By this, we mean that an interpretation of a ground term *e* in one algebra behaves the same way as an interpretation of *e* in the other algebra, when manipulated by the operations. (Informally speaking, a computation results in equivalent values in two related type algebras.)

The isomorphic equivalence of two type algebras is stronger than the isomorphism of the two type algebras if considered as they are. If D does not have any defining type, then isomorphic equivalence is the same as the isomorphism. However, if two type algebras are considered in the expanded form in which they have a domain corresponding to every data type $D'' \in (D)^*$ and a function corresponding to every operation of D'', then isomorphic equivalence is same as isomorphism. Since we do not wish to carry all this information in a type algebra and consider a type algebra in the expanded form, we assume that for each D' in Δ, the models of D' defining $V^1_{D'}$ and $V^2_{D'}$ as the value sets of D' are isomorphically equivalent and there is a bijection $\Phi_{D'}$ from $V^1_{D'}$ to $V^2_{D'}$ defined by the isomorphic equivalence relation. We thus do not use any arbitrary bijection from $V^1_{D'}$ in $\mathbf{A}_1$ to $V^2_{D'}$ in $\mathbf{A}_2$ to show isomorphic equivalence between $\mathbf{A}_1$ and $\mathbf{A}_2$. Instead, we build the bijections bottom up establishing correspondence between the values in the two algebras. The set $\{ \Phi_{D'} \mid D' \in \Delta \}$ induces a bijection Φ_D from V^1_D to V^2_D so that $\Phi = \{ \Phi_{D'} \mid D' \in \Delta' \}$ is an isomorphism from $\mathbf{A}_1$ to $\mathbf{A}_2$.

Def. 2.13 Given two type algebras $\mathbf{A}_1$ and $\mathbf{A}_2$ such that for each $D' \in \Delta$, the models defining $V^1_{D'}$ and $V^2_{D'}$ as the value sets of D' are isomorphically equivalent, which defines a bijection $\Phi_{D'} : V^1_{D'} \rightarrow V^2_{D'}$, $\mathbf{A}_1$ and $\mathbf{A}_2$ are *isomorphically equivalent* if and only if there is a bijection Φ_D from V^1_D to V^2_D such that $\Phi = \{ \Phi_{D'} \mid D' \in \Delta' \}$ is an isomorphism from $\mathbf{A}_1$ to $\mathbf{A}_2$. ∎

Note that both $\mathbf{A}_1$ and $\mathbf{A}_2$ above are either deterministic or the corresponding functions in $\mathbf{A}_1$ and $\mathbf{A}_2$ have the same amount of nondeterminism.

For examples, the models of **Bool** are isomorphically equivalent. The type algebras $\mathbf{A}_{si}$ and $\mathbf{A}^1_{si}$ of **Set-Int** are behaviorally equivalent because $\mathbf{A}_{si}$ and $\mathbf{A}^1_{si}$ /E are isomorphically equivalent. We can define three other type algebras of **Set-Int** which are similar to $\mathbf{A}^1_{si}$. The type algebras $\mathbf{A}^2_{si}$, $\mathbf{A}^3_{si}$, and $\mathbf{A}^4_{si}$ have sets represented by finite ordered sequences of nonrepeating integers, finite ordered sequences of repeating integers, and finite (unordered) sequences of repeating integers respectively; the definitions of various functions are appropriately given. It can be shown that the type algebras $\mathbf{A}_{si}$, $\mathbf{A}^1_{si}$, $\mathbf{A}^2_{si}$, $\mathbf{A}^3_{si}$, and $\mathbf{A}^4_{si}$ are behaviorally equivalent.

Note that two behaviorally equivalent type algebras need not have the same amount of nondeterminism. In fact, one could be deterministic whereas the other could be nondeterministic because the possible results returned by a nondeterministic function on an input in such a nondeterministic algebra are observably equivalent.

From the definitions of isomorphic equivalence and behavioral equivalence, we have the following:

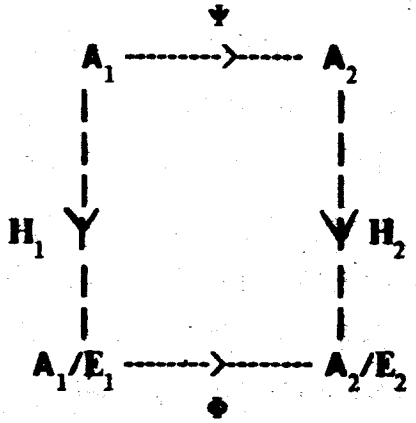
Thm. 2.3 $\mathbf{A}_1$ is isomorphically equivalent to $\mathbf{A}_2 \Rightarrow \mathbf{A}_1$ is behaviorally equivalent to $\mathbf{A}_2$.

Proof Assume $\mathbf{A}_1$ and $\mathbf{A}_2$ are isomorphically equivalent. Let E_1 and E_2 be the sets of observable equivalence relations on $\mathbf{A}_1$ and $\mathbf{A}_2$ respectively. Then, $\mathbf{A}_1/E_1$ and $\mathbf{A}_2/E_2$ can be shown to be isomorphically equivalent. (By Theorem 2.2, E_1 is the largest congruence on $\mathbf{A}_1$ and E_2 is the largest congruence on $\mathbf{A}_2$.) So, $\mathbf{A}_1$ and $\mathbf{A}_2$ are behaviorally equivalent. ∎

Thm. 2.4 The behavioral equivalence relation on type algebras is an equivalence relation.

Proof The reflexivity and symmetry property are obvious from the definition. The transitivity can be proved from the fact that composition of two isomorphisms is also an isomorphism. ∎

The behavioral equivalence of type algebras A_1 and A_2 can be expressed as



such that the above diagram commutes, i.e.,

$$\Phi \,.\, H_1 = H_2 \,.\, \Psi. \qquad (\dagger)$$

(The function f . g has the same behavior as applying g first and then applying f on the result.) E_1 and E_2 are congruences consisting of observable equivalence relations on A_1 and A_2 respectively; A_1/E_1 and A_2/E_2 are the reduced algebras corresponding to A_1 and A_2 respectively; and, Φ is the isomorphism defined by the isomorphic equivalence of A_1/E_1 and A_2/E_2. H_1 and H_2 are the homomorphisms induced by the congruences E_1 on A_1 and E_2 on A_2 respectively. The equation (†) defines the set Ψ of many to many mappings $\{ \Psi_{D'} : V^1_{D'} \rightarrow V^2_{D'} \mid D' \in \Delta \cup \{ D \}\}$ relating A_1 and A_2. In Appendix II, we discuss for two behaviorally equivalent type algebras A_1 and A_2, how a many to many mapping $\Psi_D : V^1_D \rightarrow V^2_D$ can be constructed from the set of many to many mappings $\{ \Psi_{D'} \mid D' \in \Delta \}$, where for each $D' \in \Delta$, $\Psi_{D'}$ is a many to many mapping from $V^1_{D'}$ to $V^2_{D'}$ defined by behaviorally equivalent models A'_1 and A'_2 of D' defining $V^1_{D'}$ and $V^2_{D'}$ respectively. We also show that the above definition of behavioral equivalence indeed captures the desired property that the set of interpretations of a ground term are 'equivalent' in behaviorally equivalent type algebras.

Thm. 2.5 For behaviorally equivalent algebras $\mathbf{A}_1$ and $\mathbf{A}_2$, for every ground term e of type $D'' \in (D)^*$, for every $v \in \{ e |_{\mathbf{A}_1} \}$, there is a $v' \in \{ e |_{\mathbf{A}_2} \}$ such that $\langle [v], [v'] \rangle \in \Phi_{D''}$, and vice versa.

Proof See Appendix II. ∎

The following theorem expresses that the distinguishability and observable equivalence of ground terms are invariant over behaviorally equivalent type algebras.

Thm. 2.6 For behaviorally equivalent $\mathbf{A}_1$ and $\mathbf{A}_2$, for any ground terms e_1 and e_2 of type D'', $\{ [e_1 |_{\mathbf{A}_1}] \} = \{ [e_2 |_{\mathbf{A}_1}] \} \Leftrightarrow \{ [e_1 |_{\mathbf{A}_2}] \} = \{ [e_2 |_{\mathbf{A}_2}] \}$.

Proof See Appendix II. ∎

{ [...] } stands for a set of equivalence classes.

2.2.6 Definition of a Data Type

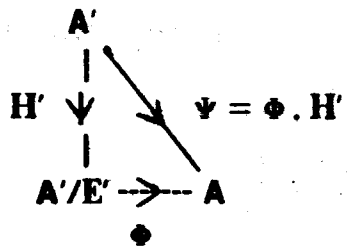
The behavioral equivalence relation on type algebras abstracts their observable behavior as shown above and captures the meaning of a data type.

Def. 2.14 A data type D is an equivalence class of algebras of type D defined by the behavioral equivalence relation. ∎

Let $\mathbf{M}_D$ stand for the set of all behaviorally equivalent algebras of type D. Every $\mathbf{A}$ in $\mathbf{M}_D$ is called a *model* of D as we have captured the semantics of the operations of D. The principal domain of a model $\mathbf{A}$ defines a *value set* of D. If a model in D is a reduced algebra, then it is called a *reduced* model. Since isomorphically equivalent algebras have the same amount of nondeterminism, all reduced models of D are either deterministic or all are nondeterministic (see p. 47). If a reduced model in D is nondeterministic, then the interpretation of an operation in every reduced model has, informally speaking, the same amount of nondeterminism. When we wish to present a data type D, we will do so by presenting an element of $\mathbf{M}_D$ as the representative of $\mathbf{M}_D$. We call this model the *denotation* of D. We often use a reduced model as the denotation of a data type.

We can order algebras in $\mathbf{M}_D$ using the onto homomorphism relation. Given two

algebras A_1 and $A_2 \in M_D$, $A_1 \leq A_2$ if and only if A_1 is an onto homomorphic image of A_2, when A_1 and A_2 are considered in their expanded form. The relation $\leq$ can be shown to be a partial order. A reduced model A of D is the least model in M_D upto isomorphic equivalence. It is also called *final* in M_D because there is a onto homomorphism from every model A' of D in M_D to A as depicted in the following diagram.



Def. 2.15 **Set-Int** is the set of all algebras behaviorally equivalent to A_{si}. ∎

So, A_{si}, A^1_{si}, A^2_{si}, A^3_{si}, and A^4_{si} are models of **Set-Int**. It can be verified that all models of **Bool** are behaviorally equivalent type algebras of **Bool**. We will use **B** as the denotation of **Bool** and A_{si} as the denotation of **Set-Int**.

It should be clear from the above definition that a data type D not having any observers consists of all type algebras of D. This is so because the definition of behavioral equivalence of type algebras depends only on the behavior of the observers.

We now compare our definition of a data type with those of Zilles [77] and the ADJ group [23]. They require a data type to be a set of all isomorphic (isomorphically equivalent to be exact) type algebras, which abstracts only the representation details from the algebras. (They assume that a data type has only deterministic operations). In their approach, a data type whose models are the reduced algebras is distinct from another data type whose models have distinct observably equivalent values even though both data types have the same observable behavior. For example, the data type consisting of models isomorphically equivalent to A_{si} would be different from the data type consisting of models isomorphically equivalent to A^1_{si}. From a programmer's point of view, both the data types are the same and cannot be distinguished. We do not understand the motivation for making the above distinction. Our definition of a data type is stronger than theirs, and it does not make the above distinction. It not only abstracts from the representations of the

values in a type algebra, but it also considers representations to be distinguishable only if they can be distinguished by the operations. It is based on the programming language view of a data type.

2.2.7 Observable Equivalence and Distinguishability of Terms

Since every value in the value set $\mathbf{V_D}$ defined by a model **A** of **D** is an interpretation of some ground term of type **D**, the observable equivalence relation and distinguishability relation on $\mathbf{V_D}$ induce the observable equivalence relation and distinguishability relation on the ground terms of type **D** as follows:

Two ground terms e_1 and e_2 of type **D** are *observably equivalent* w.r.t. **A** if and only if the possible interpretations of e_1 in **A** are *observably equivalent* to the possible interpretations of e_2 in **A**. And, e_1 and e_2 are *distinguishable* w.r.t. **A** iff they are not observably equivalent w.r.t. **A**.

For example, the ground terms **Insert(Insert(Null, 2), 3)** and **Insert(Insert(Null, 1), 2)** of type **Set-Int** are distinguishable w.r.t. $\mathbf{A_{si}}$, as their interpretations {2, 3} and {1, 2} in $\mathbf{A_{si}}$ are distinguishable, whereas **Insert(Insert(Null, 2), 3)** and **Insert(Insert(Null, 3), 2)** are observably equivalent w.r.t. $\mathbf{A_{si}}$, because they have the same interpretation in $\mathbf{A_{si}}$. The observable equivalence and distinguishability relations on ground terms of **D** w.r.t. **A** have the properties of the observable equivalence and distinguishability relations on $\mathbf{V_D}$ in **A**; remarks and observations made in Subsection 2.2.4 hold for them also.

Using the fact that all models of **D** are behaviorally equivalent and Theorem 2.6, it can be shown that every model of **D** induces the same observable equivalence relation on the ground terms of **D**. So we can say that the above relations are independent of a model and are relations on ground terms of **D**. We can use a reduced model to derive the observable equivalence relation on the ground terms of **D**.

Distinguishability and observable equivalence of the ground terms of **D** are useful in understanding the behavior of **D**. These relations characterize the behavior of **D** in the same way as these relations on the values of a type algebra characterize the behavior of the type algebra. Distinguishability captures the informal notion of the ground terms being

unequal. The models of a data type also induce observable equivalence and distinguishability relations on ground terms of type $\mathbf{D}' \in \Delta$ involving the operations of **D** in the same way as above. Understanding of the observable equivalence relation on the ground terms is helpful in writing a specification of a data type, as discussed in the next chapter. A specification of a data type can be viewed as a way to describe the observable equivalence relations on ground terms.

We can also define the observable equivalence relation on terms (possibly involving variables) as follows:

Given terms e_1 and e_2 of type $\mathbf{D}' \in \Delta'$, let X be the set of variables in e_1 and e_2; e_1 and e_2 are *observably equivalent* if and only if for some $\mathbf{A} \in \mathbf{M_D}$, for every **A**-instance V of X, the possible interpretations of $e_1[X/V]$ in **A** are observably equivalent to the possible interpretations of $e_2[X/V]$ in **A**. And, e_1 and e_2 are *distinguishable* if and only if they are not observably equivalent.

2.3 Exceptional Behavior of a Data Type

So far we have assumed that every operation of a data type D returns a normal value of its range type for any input in its domain. This assumption is not realistic, as it glosses over an important component of the behavior of a data type. In this section, we discuss the exceptional behavior of a data type. We relax the constraint that every operation terminates normally: An operation can terminate either normally by returning a value or by signalling an exception. For example, we modify the behavior of the operation **Choose** on the empty set; henceforth, we assume that it signals an exception instead of returning the integer 0. We discuss the assumptions made in the formalism about the behavior of the exception handling mechanism of a host programming language supporting the abstract data type mechanism. We extend the formalism introduced in the previous section to model the exceptional behavior.

2.3.1 Assumptions about Exception Handling Mechanism

We consider the exception handling mechanism an integral component of a host programming language supporting the data type facility. The exception handling mechanism performs two functions: *Signalling* the exceptions and *handling* the exceptions [52]. Signalling is the way a program notifies its caller of an exceptional condition, and handling is the way the caller responds to such a notification. A module implementing a data type must provide an adequate interface with the rest of the programming language for exception handling. Such an interface can be designed by *naming* the exceptions signalled by the operations along with the specification of information carried as arguments to the *exception handlers*. We will not be concerned with the semantics of the exceptional handling mechanism of a programming language in this thesis; we rather consider the exceptional handling mechanism insofar as it interacts with the data type mechanism.

Liskov and Snyder [50] discuss two models of structured exception handling - the *resumption model* and the *termination model*. In the resumption model, it is possible to resume the operation invocation signalling an exception after the exception has been handled. In the termination model, the operation invocation is assumed to be completed

once it signals an exception. Liskov and Snyder describe many advantages of the termination model over the resumption model. In particular, the behavior of the handlers for the exceptions signalled by an operation is separated from the behavior of the operation in the termination model approach; this maintains the modular structure of the operations. In the resumption model, on the other hand, the behavior of the handlers becomes a part of the operation behavior. Though there is not sufficient experience to suggest which among the two models is better suited for abstract data types, we have decided to adopt the termination model approach because of its simplicity.

In a language supporting call-by-name argument passing mechanism (or in fact, any mechanism in which the argument evaluation takes place inside the procedure body), it is possible to implement a data type whose operations can handle the exceptions signalled by the evaluation of their arguments. Few recently designed programming languages support such an argument passing mechanism for at least two reasons: (i) Its semantics is quite complex, and (ii) it is inefficient to implement. Most programming languages support call-by-value, call-by-object [52], or call-by-reference mechanism; with these mechanisms, it is not possible to implement a data type having an operation that handles exceptions signalled by the evaluation of its arguments. We assume in our work that an operation does not handle any exception signalled by the evaluation of its arguments, rather such exceptions are handled in a module in which the operation is invoked, as arguments are evaluated inside this module. Every operation is assumed to expect normal values as arguments.[10]

If an operation takes multiple arguments, many arguments may signal exceptions. The order in which the exceptions are signalled and handled depends upon the evaluation order of the arguments of a procedure invocation in the host programming language; we do not address this issue in the thesis. We would like our formalism to be compatible with any reasonable ordering scheme adopted in the host programming language.

---

10. However, our approach for defining a data type is general and flexible enough to model a data type having operations that handle exceptions signalled by its arguments. We simply have to extend the formalism proposed in this section. A data type with such behavior can also be specified by extending the specification language to be proposed in the next chapter.

We adopt CLU's view of a data type that the handlers associated with the exceptions signalled by the operations of a data type are not a part of the data type. This view keeps the behavior of the handlers separate from the type behavior and maintains the modular structure of the type mechanism. A user of a data type has the flexibility of associating different handlers for an exception in different contexts. We will not discuss the behavior of the handlers in our research.

Exceptions signalled by the operations are distinguished by naming them. An exception can carry information as its arguments from the place where the exception is signalled, and this information can be used by a handler associated with the signalled exception. An operation can signal many exceptions to exhibit different properties of an input.

For illustration, we consider the data type *bounded stack of integers, of size* $\leq$ *100*, denoted by **Stk-Int-100**. **Stk-Int-100** is an instantiation of the *parameterized stack* example in [31]; it has the following operations:

Null	a constant denoting the empty stack of integers.
Push	inserts a given integer i at the end of a given stack s. It signals the exception overflow(s, i) if the given stack is of size $\geq$ 100. A handler for overflow may examine the stack and remove the useless elements to make space for the new element, or it may do something else.
Pop	removes the last integer inserted into a given nonempty stack s. When invoked on the empty stack, it returns the empty stack back.
Top	returns the last integer inserted into a given nonempty stack s. It signals the exception no-top() if s is empty. No-top does not take any argument.
Replace	replaces the last integer inserted into a given nonempty stack s by a given integer i. It signals the exception can't-replace(i) on the empty stack.
Empty	tests whether a given stack is empty or not.

For **Stk-Int-100**, Δ = { **Int, Bool** } and Ω = { **Null, Push, Pop, Top, Replace, Empty** }.

2.3.2 Formalism

We discuss extensions of the formalism introduced in the previous section to model the exceptional behavior of the operations. We discuss modifications to the definitions and their implications. Some important definitions will be fully presented. The discussion and results of Section 2.2 are applicable once these modifications are incorporated.

We first extend the definition of a type algebra given in Subsection 2.2.1. We want to keep the normal values of every data type separate from the exceptions, because the exceptions have totally different behavior as compared to the normal values, and because the exceptions should not be typed. In addition to a domain corresponding to every $D' \in \Delta'$ containing the normal values of D', a modified type algebra has a new domain of exceptions denoted as EXV. EXV consists of all exceptions (or *exception values*) signalled by the operations of $D'' \in (D)^*$, where for every *exception name ex* of arity $D_1 \times \ldots \times D_n$, and each v_i of type D_i, $ex(v_1, \ldots, v_n)$ is called an exception value. The exception domain EXV in a type algebra **A** of **D** is specified incrementally. EXV in **A** inherits the exception domain of a model **A'** of $D' \in \Delta$ whose principal domain $V_{D'}$ is being used in **A**. The exception values signalled by the functions interpreting the operations of **D** are explicitly specified. Let *exv* stand for an exception value $ex(v_1, \ldots, v_n)$. If an operation σ signals, this is modeled as its interpretation f_σ returning an element of EXV.

We now present the modified type algebra:

Def. 2.16 An algebra **A** of type **D** is a heterogeneous algebra

$[\{ V_{D'} \mid D' \in \Delta' \}, EXV; \{ f_\sigma \mid \sigma \in \Omega \}]$, where

(i) for every defining type $D' \in \Delta$, $V_{D'}$ is a value set of D' defined by a model of D'. $V_{D'}$ consists only of the normal values returned by the constructors of D',

(ii) EXV is the exception domain including the exception domain of a model of D' defining $V_{D'}$ for each $D' \in \Delta$, and the exception values signalled by the operations of **D**,

(iii) for every $\sigma \in \Omega$, its interpretation f_σ is a total function of the appropriate arity. If D' is the range of σ, f_σ either results in a normal value in $V_{D'}$ or returns an exception value. If any argument to f_σ is in EXV, f_σ is not defined on these arguments,[11] and

(iv) V_D is the smallest set closed under finitely many applications of the functions corresponding to the constructors of D (i.e., $\{ f_\sigma \mid \sigma \in \Omega_C \}$). V_D only contains the normal values resulting from the constructors. ∎

Recall that by assumption, even if f_σ is nondeterministic, it behaves deterministically on an input on which it signals. We assume that for every $D' \in \Delta'$, it is possible to distinguish the normal values from the exceptions; this assumption is implicit in every programming language supporting exception handling.

2.3.2.1 Terms, Exception Terms, and Interpretations

In addition to terms as defined in Subsection 2.2.3, we have exception terms defined as follows.

Def. 2.17 For every exception name *ex* of arity $D_1 \times \ldots \times D_n$, $ex(e_1, \ldots, e_n)$ is an *exception* term if each e_i is a term of type D_i. ∎

An exception term not having any variables is called a *ground exception* term.

An interpretation of a ground term *e* in a type algebra A is not defined if any of its subterms interprets to an exception value. So, Proposition 2.1 in Subsection 2.2.3 gets modified to

---

11. An equivalent interpretation is to have f_σ signal a distinguished exception value, say abort() for example. We have not chosen this interpretation because it gives the impression of the exception value being passed as an argument to the operation. If we wish to model a data type with an operation handling exceptions signalled by the evaluation of its arguments, we cannot make the above assumption. An operation σ could return normal values even when its arguments signal exceptions, so f_σ could return a normal value in that case.

Prop. 2.3 An interpretation of a ground term of type $D'' \in (D)^*$ in an algebra **A** of type D is either a normal value, an exception value, or undefined. ∎

If an interpretation of *e* is an exception value or is undefined, then *e* has a unique interpretation in **A**. An interpretation of an instantiated term as well as a term in **A** are similarly defined. Proposition 2.2 in Subsection 2.2.3 directly extends to a modified type algebra.

An interpretation of an exception ground term $ex(e_1, ..., e_n)$ in **A** is defined only if each $e_i|_{\mathbf{A}}$ is a normal value of type D_i; then, $ex(e_1, ..., e_n)|_{\mathbf{A}} = ex(e_1|_{\mathbf{A}}, ..., e_n|_{\mathbf{A}})$. Otherwise, $ex(e_1, ..., e_n)|_{\mathbf{A}}$ is undefined. The definition of an interpretation of an instantiated exception term and of an exception term in **A** can be given using the above definitions.

2.3.2.2 Examples of Modified Type Algebras

The type algebras $\mathbf{A}_{si}$ and $\mathbf{A}^1_{si}$ of Set-Int given in Subsection 2.2.2 are modified to incorporate the exceptions. We will use the symbols $\mathbf{A}_{si}$ and $\mathbf{A}^1_{si}$ to stand for the modified type algebras.

$$\mathbf{A}_{si} = [\{ S, Z, B \}, EXV; \{ Nu, In, Re, Ha, Si, Ch \}],$$

The Choose operation signals the exception **no-element**, which is included in EXV; so

$$Ch(\varnothing) \triangleq \textbf{no-element}(),$$

instead of 0. Otherwise, the definitions of the functions remain the same. Similarly, for $\mathbf{A}^1_{si}$, we have

$$\mathbf{A}^1_{si} = [\{ SQ^1, Z, B \}, EXV; \{ Nu^1, In^1, Re^1, Ha^1, Si^1, Ch^1 \}],$$

where $Ch^1(\langle\rangle) \triangleq$ **no-element()**, and the definitions of other functions remain the same.

We present a type algebra $\mathbf{A}_{stk}$ of Stk-Int-100.

$$\mathbf{A}_{stk} = [\{ SQ', Z, B \}, EXV; \{ Nu, Pu, Po, To, Re, Em \}],$$

where Z and B are the value sets defined by the models of Int and Bool respectively. And, SQ' is the set of all sequences of integers of length ≤ 100.

$$SQ' = \{ \langle\rangle, \langle 0\rangle, \langle 1\rangle, \langle -1\rangle, \langle 2\rangle, \langle -2\rangle, \langle 0, 0\rangle, \langle 0, 1\rangle, \langle 0, -1\rangle, \ldots, \}$$

The interpretations of the operation names are defined as follows:

$$\mathbf{Nu} \triangleq \langle\rangle$$

$$\mathbf{Pu}(\langle i_1, \ldots, i_m \rangle, \mathbf{i}) \triangleq \begin{cases} \mathbf{overflow}(\langle i_1, \ldots, i_m \rangle, \mathbf{i}) & \text{if } m \geq 100 \\ \langle i_1, \ldots, i_m, \mathbf{i} \rangle & \text{otherwise} \end{cases}$$

$$\mathbf{Po}(\langle i_1, \ldots, i_m \rangle) \triangleq \begin{cases} \langle\rangle & \text{if } m = 0 \\ \langle i_1, \ldots, i_{m-1} \rangle & \text{otherwise} \end{cases}$$

$$\mathbf{To}(\langle i_1, \ldots, i_m \rangle) \triangleq \begin{cases} \mathbf{no\text{-}top}() & \text{if } m = 0 \\ i_m & \text{otherwise} \end{cases}$$

$$\mathbf{Re}(\langle i_1, \ldots, i_m \rangle, \mathbf{i}) \triangleq \begin{cases} \mathbf{can't\text{-}replace}(\mathbf{i}) & \text{if } m = 0 \\ \langle i_1, \ldots, i_{m-1}, \mathbf{i} \rangle & \text{otherwise} \end{cases}$$

$$\mathbf{Em}(\langle i_1, \ldots, i_m \rangle) \triangleq \begin{cases} \mathbf{T} & \text{if } m = 0 \\ \mathbf{F} & \text{otherwise.} \end{cases}$$

Henceforth, by a type algebra, we mean a modified type algebra unless stated otherwise.

2.3.2.3 Observable Behavior and Distinguishability

The definition of **Bool** given in Subsection 2.2.4 remains the same, because no boolean operation signals.

As was stated earlier, if the operations of a data type exhibit exceptional behavior, its values can also be distinguished due to its exceptional behavior. If a sequence of operations signals an exception on one value and does not signal on the other, then the two values are distinguishable. If a sequence of operations signals on both values, the two values are distinguishable if the sequence signals different exceptions. Thus the behavior of the values of a data type can also be observed using the exception handling mechanism of the host programming language. Even if a data type does not have any defining types, its values can be distinguished if its operations signal exceptions.

We define the distinguishability relation on V_D and the distinguishability relation on the exception domain **EXV** in **A** mutually recursively, using the distinguishability relations on the domains corresponding to the defining types. It should be made sure that arguments to exception names are such that the two definitions are well founded. The definition of distinguishability on exception values incorporates that (i) two exceptions

having different names are distinguishable, and (ii) two exceptions having the same name but distinguishable arguments are distinguishable.

Def 2.18 Given two exception values $ex_1(v_1, \ldots, v_n)$ and $ex_2(v'_1, \ldots, v'_m)$ in **EXV**, they are *distinguishable* iff (i) $ex_1 \neq ex_2$, or (ii) if $ex_1 = ex_2$ and $m = n$, then for some $1 \leq i \leq m$, v_i is distinguishable from v'_i. Two exception values are *observably equivalent* iff they are not distinguishable. ∎

We denote the observable equivalence relation on **EXV** by $\mathbf{E_{EXV}}$.

Def. 2.19 For an algebra **A** of type D having no defining types and whose operations do not signal, all values in $\mathbf{V_D}$ are *observably equivalent.* ∎

Def 2.20 Two normal values v_1 and v_2 in $\mathbf{V_D}$ of an algebra **A** of type D are *distinguishable* iff there exists a term with one variable of type D, expressed as $c(x)$, such that one of the following conditions holds:

(i) the instantiated terms $c[x/v_1]$ and $c[x/v_2]$ interpret to distinguishable exception values in **A**,

(ii) $c[x/v_1]$ interprets to a normal value and $c[x/v_2]$ interprets to an exception value or vice versa, and

(iii) $c[x/v_1]\|_{\mathbf{A}}$ and $c[x/v_2]\|_{\mathbf{A}}$ are normal values and $\{ c[x/v_1]\|_{\mathbf{A}} \}$ is distinguishable from $\{ c[x/v_2]\|_{\mathbf{A}} \}$. ∎

Note that in the above definition of distinguishability, we have not included the case in which exactly one of $c[x/v_1]$ and $c[x/v_2]$ is not defined because the condition (ii) above takes care of it.

Def. 2.21 Two normal values v_1 and v_2 are *observably equivalent* iff they are not distinguishable. ∎

Theorem 2.1 of Subsection 2.2.4 extends to the above definition of observable equivalence relation. $\mathbf{E_{EXV}}$ is also an equivalence relation.

We extend the definitions of congruence, homomorphism, and isomorphism for

type algebras having exception domains. The mappings from the normal domains of a type algebra $\mathbf{A}_1$ to the corresponding normal domains of another type algebra $\mathbf{A}_2$ induce a mapping Φ_{EXV} from the exception domain $\mathbf{EXV}_1$ in $\mathbf{A}_1$ to the exception domain $\mathbf{EXV}_2$ in $\mathbf{A}_2$. The exception names act like operations; they preserve these mappings. Given

$$\mathbf{A}_1 = [\{ V^1_{D'} \mid D' \in \Delta' \}, \mathbf{EXV}_1 ; \{ f^1_\sigma \mid \sigma \in \Omega \}]$$
$$\mathbf{A}_2 = [\{ V^2_{D'} \mid D' \in \Delta' \}, \mathbf{EXV}_2 ; \{ f^2_\sigma \mid \sigma \in \Omega \}],$$

for every exception name *ex* of arity $\mathbf{D}_1 \times \ldots \times \mathbf{D}_n$,

$$\langle ex(v_1, \ldots, v_n), ex(\Phi_{D_1}(v_1), \ldots, \Phi_{D_n}(v_n)) \rangle \in \Phi_{EXV}.$$

Theorem 2.2 modified to say that $E = \{ E_{D'} \mid D' \in \Delta' \} \cup \{ E_{EXV} \}$ is the largest congruence in **A** holds; the proof is similar to the proof of Theorem 2.2. E captures the normal as well as the exceptional behavior of the functions of a type algebra **A**.

We define a reduced algebra in the same way as in Subsection 2.2.4 using the congruence E. The definition of behavioral equivalence relation on type algebras is the same as in Subsection 2.2.5. The definition of isomorphic equivalence used in the definition of behavioral equivalence is extended by including the mapping Φ_{EXV} in the family Φ and requiring Φ_{EXV} also to be a bijection. The theorems of Subsection 2.2.5 exhibiting that the definition of behavioral equivalence of unmodified type algebras indeed captures the desired intuition extend to the modified type algebras. The results and proofs are modified to incorporate the fact a ground term *e* (respectively, an instantiated term *e*[*X*/*V*]) may interpret to a normal value, an exception value, or be undefined (see Appendix II).

A data type **D** is defined in the same way as in Subsection 2.2.6 as a set of behaviorally equivalent type algebras. Let $\mathbf{M}_D$ stand for this set. Every model in $\mathbf{M}_D$ now has the exception domain **EXV**. The observable equivalence and distinguishability relations on the ground terms of type D are defined as in Subsection 2.2.7. We incorporate the facts that two ground terms whose interpretation in every model in $\mathbf{M}_D$ are undefined, are observably equivalent, and that if one of the ground terms has an undefined interpretation whereas the other does not, then the two ground terms are distinguishable.

2.3.2.4 Comparison with Goguen's Approach

Our approach is similar to Goguen's approach [20, 21] of modeling the exceptional behavior of a data type in the sense that exceptions are named and can have arguments. However, there are crucial differences in the two design philosophies. In Goguen's approach, the definition of a new data type can possibly extend the definitions of its defining types. This is so because the exceptions (called *not-ok* values in [20]) are typed just like the normal values (called *ok* values in [20]). Instead of having a single domain of exceptions, Goguen partitions a value set of **D** into the exception values and the normal values; the exception value part of the value set expands as new types using **D** are defined. For example, the definition of **Stk-Int-100** would extend the definition of **Int** by defining a new integer **no-top** (which is a not-ok value). We consider this as violating the modular structure of the definitions.

The OBJ language of Goguen and Tardo [21] allows the handlers for the exceptions signalled by the operations to be specified as a part of the type specification, thus making the type behavior complex. We suspect that they adopt this approach because of their attempt to develop the algebraic semantics of a complete programming language including the control structures. So, they do not separate the semantics of the exception handling mechanism from the data type.

In contrast, we have concentrated on the behavior of data types only. We have separated the exception handling mechanism from the data type mechanism. We have only considered components of the exception handling mechanism related to the type definition mechanism. We do not consider the behavior of exception handlers as a part of a data type for reasons discussed earlier. We believe that the type mechanism should only provide an adequate interface to the exception handling mechanism of the host programming language. We separate the exception domain from the domain of normal values as exceptions have different behavior from the normal values. We do not type exceptions either because doing so seems meaningless. In this way, we have been able to define the behavior of the operations of a data type completely and uniformly, without extending the definition of any of its defining types thus preserving the modular structure of the type mechanism.

2.3.3 A Simpler Approach

In this subsection, we discuss another approach for modeling the exception behavior of a data type, which is simpler than the approach discussed earlier. This approach has been generally assumed in the literature on algebraic specification of data types when the authors do not wish to discuss the exception behavior of the operations [29, 77]. The ADJ group's work [23] is an attempt to formalize it, and Guttag [31] embeds it in a rich way in a specification language. We discuss this approach for two reasons: (i) our discussion is simpler and more natural than that of [23], (ii) our discussion would place the works of those who have implicitly or explicitly assumed this approach of modeling exceptional behavior on a firm basis, and (iii) our discussion provides a semantic basis of Guttag's specification language.

In this approach, exceptions signalled by operations having the same range are not distinguished and no information is passed with an exception to its handler. An operation on an input either returns a normal value or signals an exception **failure**. For example, the operations **Push**, **Pop**, and **Replace** signal the same exception **failure**. Every operation is assumed to expect normal values as arguments. If an argument to an operation signals **failure**, then the operation propagates it by signalling it.

Such exceptional behavior of the operations can be modeled by extending the domain of every $D' \in \Delta'$ in an algebra **A** of type **D** (as defined in Subsection 2.2.1) with a special exception **failure** ; we denote it by $\mathbf{err}_{D'}$. Whenever an operation σ signals **failure**, its interpretation f_σ in **A** returns $\mathbf{err}_{D'}$, where D' is the range type of σ. So we have

$$\mathbf{A} = [\{ V_D \cup \{ \mathbf{err}_D \}\} \cup \{ V_{D'} \cup \{ \mathbf{err}_{D'} \} \mid D' \in \Delta \}; \{ f_\sigma \mid \sigma \in \Omega \}].$$

If any of the x_i's is $\mathbf{err}_{D_i}$, then $f_\sigma(x_1, \ldots, x_n) = \mathbf{err}_{D'}$, i.e., f_σ is *strict* with respect to its arguments. We assume that for every $D' \in \Delta'$, it is possible to distinguish between the normal values and the exception value $\mathbf{err}_{D'}$.

We modify the definition of **Bool** given in Section 2.2. The model **B** of **Bool** is extended to have the exceptional value $\mathbf{err}_b$.

$$\mathbf{B}' = (\{\{ \mathbf{true}, \mathbf{false}, \mathbf{err}_b \}\}; \{ T, F, \vee, \sim, \wedge, \Rightarrow, \Leftrightarrow \}),$$

where the definitions of the boolean operations remains the same on normal values.

Besides, every function is strict. **Bool** is defined as the set of all type algebras isomorphic to **B'**.

We discuss a type algebra $\mathbf{A}'_{\mathbf{stk}}$ of **Stk-Int-100**.

$$\mathbf{A}'_{\mathbf{stk}} = [\{ SQ' \cup \{ \mathbf{err}_{\mathbf{stk}} \}, Z', B' \}; \{ Nu', Pu', Po', To', Re', Em' \}],$$

where $B' = B \cup \{ \mathbf{err}_{\mathbf{b}} \}$,

$Z' = Z \cup \{ \mathbf{err}_{\mathbf{i}} \}$,

$$Nu' \triangleq \langle\rangle$$

$$Pu'(\langle i_1, \ldots, i_m \rangle, i) \triangleq \begin{cases} \mathbf{err}_{\mathbf{stk}} & \text{if } m \geq 100 \\ \langle i_1, \ldots, i_m, i \rangle & \text{otherwise} \end{cases}$$

$$Po'(\langle i_1, \ldots, i_m \rangle) \triangleq \begin{cases} \langle\rangle & \text{if } m = 0 \\ \langle i_1, \ldots, i_{m-1} \rangle & \text{otherwise} \end{cases}$$

$$To'(\langle i_1, \ldots, i_m \rangle) \triangleq \begin{cases} \mathbf{err}_{\mathbf{i}} & \text{if } m = 0 \\ i_m & \text{otherwise} \end{cases}$$

$$Re'(\langle i_1, \ldots, i_m \rangle, i) \triangleq \begin{cases} \mathbf{err}_{\mathbf{stk}} & \text{if } m = 0 \\ \langle i_1, \ldots, i_{m-1}, i \rangle & \text{otherwise} \end{cases}$$

$$Em'(\langle i_1, \ldots, i_m \rangle) \triangleq \begin{cases} T & \text{if } m = 0 \\ F & \text{otherwise.} \end{cases}$$

The theory discussed in Section 2.2 directly extends to the above algebras also. The definition of the interpretation of a term in Subsection 2.2.3 easily extends. A ground term of type D' or an instantiated term may interpret to $\mathbf{err}_{D'}$. The definition of distinguishability of values of D in a type algebra also extends in a straightforward manner. We want to add to the definition that (i) every normal value of D is distinguishable from the exceptional value $\mathbf{err}_{D}$, and (ii) two normal values v_1 and v_2 in V_D of **A** are also distinguishable if there is a term $c(x)$ such that $c[x/v_1]$ interprets to an exceptional value, whereas $c[x/v_2]$ interprets to a normal value, or vice versa.

The behavioral equivalence relation on modified type algebras is a simple extension of the definition given in Subsection 2.2.5. The modified definition of isomorphic equivalence requires that every mapping $\Phi_{D'}$ in Φ maps the exception value $\mathbf{err}_{D'}$ in $\mathbf{A}_1$ to $\mathbf{err}_{D'}$ in $\mathbf{A}_2$. Other conditions remain the same in the definition. A data type

D is a set consisting of all behaviorally equivalent type algebras of the above kind. The observable equivalence and distinguishability relations on ground terms are defined in the same way as in Subsection 2.2.7.

2.4 Mutually Recursive Data Types

We have assumed so far that data types can be designed hierarchically one at a time and that the data types on which a data type **D** depends can be designed independently of **D**. These assumptions are not valid for a subclass of data types. In some cases, it may be more meaningful to associate an operation with a collection of data types, instead of a single data type; for example the *conversion operations* between the data types *fixed point number* and *floating point number*. Or a group of data types may be mutually dependent such that they cannot be defined one at a time, for example, data types *picture*, *contents*, *component*, and *view* in [32] are mutually recursive. In the latter case, the dependency relation on data types as defined in Section 2.1 will have cycles.

For the above cases, we consider groups of mutually recursive data types together as one entity, and define direct dependency and dependency relation on such groups and nonrecursive data types in an analogous manner so that the relations do not have any cycles. A group of mutually recursive data types can be then defined hierarchically when considered as one entity.

Let $\boldsymbol{D}$ stand for a group of new types being defined together. Let Δ stand for the set of their defining types, assumed to be defined elsewhere, and Ω stand for the set of their operation names.

A type algebra for a group of new data types $\boldsymbol{D}$ is a straightforward extension of a type algebra for a single data type **D**. It has a domain corresponding to every $D \in \boldsymbol{D}$ in addition to the domains corresponding to every defining type $D' \in \Delta$ and the exception domain EXV. It also has a total function (deterministic or nondeterministic) corresponding to every operation name in Ω. Instead of having a single principal domain as in case of a type algebra for a single data type, we have many distinguished domains in a type algebra for $\boldsymbol{D}$: Every domain corresponding to $D \in \boldsymbol{D}$ is a distinguished domain. In order for the distinguished domains to be nonempty, it is necessary that at least one of the data types in $\boldsymbol{D}$ has a basic constructor (a constructor that does not take any argument of a type in $\boldsymbol{D}$). Furthermore, all the distinguished domains must be constructible mutual recursively.

The theory developed for a single data type easily extends to a group of mutually recursive data types. We can directly extend the definition of the interpretation of a term

in a type algebra defined above. The observable equivalence and distinguishability relations can be similarly defined on V_D for each $D \in \boldsymbol{D}$. They induce the observable equivalence and distinguishability relations on the ground terms of type D. Behavioral equivalence relation on type algebras can also be defined analogously.

A group of mutually recursive data types $\boldsymbol{D}$ is a set of all behaviorally equivalent type algebras of the above kind. Every type algebra in the equivalence class is a model of $\boldsymbol{D}$. A model of $\boldsymbol{D}$ defines a value set of each $D \in \boldsymbol{D}$, which is the distinguished domain corresponding to D in the model.

3. Specification of an Abstract Data Type

In this chapter, we discuss a method for specifying abstract data types. Like the definition method, the specification method is hierarchical and modular. We describe a specification language in which data types having nondeterministic operations and having operations exhibiting exceptional behavior can be specified. The main goal in designing the language has been to develop a good notation for expressing the design of the data component of programs. The specification language should be as flexible as possible to enable a designer to conveniently express his/her intent. We do not restrict a specification to specify a single data type only, instead a specification in general specifies a set of related data types sharing a common behavior. A specification only expresses properties particular to the data type(s) being specified. Properties common to all data types, for instance, the minimality property, are not specified. They are instead assumed in the semantics of the specification language.

Since a data type is a set of models, its specification(s) must capture the properties common to these models. The specification must specify the syntactic structure as well as the observable behavior of these models. There can be many ways to do this. One way is to present a model that acts as a representative of the above set. For instance, the definition of a denotation of a data type D can serve as its specification; as an example, the model $\mathbf{A_{si}}$ of **Set-Int** can serve as a specification of **Set-Int**. A data type is specified in this way in the model approach [3], which is briefly discussed in Section 1.2. This method has a disadvantage that since a particular representation of the values of the data type is used to specify the data type, there is a danger of the irrelevant properties of the model being associated with the data type. This shortcoming of the model approach can be circumvented by choosing an appropriate semantics of the specification method as in [3].

Another way is to specify the properties that characterize the observable behavior of all models of a data type. We adopt this approach, which is called the axiomatic approach in Section 1.2. We specify the observable behavior as a finite set of properties of the operations of D. These properties are expressed abstractly without referring to any particular model of D and without assuming any particular representation of the values of

D. They are presented as first order formulas relating sequences of operations that return observably equivalent values. The reasons for choosing the axiomatic approach are:

(i) A theory of a data type can be directly developed from its axiomatic specification without referring to any other domain of discourse,

(ii) our work can be integrated with the work on the development of axiomatic systems for reasoning about control structures [17, 36] and the automation of the verification process, and

(iii) the methodology for proving the correctness of an implementation of the data type with respect to its specification is simple and natural for a wide class of specifications.

Instead of allowing arbitrary first order formulas, we restrict the axioms to be equations because

(i) an equational specification is amenable for deducing the properties of a data type (see the next chapter, where the proof theory of a data type is developed from its specification; also see Musser [60] for discussion of a theorem prover for equational specifications),

(ii) an equational specification is easier for a programmer to understand (see [29] for a discussion on viewing equational axioms as recursive programs),

(iii) certain desirable properties of specifications can be guaranteed by putting constraints on equations [28],

(iv) an equational specification has been found to be more suitable for semi-automatically deriving an implementation of a data type [64, 68], and

(v) a model can be more easily constructed from a equational specification than from a specification whose axioms use existential quantifiers [16].

Our specification language allows a specification to introduce a finite set of auxiliary functions to express the properties of the operations. An auxiliary function is not an operation of a data type; rather it is a helping function in a specification. So it is a part

of a specification of a data type, and not a part of the data type itself.[1] The use of auxiliary functions in a specification is a necessity, because if axioms are restricted to be equations without auxiliary functions, many data types cannot be specified [2, 53, 71, 43].[2] With the help of a finite set of auxiliary functions, one can specify using a finite set of equations,

(i) any data type with a recursively enumerable (r.e.) value set and a finite set of total deterministic computable functions [28, 43], and

(ii) any data type that can be specified using a recursively enumerable set of equations, restricted conditional equations, or positive conditional equations [43].

In this sense, our specification language is quite expressive. (For a detailed discussion of the expressive power of an equational language with auxiliary functions and how it compares with other algebraic languages for specifying data types, see [43].) Besides, we have found auxiliary functions convenient and useful in expressing the properties of complex operations; the judicious choice of auxiliary functions often results in specifications that are relatively easier to write and understand as compared with equivalent specifications written without using the auxiliary functions.[3]

We discuss the specification language in the first section. Different components of a specification are described. The semantics of a specification is given in the second section. It is defined to be a set of related data types sharing the common behavior captured by the specification. In the third section, we state what it means for a data type to be (precisely) specifiable by a specification; equivalence among specifications is defined. The fourth section discusses the specification of the data type *boolean*. In the fifth section, we discuss two structural properties of a specification, consistency and behavioral

---

1. An auxiliary function should not be confused with an internal procedure needed in an implementation of a data type to implement its operations. (Chapter 5 discusses internal procedures.) An auxiliary function however serves the same purpose in a specification as an internal procedure in an implementation. It is not available to the users of a data type, and is used only for expressing and proving properties of the data type from its specification.

2. We conjectured in [43] that even if axioms are allowed to be conditional equations (restricted, positive, or unrestricted), there are many interesting data types that cannot be specified without auxiliary functions.

3. Guttag [31] rightly compares the use of auxiliary functions in a specification with the use of subroutine (procedure) abstraction while writing a complex piece of software.

completeness, expressed in terms of relationships among the set of data types specified by the specification. The consistency property requires that a specification specifies at least one data type. The behavioral completeness property requires that a specification completely specifies the observable behavior of the operations on intended inputs; it rules out only intentional incompleteness in a specification. In the sixth section, we compare our specification language with the works of Zilles [77], Guttag et al. [29, 31], the ADJ group [23], Goguen [20], Burstall and Goguen [7], Goguen and Tardo [21], and Nakajima et al. [62].

3.1 Specification Language

The specification language has a single syntactic unit, called a *specification module* (or simply a *specification*), which in general specifies a set of related data types. We first discuss specifications of hierarchically structured (nonrecursive) data types; at the end of the section we discuss a specification for mutually recursive data types.

We will use a single name to stand for any of the data types specified by a specification. We may use the same name as the name of its specification whenever it is possible to disambiguate from the context whether a name refers to a data type or its specification. When we consider more than one specification of a data type, we use different names for different specifications. Though a long name for a concept may convey information about the behavior of the concept, the long name can be inconvenient to use, so we allow abbreviations for long names to be introduced in a specification preceded by the symbol *as*. Let D stand for a type being specified by a specification S.

A specification in general has four components:

(i) *Operations*,

(ii) *Auxiliary Functions*,

(iii) *Restrictions*, and

(iv) *Axioms*.

The operations component specifies the syntactic properties of D, and the restrictions component and the axioms component specify its semantic properties. We illustrate different components of a specification using the specifications given in Figures 3.1 and 3.2. Figure 3.1 is a specification of **Set-Int**. Figure 3.2 is a specification of a set **Stk-Int** of data types; the data type **Stk-Int-100** defined in Chapter 2 is in this set.

A specification is hierarchically structured; it refers to the specifications of data types other than D assuming that these specifications are given elsewhere. Data types other than D may have already been specified, or they will be specified later. For example, the specification of **Set-Int** in Figure 3.1 refers to a specification of a data type **Int**. We assume that **Int** is specified elsewhere. Since a specification of **Int** can specify a set of data types, **Int** in Figure 3.1 stands for any data type in the set.

Figure 3.1. Specification of Set-Int

Operations

Null	**: → Set-Int**	*as* **Ø**
Insert	**: Set-Int X Int → Set-Int**	
Remove	**: Set-Int X Int → Set-Int**	
Has	**: Set-Int X Int → Bool**	*as* $x_2 \in x_1$
Size	**: Set-Int → Int**	*as* $\#(x_1)$
Choose	**: Set-Int → Int**	*nondeterministic*
	→ no-element()	

Restrictions

#(s) = 0 ⇒ Choose(s) *signals* **no-element**

Axioms

Remove(Ø, i) ≡ Ø
Remove(Insert(s, i1), i2) ≡ if i1 = i2 then Remove(s, i1) else Insert(Remove(s, i2), i1)
i ∈ Ø ≡ F
i1 ∈ Insert(s, i2) ≡ if i1 = i2 then T else i1 ∈ s
#(Ø) ≡ 0
#(Insert(s, i)) ≡ if i ∈ s then #(s) else #(s) + 1
Choose(s) ∈ s ≡ T

Whenever we introduce a new construct of a specification in this section, we informally discuss its meaning for motivation and clarity of exposition. As was stated above, the precise semantics of a specification will be given in the next section.

3.1.1 Operations

This component specifies (i) the domain and range, and (ii) the names of the exceptions signalled by every operation of D on its intended inputs, along with the types of the arguments to the exceptions. It is a sequence of specifications of the following form:

$$\sigma : D_1 \times \ldots \times D_n \rightarrow D'$$
$$\rightarrow ex_1(D_{11}, \ldots, D_{1m_1})$$
$$\vdots$$
$$\rightarrow ex_k(D_{k1}, \ldots, D_{km_k}),$$

where $D_1 \times \ldots \times D_n$ is the domain of σ and D' is its range. σ signals exceptions having names $ex_1, \ldots, ex_k$, whose argument types are also specified. If an operation is specified to signal an exception, the exception must be listed in its syntactic specification. If σ does not take any argument, then it is a constant of its range type. If an exception name *ex* does not take any argument, it is expressed as *ex*() or simply *ex*. The operations component of a specification of D indirectly specifies the Δ and Ω of D.

When an abbreviation is introduced for an n-ary operation name, we can specify how the abbreviation distributes over the arguments using the argument place holders $x_1, \ldots, x_n$. For example, the operation Has of Set-Int is abbreviated to '$\in$' and it is used as '$x_2 \in x_1$.' We discuss later (Subsection 3.1.5) how nondeterministic operations are specified.

3.1.2 Auxiliary Functions

This component is optional; it exists if auxiliary functions are used in writing the *Axioms* and the *Restrictions*. As was discussed before, auxiliary functions are introduced to enhance the expressive power of the specification language and to make the language more flexible so that specifications are easier to write and understand. We do not recommend choosing auxiliary functions randomly to express the behavior of the operations. Instead, they should be chosen with care. An auxiliary function should embody a subsidiary procedural abstraction needed to express the operation behavior. It is a good design practice to completely specify an auxiliary function even if its behavior is needed only for a subset of its input domain. Furthermore, if an auxiliary function is of the result type D, it should not have to construct values that cannot be constructed by the constructors of D.

Every auxiliary function is deterministic, and there are no restrictions associated with it.[4] For example, the specification of **Stk-Int** in Figure 3.2 uses the auxiliary function **Size**.

We specify the domain and range of every auxiliary function used in the specification in the same way as the operations. Let A_f stand for the set of all auxiliary functions used in a specification. An auxiliary function may use a data type not in Δ' $(= \Delta \cup D)$ as a component of its domain or as its range; we call such a data type as an *auxiliary* type. Like a defining type, every auxiliary type is assumed to be specified elsewhere. Let A_t stand for the set of auxiliary types used by the auxiliary functions in A_f. If a specification does not have the auxiliary functions component, then $A_f = \emptyset$ and $A_t = \emptyset$.

We extend the definition of a term in Subsection 2.2.3 to include terms constructed using the auxiliary functions and the operation symbols of the auxiliary types.

Def. 3.1 An *auxiliary term* of type $D' \in \bigcup_{D'' \in \{D\} \cup A_t} (D'')^*$ is defined inductively as

(i) a term of type D',

(ii) if $\sigma \in A_f$ such that its domain is $D_1 \times \ldots \times D_n$ and its range is D', then '$\sigma(e_1, \ldots, e_n)$' is an auxiliary term of type D' if and only if each e_i is an auxiliary term of type D_i. ∎

Clearly, if A_f and A_t are the empty sets, the definitions of an auxiliary term and a term coincide. An auxiliary exception term can be defined by replacing terms by auxiliary terms in the definition of an exception term in Subsection 2.3.2. Henceforth, by a term, we mean an auxiliary term, and by an exception term, we mean an auxiliary exception term, unless stated otherwise.

---

4. These constraints on auxiliary functions are imposed for convenience and simplicity. Our formalism would work equally well if these constraints are not imposed.

Figure 3.2. Specification of Stk-Int

Stk-Int *as* **Stk**

Operations

Null : → **Stk**
Push : **Stk X Int** → **Stk**
→ **overflow(Stk, Int)**
Pop : **Stk** → **Stk**
Top : **Stk** → **Int**
→ **no-top()**
Replace : **Stk X Int** → **Stk**
Empty : **Stk** → **Bool**

Auxiliary Functions

Size : **Stk** → **Int** *as* **#(x)**

Restrictions

Pre(**Pop(s)**) :: ~ **Empty(s)**
Pre(**Replace(s, i)**) :: ~ **Empty(s)**

Empty(s) ⇒ **Top(s)** *signals* **no-top()**
Push(s, i) *signals* **overflow(s, i)** ⇒ **#(s)** ≥ **100**

Axioms

Pop(Push(s, i)) ≡ **s**
Top(Push(s, i)) ≡ **i**
Replace(s, i) ≡ **Push(Pop(s), i)**
Empty(Null) ≡ **T**
Empty(Push(s, i)) ≡ **F**
#(Null) ≡ **0**
#(Push(s, i)) ≡ **#(s)** + **1**

3.1.3 Restrictions

The restrictions and axioms components of a specification specify the normal as well as the exceptional behavior of the operations. They also define the auxiliary functions, if any, used in the specification. The axioms component specifies the normal behavior of the operations. The exceptional behavior is specified as a separate layer over the normal behavior. This is achieved by specifying *restrictions* on the operations in the restrictions component. An axiom in the axioms component holds only if the operations used in the axiom satisfy the specified restrictions. The restrictions component is an extension of the *Restrictions Specifications* of Guttag [31].

The restrictions component is a set of restrictions; every restriction is associated with an operation. There are two kinds of restrictions:

(i) *Preconditions*, and

(ii) *Exception Conditions*.

Every exception listed in the syntactic specification of an operation should have an associated restriction specifying the input condition when the exception is signalled or may be signalled by the operation. The boolean conditions in the exception conditions for an operation must be disjoint. Another constraint on the boolean conditions when they use nondeterministic operations is discussed later. As is stated in the first chapter, for operations having complex behavior, it may be very difficult to specify conditions on their inputs under which they signal a particular exception. This approach of specifying the exceptional behavior is not suitable for such operations.

3.1.3.1 Preconditions

The precondition restriction for an operation specifies the subset of its input domain on which the operation behavior is of interest. The operation is expected to be invoked on inputs in this subset; it is the user's responsibility to ensure this. The operation behavior is specified only on these inputs; it is left unspecified on inputs outside the subset because it does not matter. An operation can either signal an exception or return a value on an input not satisfying the precondition. For example, in certain applications, we may

not care how the operation **Replace** in Figure 3.2 behaves on the empty stack as it is never going to be invoked on the empty stack. It could either return a stack value or signal an exception. Also see [51, 32] for more examples of such operations. If a specification commits to a particular behavior on an input not satisfying the precondition, for instance signalling an exception, many implementations would be ruled out. Our approach is to encourage a designer to specify only that portion of the data type behavior which is of interest to him and allow the rest of the type behavior to be left unspecified so that an implementor has the maximum flexibility.

The precondition restriction for an operation $\sigma \in \Omega$ is specified as:

$$Pre(\sigma(X)) :: \mathbf{P}(X),$$

where $\mathbf{P}(X)$ is a boolean term having $x_1, \ldots, x_n$ (the input X) as its variables, and it cannot signal on X. The axioms involving σ hold only if the input to every invocation of σ satisfies the precondition $\mathbf{P}(X)$. If the *Restrictions* component does not specify a precondition for an operation, the operation is assumed to be specified for its entire syntactic domain, i.e., its precondition is T. For example, **~ Empty(s)** is the precondition for **Pop** as well as **Replace** in the specification of **Stk-Int** in Figure 3.2. The specification does not specify the behavior of these operations for the empty stack. No precondition is specified for any other operation, so their preconditions are T. Similarly, no precondition is specified for any operation in the specification of **Set-Int** in Figure 3.1. If a precondition different from T is specified for an operation σ, σ is said to have a *nontrivial* precondition. Let $\mathbf{P}_\sigma$ stand for the precondition for σ.

If an operation σ does not signal on an input not satisfying its precondition, it cannot return an arbitrary value. If σ is a constructor, as for example, the operations **Pop** and **Replace** in Figure 3.2, the result must be constructible by the constructors of **D** using inputs satisfying the associated preconditions. Similarly, if σ is an observer, then it must return a value of its result type.

3.1.3.2 Exception Conditions

There are two kinds of exception conditions:

(i) *Required* exception conditions, and

(ii) *optional* exception conditions.

A required exception condition for an operation σ is expressed as

$R(X) \Rightarrow \sigma(X)$ *signals* $ex(e_1, \ldots, e_k)$,

stating that if the input X satisfies the precondition P_σ and the boolean condition $R(X)$, which is a boolean term, then the operation σ must signal the exception *ex* having e_1, ..., e_k as the arguments to its handler(s). The exception name *ex* is of arity $D_1 \times \ldots \times D_k$, and each e_i is a term of type D_i having variables only from the set $\{ x_1, \ldots, x_n \}$. For example, in Figure 3.1, the operation **Choose** is specified to signal the exception **no-element** on the empty set. In Figure 3.2, the operation **Top** signals **no-top** on the empty stack. We call the above exception condition required because the operation is required to signal the exception. It is possible to specify an operation signalling different exceptions for different subsets of inputs.

In certain applications, it may be restrictive to require that an operation signal an exception when its input satisfies a condition. At the same time, it may not be desirable to leave the operation behavior completely unspecified. Instead, we would like to place constraints on the behavior. If an input to the operation satisfies the specified condition, the operation is specified to have the option of either signalling the specified exception or returning a normal value. In case the operation chooses not to signal, it must behave as specified by the axioms. Optional exception conditions are introduced to capture such behavior of an operation. An optional exception condition is expressed as

$\sigma(X)$ *signals* $ex(e_1, \ldots, e_k) \Rightarrow O(X)$,

stating that in case σ signals an exception *ex* having e_1, ..., e_k as arguments and the input X satisfies the precondition P_σ, then the input X must also satisfy the boolean condition $O(X)$, a boolean term.

Optional exceptions are especially useful for specifying a set of similar data types having values whose capacity (size) has different upper bounds. It is possible to state a size

requirement on the values of the data type, but at the same time not be very restrictive about the requirement. An implementor could decide on the exact bound based on convenience insofar as the specified bound condition is met. Such behavior of a data type is specified by stating that the constructors have the option to signal exceptions.

For example, in the data type **Stk-Int-100** defined in the previous chapter, the operation **Push** signals if its stack argument is of size 100. If the desired requirement is that a stack value be able to store at least 100 integers, this behavior of **Push** is very restrictive. It rules out a implementation supporting stack values of size > 100, even though the implementation has the desired behavior except that **Push** does not signal exactly on stacks of size 100, but rather on stacks of size 128. We specify the desired requirement in Figure 3.2 by stating that **Push** optionally signals; whenever **Push** signals overflow, its stack argument must be at least of size 100. In this way, a specification specifies the least upper bound on the size of the values of a data type, and the responsibility of deciding the exact upper bound is delegated to an implementor. Such a specification is flexible and not restrictive.

3.1.3.3 Discussion

Note that the nontrivial precondition restrictions and the optional exception conditions leave the specification of the operations incomplete because the operation behavior is not completely specified on a subset of inputs. An operation could behave on such inputs in any way consistent with the specified behavior. That is why a specification in general specifies a set of related data types; the operations of these data types have the same behavior for a subset of their syntactic domains. For example, **Stk-Int** specifies data types having stack values whose size has different upper bounds ≥ 100. The operations of these data types behave the same way on stacks of size ≤ 100, except that **Pop** and **Replace** of different data types may behave differently on the empty stack. We call such incompleteness in a specification as *intentional incompleteness*, in contrast to unintentional incompleteness introduced because of the omission on the part of a designer in specifying the properties of the operations.

It should be intuitively clear that if no nontrivial precondition and no optional

exception condition are associated with any operation, and the axioms completely capture the observable behavior of the operations, then a specification specifies a single data type in case the specification of every defining type also specifies a single data type. We elaborate this informal statement later in the chapter.

3.1.4 Axioms

This component specifies the normal behavior of the operations in Ω and the auxiliary functions in $\mathbf{A_f}$ if they are used in a specification. The behavior is specified as a finite set of *equations* of the form '$e_1 \equiv e_2$,' where e_1 and e_2 are auxiliary terms of the same type; at least one of e_1 and e_2 must have its outermost symbol in $\Omega \cup \mathbf{A_t}$, otherwise an equation would not be specifying a property of **D**. '$e_1 \equiv e_2$' informally means that the sequences of operations expressed by the terms e_1 and e_2 have the same behavior, i.e., when values are substituted for variables in e_1 and e_2, the instantiated terms interpret to observably equivalent values. The symbol '$\equiv$' is interpreted as the observable equivalence relation. The equations attempt to capture the observable equivalence relations on ground terms defined by the data type(s) being specified, which is discussed in Chapter 2.

If a specification does not have the restrictions component (i.e., the operations do not signal exceptions and there is no nontrivial precondition associated with any operation), then the variables in an axiom are universally quantified: Any value of the appropriate type can be freely substituted for a variable.

If a specification has a restrictions component, then an axiom is interpreted in a different way; the variables in an axiom cannot be freely substituted. We must also consider the restrictions imposed on the operations appearing in the axioms. The values substituted for the variables must satisfy the following two conditions:

(i) For every operation σ having a nontrivial precondition $\mathbf{P_\sigma}$, the arguments to every invocation of σ in the axiom must satisfy $\mathbf{P_\sigma}$, and

(ii) an instantiation of any subexpression in the axiom must not interpret to an exception value.

The condition (ii) above is equivalent to requiring that an interpretation of an instantiation of e_1 or e_2 is neither undefined nor an exception value. For example, consider the axiom

Replace(s, i) ≡ Push(Pop(s), i) (*)

in the specification of **Stk-Int** in Figure 3.2. It applies only for the values of **s** for which **~ Empty(s)** holds, which is the precondition for both **Replace** and **Pop**. Furthermore, **Push** must not signal **overflow** on the result returned by **Pop**, which it cannot in any case. The equations characterize the normal behavior of the operations in this way.

It is often the case that two terms are observably equivalent only when a condition is placed on their variables; for example, in the second axiom in the specification of **Set-Int** in Figure 3.1, **Remove(Insert(s, i1), i2)** is observably equivalent to **Insert(Remove(s, i2), i1)** only if i1 and i2 are not equal. So, while writing the axioms, it is convenient to assume an auxiliary function **if-then-else** corresponding to every $D' \in \Delta' \cup A_f$. The definition of **if-then-else** is given as:

if-then-else : Bool X D' X D' → D' *as* **if** x_1 **then** x_2 **else** x_3

if T then x else y ≡ x
if F then x else y ≡ y.

Since these functions are used frequently, they are assumed to be implicitly defined whenever needed. They are not explicitly stated in the auxiliary functions component of the specification, and are not in A_f. If **Bool** is not a defining type, then **Bool** is assumed to be an auxiliary type. An axiom of the form 'e_1 ≡ **if** b **then** e_2' stands for the equation 'e_1 ≡ **if-then-else**(b, e_2, e_1).' We call 'e_1 ≡ **if** b **then** e_2' a *conditional equation*.[5] It is equivalent in its interpretation to the formula '$b \equiv T \Rightarrow e_1 \equiv e_2$.' An axiom of the form 'e_1 ≡ **if** b **then** e_{11} **else** e_{12}' stands for the equation 'e_1 ≡ **if-then-else**(b, e_{11}, e_{12}).' It is equivalent to the following two conditional equations

'e_1 ≡ **if** b **then** e_{11}'
'e_1 ≡ **if ~** b **then** e_{12}'

---

5. Note that a conditional equation as defined above is different from a positive conditional equation of the ADJ [71], in which the condition in the axiom can be expressed using ≡ positively. A conditional equation of the above form is called a *restricted* conditional equation in [43]. We have chosen such axioms because of simplicity, as even using positive conditional equations as axioms does not add to the expressive power of the specification language [43]. Furthermore, homomorphisms do not preserve positive conditional equations.

3.1.5 Specifying Nondeterministic Operations

If an operation is nondeterministic, this is specified using the symbol *nondeterministic* following its range specification, as for the **Choose** operation of **Set-Int** in Figure 3.1. The behavior of a nondeterministic operation is specified in the same way as of a deterministic operation. The restrictions component may specify a precondition, a set of required exception conditions, and a set of optional exception conditions for a nondeterministic operation. For a nondeterministic observer returning many possible results on an input, the axioms do not specify the results; instead, they specify the properties of the results. For example, the axiom specifying the behavior of the nondeterministic operation **Choose** of **Set-Int** on an nonempty set s states that a result returned by **Choose** on s must be an element of the set s. For a nondeterministic constructor, its behavior is characterized by specifying the results returned by the observers on the possible values constructed by it.

If a boolean condition in a restriction is expressed using nondeterministic operations, we require that for every input X, the boolean condition behaves deterministically, i.e., it returns either **T** or **F**. It is meaningless for a boolean condition to return **T** as well as **F** on X: In case of a precondition, the instantiated boolean condition returning **T** as well as **F** would mean that the input satisfies the precondition as well as does not satisfy the precondition. In case of an exception condition, this would mean that σ signals or may signal on the input as well as that σ does not signal on the input.

For an equational axiom '$e_1 \equiv e_2$' expressed using nondeterministic operations, we use the following interpretation: For an instantiation of the variables in the axiom allowed by the preconditions and restrictions, the set of possible values returned by the instantiated e_1 is observably equivalent to the set of possible values returned by the instantiated e_2 (i.e., for every choice of nondeterministic operations in e_1, the value returned by the instantiated e_1 is observably equivalent to a value returned by the instantiated e_2 for some choice of nondeterministic operations in e_2, and vice versa). We have rejected another possible interpretation which is that for any choice of nondeterministic operations in both e_1 and e_2, the values returned by the instantiated e_1 and e_2 are observably equivalent, because under this interpretation, the axiom does not hold when e_1 and e_2 exhibit nondeterministic

behavior; an equational axiom thus does not express any useful property. If an axiom is a conditional equation 'e_1 ≡ **if** b **then** e_2,' where the boolean condition b involves nondeterministic operations, then we require that for an instantiation of the variables x_1, ..., x_n, b behaves deterministically. As in case of a boolean condition in a restriction, an instantiation of b behaving nondeterministically and returning **T** as well as **F** does not make any sense in a conditional equation.

An alternate approach for specifying a nondeterministic operation would be to indirectly specify it by having the axioms specify its relation, which is deterministic. The relation can be specified using equations and conditional equations. However, the constraint that if the nondeterministic operation returns a normal value on an input, then the relation holds for the input and at least one result, cannot be expressed in terms of equations and conditional equations. This can be circumvented by assuming that every such relation satisfies the above constraint. If a nondeterministic operation signals on an input, some convention about the behavior of the relation on such an input must be decided. Using this approach, it is possible to specify the precise amount of nondeterminism an operation should have. However, we have adopted the former approach because of the following reasons:

(i) We do not want the specification to specify the precise amount of nondeterminism an operation should have; instead, we leave this decision to the designer of an implementation,

(ii) it seems more natural to directly specify the behavior of an operation than specifying the corresponding relation,

(iii) the semantics of a specification designed using the latter approach would have to be derived indirectly, as should be evident from the discussion in the next section, and

(iv) if we adopt the latter approach, the normal behavior of the nondeterministic operation would be indirectly specified by specifying its relation, whereas its exceptional behavior would be directly specified. We would like to avoid using two notations for the same concept.

But one major advantage of adopting the latter approach is that we do not have to develop any additional formalism for nondeterministic operations. The theory developed for

specifications specifying only deterministic operations applies to nondeterministic operations also.

3.1.6 Specification of Mutually Recursive Data Types

A specification for mutually recursive data types is similar to a specification for nonrecursive data types. Let ***D*** stand for an instance of a group of mutually recursive data types being specified. The specification is given either the name of some data type in ***D*** or a name different from the names of data types in ***D***. Like a specification of a nonrecursive data type, it has four components:

(i) *Operations*,

(ii) *Auxiliary Functions*,

(iii) *Restrictions*, and

(iv) *Axioms*.

The *Operations* component specifies the syntactic properties of the operations of ***D***. It is divided into subcomponents. There is a subcomponent entitled **D** corresponding to every data type D in ***D*** specifying the operations of **D**. So, a subcomponent is like the operations component of a nonrecursive data type as discussed above. Besides, there is another subcomponent entitled *Combined Operations*, which specifies the syntactic properties of the operations not belonging to any particular data type, but rather to the whole group ***D***. The remaining three components are the same as in a specification of a single data type. If ***D*** does not have any combined operations, the specifications of data types in ***D*** can be given separately like nonrecursive data types. However, the semantics of these specifications must be given together.

Henceforth, we discuss only nonrecursive data types. From the following discussion, it should be clear how to extend the results and the theory to mutually recursive data types. For instance, we can give the semantics of such a specification in a similar way as for nonrecursive data types (discussed in the next section) except that we will need to use type algebras defined in Section 2.4.

3.2 Semantics of Specification Language

The semantics of a specification S is defined to be a set of related data types. Each data type in the set is said to be specified by S. Let $\mathcal{D}(S)$ stand for this set. Since a specification S refers to other specifications assuming them to be *given*, for example, the specification of **Set-Int** refers to the specifications of **Int** and **Bool**, the semantics of S is given using their semantics. For a defining type $D' \in \Delta$ used in S, we assume that D' has a specification S' having a nonempty set of data types as its semantics; D' stands for any data type in $\mathcal{D}(S')$.

If S does not specify any nondeterministic operation, then every data type in $\mathcal{D}(S)$ can be shown to be deterministic. Operations of different data types in $\mathcal{D}(S)$ share the common behavior specified by S. Different data types differ in the way their operations behave on inputs not satisfying the preconditions specified for the operations and/or on inputs on which the operations are specified to have the option between signalling and returning a value. If the axioms do not completely capture the observable behavior of the operations, then data types in $\mathcal{D}(S)$ have operations having different behavior on input on which the axioms leave their behavior unspecified.

In case S specifies nondeterministic operations, then data types in $\mathcal{D}(S)$ also differ in the amount of nondeterminism their operations have. $\mathcal{D}(S)$ has data types in which the operations specified to be nondeterministic are deterministic as well as data types in which such operations have the maximum amount of nondeterminism allowed by S. For example, the semantics of the specification of **Set-Int** given in Figure 3.1 has a data type in which the operation **Choose** is deterministic, returning the maximum integer in a nonempty set s passed as the argument to **Choose**. It also has the data type **Set-Int** defined in the previous chapter in which the **Choose** nondeterministically picks any element of s. In general, a data type in $\mathcal{D}$(**Set-Int**) has the operation **Choose** return an element from a nonempty subset of s.

The semantics of a specification specifying nondeterministic operations is thus necessarily a set of data types differing in the amount of nondeterminism these operations have, even if the specification does not specify any precondition or any optional exception condition for the operations and the specification completely specifies the observable

behavior of the operations. This semantics of a specification is chosen because of our view that a specification should not constrain an implementation to have any precise amount of nondeterminism, and that the decision about how much nondeterminism an implementation should have, be left to the designer of the implementation. Since a specification serves as an interface between the programs using the data type and the implementation(s), every theorem derived from the specification, as discussed in the next chapter, must hold for a correct implementation when interpreted appropriately.

It is possible to write a specification in our language which specifies unbounded nondeterminism. (The term unbounded nondeterminism used here is different from the way it is used in [13, 35].) For example, in the specification of N_1 (a version of the data type *natural number*) in Figure 3.3 specifies unbounded nondeterminism because the operation **Pick** is specified to have unbounded nondeterminism. For such a specification there does not exist any data type having maximal amount of nondeterminism. We will precisely state the condition when a specification S specifies unbounded nondeterminism. For a specification specifying bounded nondeterminism, we define data types having maximal amount of nondeterminism allowed by the specification.

Instead of giving the semantics of S directly in terms of data types, we give its semantics as a set of (well formed) type algebras. Let $\mathcal{F}(S)$ stand for this set. We then partition this set using the behavioral equivalence relation on type algebras and get the set $\mathcal{D}(S)$ of data types. Each type algebra in $\mathcal{F}(S)$ is a model of some data type specified by S. We first assume that S does not use any auxiliary functions, i.e., $A_f = \emptyset$ and $A_t = \emptyset$. Later, we discuss the semantics of S assuming that $A_f \neq \emptyset$ and $A_t \neq \emptyset$.

3.2.1 Specifications without Auxiliary Functions

A type algebra in $\mathcal{F}(S)$ must have the syntactic structure as specified in the operations component of S and the observable behavior as specified by the axioms and the restrictions in S. $\mathcal{F}(S)$ is inductively defined; as in Chapter 2, we combine the basis and inductive steps into a single step. $\mathcal{F}(S)$ consists of all (well formed) type algebras of the form

$$\mathbf{A} = [\{ V_{D'} \mid D' \in \Delta' \}, EXV ; \{ f_\sigma \mid \sigma \in \Omega \}]$$

Figure 3.3. Specification of N_1

Operations

0	: → N_1	
S	: N_1 → N_1	
P	: N_1 → N_1	
	→ no-pred()	
=	: N_1 X N_1 → Bool	*as* $x_1 = x_2$
≥	: N_1 X N_1 → Bool	*as* $x_1 \geq x_2$
Pick	: → N_1	*nondeterministic*

Restrictions

x = 0 ⇒ P(x) *signals* no-pred()

Axioms

P(S(x)) ≡ x
x ≥ x ≡ T
x ≥ z ≡ if (x ≥ y ∧ y ≥ z) then T
S(x) ≥ x ≡ T
x ≥ S(x) ≡ F
x ≥ S(y) ≡ if ~ x ≥ y then F
x = y ≡ (x ≥ y ∧ y ≥ x)
Pick() ≥ 0 ≡ T

---

such that **A** *satisfies* the restrictions and the axioms in S, where for each D' ∈ Δ, $V_{D'}$ is the principal domain of an algebra **A'** ∈ *F*(S'). **A'** is a model of a data type D' in *D*(S').

We first discuss when a type algebra **A** satisfies restrictions; later we discuss the axioms. Let $X = \{ x_1, \ldots, x_n \}$ stand for all variables in an axiom or a restriction. Let $V = \{ v_1, \ldots, v_n \}$, where each v_i is a normal value of the appropriate type, stand for a **A**-instance of X, i.e., each v_i is an instance of x_i.

3.2.1.1 Restrictions

If a nontrivial precondition P_σ is specified for a constructor σ, then on an input V such that $P_\sigma[X/V]$ interprets to F, $f_\sigma(v_1, \ldots, v_n)$ either signals or returns a value constructible by the constructor functions using arguments satisfying their preconditions. It would be meaningless to allow f_σ to return an arbitrary value that cannot even be

constructed. For example, if a data type satisfying the specification in Figure 3.2 has its **Push** operation signal **overflow** on stacks of size 128, it is absurd to let the operation **Pop** return a stack of size 1000 when applied on the empty stack, the input that does not satisfy the precondition specified for **Pop**. Similarly, if σ is an observer, then $\mathbf{f}_\sigma(v_1, ..., v_n)$ either signals or returns a value in $\mathbf{V}_{D'}$, where D' is the result type of σ.

If the restrictions component specifies a required exception condition on σ as

$$\mathbf{R}(X) \Rightarrow \sigma(X) \textit{ signals } ex(e_1, \ldots, e_k),$$

then for every V, if both $\mathbf{P}_\sigma[X/V]$ and $\mathbf{R}[X/V]$ interpret to **T**, then $\mathbf{f}_\sigma(V)$ must signal the exception value $ex(e_1[X/V]\|_{\mathbf{A}}, ..., e_k[X/V]\|_{\mathbf{A}})$ for **A** to satisfy the above restriction.

If the restrictions component specifies σ to optionally signal an exception, i.e.,

$$\sigma(X) \textit{ signals } ex(e_1, \ldots, e_k) \Rightarrow \mathbf{O}(X),$$

then for every V such that $\mathbf{P}_\sigma[X/V]$ interprets to **T** and $\mathbf{f}_\sigma(V)$ signals the exception ex with the interpretations of $e_1[X/V], \ldots, e_k[X/V]$ as arguments to its handlers, $\mathbf{O}[X/V]$ must interpret to **T** for **A** to satisfy the above restriction.

Since the restrictions are assumed to completely specify the exceptional behavior of the operations, for every operation σ, the interpretation $\mathbf{f}_\sigma$ in **A** must be such that $\mathbf{f}_\sigma(v_1, \ldots, v_n)$ is a normal value if (i) $\mathbf{P}_\sigma[X/V]$ holds, (ii) none of $\mathbf{R}_i[X/V]$ holds, and (iii) none of $\mathbf{O}_j[X/V]$ holds.

3.2.1.2 Axioms

A (*behaviorally*) *satisfies* an equation '$e_1 \equiv e_2$' (or '$e_1 \equiv e_2$' *holds* in **A**) if and only if for every V, one of the following conditions holds:

(i) The instantiation of e_1 or of e_2 interprets to an exception or is undefined,

(ii) the input to an invocation of some $\mathbf{f}_\sigma$ on $v_1', \ldots, v_m'$ does not satisfy the precondition associated with σ (i.e., $\mathbf{P}_\sigma(v_1', \ldots, v_m')$ interprets to **F**) when the instantiations of e_1 and e_2 are interpreted, and

(iii) $\{ e_1[X/V]\|_{\mathbf{A}} \}$ is observably equivalent to $\{ e_2[X/V]\|_{\mathbf{A}} \}$.

In the previous section, we informally described the semantics of conditional equations using the auxiliary functions **if-then-else**. Here we formalize the discussion. To

check whether a conditional equation '$e_1 \equiv$ **if** b **then** e_2' holds in **A**, we extend **A** to include the interpretation of the auxiliary function **if-then-else** : **Bool** x D' x D' → D' corresponding to every D' ∈ Δ'. The interpretation $f_{\textbf{if-then-else}}$ in the extended algebra has the following behavior:

$f_{\textbf{if-then-else}}(T, v_1, v_2) \triangleq v_1$,
$f_{\textbf{if-then-else}}(F, v_1, v_2) \triangleq v_2$.

The interpretation of a conditional equation involving **if-then-else** can be verified to be equivalent to interpreting the formula '$b \equiv T \Rightarrow (e_1 \equiv e_2)$' as we require that b behave deterministically for every **A**-instance. Henceforth, we view a conditional equation as a formula '$b \Rightarrow e_1 \equiv e_2$' so that we do not have to consider the auxiliary functions **if-then-else**.

If a type algebra **A** is in $\mathcal{F}$(S), then we say that **A** *behaviorally satisfies* S, and call **A** a *model* of the specification S. Note that **A** satisfies the axioms under the interpretation of the symbol '≡' as the observable equivalence relation on the domains of a type algebra. If a model **A** of S satisfies the axioms interpreting '≡' as the identity relation as in Logic, we say that **A** *identically satisfies* S.

For example, the models $\mathbf{A}_{si}$ and $\mathbf{A}^1_{si}$ of the data type **Set-Int** discussed in Chapter 2 can be shown to be in $\mathcal{F}$(**Set-Int**). So, they are also the models of the specification of **Set-Int** given in Figure 3.1. $\mathbf{A}_{si}$ identically satisfies the specification of **Set-Int**. It should be easy to see that every reduced algebra in $\mathcal{F}$(S) identically satisfies a specification S because the observable equivalence relations are the identity relations.

Using the fact that the set E of observable equivalence relations on the domains in **A** above is a congruence, we have

Thm. 3.1 **A** ∈ $\mathcal{F}$(S) iff **A**/E ∈ $\mathcal{F}$(S). ∎

So, to check whether a type algebra **A** is in $\mathcal{F}$(S), we can check whether its reduced algebra **A**/E identically satisfies S. Using the above theorem, we get

Thm. 3.2 If **A** ∈ $\mathcal{F}$(S), then every type algebra behaviorally equivalent to **A** is in $\mathcal{F}$(S). ∎

3.2.2 Specifications with Auxiliary Functions

An auxiliary function is not a part of a data type, so a model in $F(S)$ cannot have any interpretation for the auxiliary functions. We first define an *extended* data type D_1 from D, whose operation set is $\Omega \cup A_f$ and the set of defining types is $\Delta \cup A_t$. If the *Auxiliary Functions* component is included in the *Operations* component in S, the modified specification S_1 is a specification of data types having the same syntactic structure as D_1, and S_1 does not use any auxiliary functions. We define $F(S_1)$ for the modified specification S_1 as discussed above. An algebra A_1 of type D_1 in $F(S_1)$ is

$$A_1 = [\{ V^1_{D'} \mid D' \in \Delta' \cup A_t \} ; \{ f^1_\sigma \mid \sigma \in \Omega \cup A_f \}].$$

So an auxiliary term can be interpreted in A_1. The axioms in S expressed using the auxiliary functions in A_f hold in A_1.

For every algebra A_1 of type D_1 in $F(S_1)$, we obtain an algebra A of type D in $F(S)$ as follows:

$$A = [\{ V_{D'} \mid D' \in \Delta' \} ; \{ f_\sigma \mid \sigma \in \Omega \}],$$

where for each $D' \in \Delta$, $V_{D'} = V^1_{D'}$, and $V_D \subseteq V^1_D$. A function f_σ is a restriction of f^1_σ to the domains of A such that V_D is the smallest set closed under finitely many applications of the functions in $\{ f_\sigma \mid \sigma \in \Omega_C \}$. V_D can be a proper subset of V^1_D, because S may use an auxiliary function having D as its range that constructs some extraneous values (see [70] for an example of such a specification).[6]

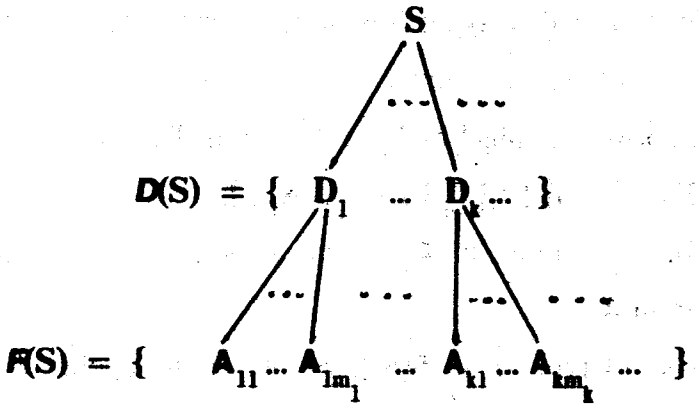
For example, the model A_{stk} of the data type Stk-Int-100 discussed in Chapter 2 can be shown to be in F(Stk-Int). We must extend A_{stk} to include the interpretation Si of the auxiliary function Size such that $Si(\langle i_1, \ldots, i_m \rangle) \triangleq m$, and use the extended algebra for proving that it satisfies the axioms and restrictions in Figure 3.2.

---

6. However, we do not encourage specifications in which auxiliary functions are of result type D and generate values not constructible by the constructors of D.

3.2.3 Semantics of a Specification

Using Theorem 3.2, we partition $\mathcal{F}(S)$ using the behavioral equivalence relation on type algebras, and get the set $\mathcal{D}(S)$ of data types as the semantics of S. A reduced algebra in every equivalence class in the partition on $\mathcal{F}(S)$ can serve as a representative of the data type defined by the equivalence class. This can be pictorially expressed as



where $D_1, \ldots, D_k, \ldots$, are the data types in $\mathcal{D}(S)$, and $A_{k1}, \ldots, A_{km_k}, \ldots$, are the models of a data type D_k.

It should be clear from the discussion in the last two subsections that the operations of different data types in $\mathcal{D}(S)$ share the behavior specified by S. However, they differ in

(i) the amount of nondeterminism they have, if specified to be nondeterministic by S,

(ii) their behavior on inputs not satisfying the preconditions specified by S,

(iii) their behavior on inputs satisfying the preconditions and optional exception conditions specified by S, and

(iv) their behavior on inputs on which their behavior is unintentionally omitted in S.

If S specifies σ to optionally signal on a subset of inputs, σ for different data types may or may not signal for some of the inputs in the subset. If the constructors are specified to optionally signal for expressing the size requirement on the values of a data type, different data types have different upper bounds on the size of their values.

For example, $\mathcal{D}$(Set-Int) defines different data types in which **Choose** behaves differently because it has different amounts of nondeterminism, as was discussed earlier.

***D*(Stk-Int)** has different data types whose operations **Pop** and **Replace** have different behavior on the empty stack, and the operation **Push** behaves differently on stacks of size ≥ 100. Some of the data types differ in the maximum size allowed of the stacks. The data type **Stk-Int-100** defined in Chapter 2 is in ***D*(S)**.

3.3 Specification of a Data Type and Equivalence of Specifications

Def. 3.2 A specification S *specifies* a data type D iff $D \in \mathcal{D}(S)$ (i.e., $\mathbf{M}_D \subseteq \mathcal{F}(S)$).[7] ∎

If a specification S specifies the data type D, the specification need not be precise in the sense that it may not completely specify the behavior of D; a portion of the behavior may not be, in fact, captured by S at all. There may be data types in $\mathcal{D}(S)$ different from D. We introduce the following stronger definition for specifications specifying deterministic operations only.

Def. 3.3.1 S *precisely specifies* D iff $\mathcal{D}(S) = \{ D \}$ (i.e., $\mathbf{M}_D = \mathcal{F}(S)$). ∎

The above definition requires that the specification of a defining type $D' \in \Delta$ also precisely specifies D'.

For a specification specifying nondeterministic operations, its semantics has data types differing in the amount of nondeterminism their operations have. nondeterminism allowed by S. We define a partial ordering on type algebras in $\mathcal{F}(S)$ which orders data types in $\mathcal{D}(S)$ based on the amount of nondeterminism in their operations that are specified to be nondeterministic by S. Instead of comparing two arbitrary type algebras in $\mathcal{F}(S)$, it is convenient to compare algebras having the same domains but differing in their functions.

Def. 3.4 Given two type algebras A and A' of D

$$A = [\{ V_{D'} \mid D' \in \Delta' \}, EXV ; \{ f_\sigma \mid \sigma \in \Omega \}]$$

$$A' = [\{ V_{D'} \mid D' \in \Delta' \}, EXV ; \{ f'_\sigma \mid \sigma \in \Omega \}],$$

A' is *at least as nondeterministic as* A, expressed as $A \leq_{nd} A'$, if and only if

for every operation $\sigma \in \Omega$, and for each $v_1, ..., v_n$,

$$\{ f_\sigma(v_1, \ldots, v_n) \} \subseteq \{ f'_\sigma(v_1, \ldots, v_n) \}.$$ ∎

Informally, the above means that every function in A' is at least as much nondeterministic

---

7. Recall that $\mathbf{M}_D$ is the set of all models of the data type D.

as the corresponding function in **A**. We say that $\mathbf{A} <_{nd} \mathbf{A}'$ if and only if $\mathbf{A} \leq_{nd} \mathbf{A}'$ and there is at least one nondeterministic function f'_σ in **A**′ such that for some $v_1, \ldots, v_n$, $\{ f_\sigma(v_1, \ldots, v_n) \} \subseteq \{ f'_\sigma(v_1, \ldots, v_n) \}$ and $\{ f_\sigma(v_1, \ldots, v_n) \} \neq \{ f'_\sigma(v_1, \ldots, v_n) \}$.
We can order the reduced models in $\mathcal{F}(S)$ using $\leq_{nd}$ relation.

Def. 3.5 A reduced model **A** in $\mathcal{F}(S)$ has *maximal amount of nondeterminism allowed by* S if and only if there is no reduced model **A**′ in $\mathcal{F}(S)$ such that $\mathbf{A} <_{nd} \mathbf{A}'$. ∎

If a reduced algebra $\mathbf{A} \in \mathcal{F}(S)$ has maximal amount of nondeterminism allowed by S, then it can be shown that any algebra behaviorally equivalent to **A** also has maximal amount of nondeterminism allowed by S. Using this, we get

Def. 3.6 A data type $D \in \mathcal{D}(S)$ has *maximal amount of nondeterminism allowed by* S if its reduced model has maximal amount of nondeterminism allowed by S. ∎

For example, the model $\mathbf{A}_{si}$ has maximal amount of nondeterminism allowed by the specification of **Set-Int** in Figure 3.1, so the data type **Set-Int** defined in Chapter 2 has maximal amount of nondeterminism allowed by the specification in Figure 3.1. It is easy to see that no model of the specification of N_1 in Figure 3.3 can have maximal amount of nondeterminism; given any model **A**, we can find a **A**′ such that $\mathbf{A} <_{nd} \mathbf{A}'$.

Def. 3.7 A specification S *specifies unbounded nondeterminism* if and only if $\mathcal{D}(S)$ is not empty and there does not exist a data type in $\mathcal{D}(S)$ with maximal amount of nondeterminism allowed by S. ∎

So, the specification of N_1 specifies unbounded nondeterminism because of the operation **Pick**. The specification of **Set-Int** specifies bounded nondeterminism as there are data types with maximal amount of nondeterminism allowed by the specification of **Set-Int** in $\mathcal{D}$(**Set-Int**). In this thesis, we have considered data types with operations having only finite nondeterminism, so we are interested in specifications that specify bounded nondeterminism. Henceforth, we assume that a specification S does not specify unbounded nondeterminism.

In case of a specification specifying nondeterministic operations, we have

Def. 3.3.2 S *precisely specifies* D if $\{ D \} = \{ D_{max} \mid D_{max} \in \mathcal{D}(S)$ and D_{max} has maximal amount of nondeterminism allowed by S $\}$. ∎

The above definition also covers the case 3.3.1 above, as in case of a specification specifying only deterministic operations, the set $\{ D_{max} \mid D_{max} \in \mathcal{D}(S) \}$ is the same as $\mathcal{D}(S)$. For example, the specification in Figure 3.1 precisely specifies the data type **Set-Int** defined in Chapter 2, whereas the specification in Figure 3.2 does not precisely specify the data type **Stk-Int-100** defined in Chapter 2.

We can also show that a specification S is *correct* w.r.t. a model **A** by showing that $\mathbf{A} \in \mathcal{F}(S)$.

We can define equivalence among specifications as follows:

Def. 3.8 Two specifications S_1 and S_2 are *equivalent*, expressed as $S_1 = S_2$, iff $\mathcal{D}(S_1) = \mathcal{D}(S_2)$ (i.e., $\mathcal{F}(S_1) = \mathcal{F}(S_2)$). ∎

Note that we do not make any distinction between a specification in which the constructors are 'completely' specified and another specification in which some of the properties of the constructors are not specified. For example, the specification of **Set-Int** does not specify the property of **Insert** that the order in which integers are inserted does not matter. The specification in Figure 3.1 is equivalent to the new specification obtained by adding the following axiom because both have the same semantics:

Insert(Insert(s, i1), i2) ≡ if i1 ≠ i2 then Insert(s, i1) else Insert(Insert(s, i2), i1).

However, as we discuss in Chapter 4, it is possible to prove more properties about **Set-Int** using the specification with the above axiom than the specification given in Figure 3.1. We distinguish between the two specifications there, and define a stronger equivalence relation on specifications which incorporates this distinction.

We have discussed above one way of precisely specifying a data type D. As stated in the beginning of this chapter, D can be presented in many ways.[8] One way is to present

---

8. We have deliberately used the word 'presented' instead of 'specified' to avoid confusion, as we have precisely characterized above when a data type can be specified.

a representative model **A** and define the semantics of such a presentation to be { **A'** | **A'** is behaviorally equivalent to **A** }, as in [3]. There could be other ways of presenting data types. If the semantics of these methods can be given in terms of type algebras using our formalism, we can relate specifications given using different methods (see discussion in Section 3.6).

3.4 Specification of Bool

In Chapter 2, we defined the data type **Bool** which serves as the basis of our formalism. Figure 3.4 contains a specification of **Bool**; this specification cannot be expressed in the proposed specification language because it has an inequality

$$T \not\equiv F$$

as an axiom. This axiom is introduced to capture the property that the boolean constants T and F are distinguishable from each other. The semantics of the specification is the data type **Bool**; it can be verified that every axiom in the specification holds in a model of **Bool**. Because of the inequality, we do not need to introduce inequalities in the specifications of other data types; we will show in the next chapter (Subsection 4.2.3) how to deduce them using the above inequality. The specification of **Bool** is assumed to be given.

Figure 3.4. Specification of Bool

Operations

T	: → Bool	
F	: → Bool	
not	: Bool → Bool	*as* $\sim x$
or	: Bool X Bool → Bool	*as* $x_1 \vee x_2$
and	: Bool X Bool → Bool	*as* $x_1 \wedge x_2$
implies	: Bool X Bool → Bool	*as* $x_1 \Rightarrow x_2$
eqv	: Bool X Bool → Bool	*as* $x_1 \Leftrightarrow x_2$

Axioms

$T \not\equiv F$
$\sim T \equiv F$
$\sim F \equiv T$
$x \vee y \equiv y \vee x$
$x \vee T \equiv T$
$F \vee F \equiv F$
$x \wedge y \equiv \sim((\sim x) \vee (\sim y))$
$(x \Rightarrow y) \equiv (\sim x) \vee y$
$x \Leftrightarrow y \equiv (x \Rightarrow y) \wedge (y \Rightarrow x)$

3.5 Properties of a Specification

We discuss two properties of a specification, namely *consistency* and *behavioral completeness*, based on its semantics. These properties are different from the consistency and sufficient completeness properties defined by Guttag and Horning [28], which are proof theoretic (i.e., based on what can be deduced from a specification). We discuss the relationships between the properties introduced in this section and the properties defined by Guttag and Horning in the next chapter.

Consistency and behavioral completeness are both structural properties; they ensure proper relationships among different components of a specification. Generally speaking, consistency means that a property assumed already is not invalidated. In this case, it means that properties expressed in the specification of a defining type or an auxiliary type, or the assumptions made about the way the exceptional behavior of the operations be specified, are not invalidated. It ensures that a specification specifies at least one data type.

Behavioral completeness captures the intuition that a specification completely specifies the observable behavior of the operations on the intended inputs (i.e., inputs satisfying the associated preconditions). A designer of a specification intentionally leaves the operation behavior unspecified by associating preconditions and optional exception conditions with the operations. Apart from intentional incompleteness, a specification may be incomplete because the designer unintentionally omitted some axioms. The behavioral completeness property ensures that a specification is only intentionally incomplete. So, it warns against any omission. It is a desirable property for most of the specifications.

We first discuss the consistency property; later, we discuss the behavioral completeness property.

3.5.1 Consistency

A specification S is, informally speaking, inconsistent

(i) if S specifies ground terms of a defining type (or an auxiliary type) that are specified to be distinguishable by its specification, to be observably equivalent, or

(ii) if S specifies ground terms of a defining type (or an auxiliary type) that are specified to be observably equivalent by its specification, to be distinguishable.

An example of the first case would be a specification S using the specification of **Bool** and specifying T and F to be observably equivalent. An example of the second case is the specification of **EX1** given in Figure 3.5. The data type **EX1** has only one value. The predicate P distinguishes among observably equivalent ground terms of **Set-Int** : P returns T if and only if in its set argument, an integer has been inserted more than once; otherwise, it returns F. This property of the set values is not observable by the operations of **Set-Int** as specified in Figure 3.1.

In either case, S does not have any models, i.e., $F(S) = \emptyset$. In the first case, no type algebra can satisfy S because one of the axioms would want two distinguishable values in the domain of D' to be observably equivalent. In the second case, S does not have any models because of the well formedness property of a type algebra (which is that the set of observable equivalence relations is a congruence).

EX1 cannot be implemented in any programming language in which an implementation of a data type is hierarchically structured and the representation of a data type is hidden from the users of the data type, since only the external behavior of **Set-Int** can be observed. Thus the predicate P cannot be implemented because the implementation of P must distinguish between, for example, the observably equivalent ground terms **Insert(Insert(∅, 0), 0)** and **Insert(∅, 0)**. Polajnar [67] has also discussed such a violation by a specification S of the specifications of the defining types. He said such a

---

Figure 3.5. Specification of EX1

Operations

```
a      : → EX1
P      : EX1 X Set-Int → Bool
```

Axioms

```
P(a, ∅) ≡ F
P(a, Insert(s, i)) ≡ if i ∈ s then T else P(a, s)
```

specification had *protection errors.*

A specification can also be inconsistent because the exceptional behavior of the operations is not properly specified, for example, the boolean conditions in exception condition restrictions may not be disjoint.

Def. 3.9 A specification S is *consistent* if and only if (i) the specification S' of D', for each $D' \in \Delta \cup A_t$, is consistent, and (ii) $\mathcal{D}(S)$ is not the empty set. ∎

A specification S defines observable equivalence relations on ground terms just like a data type does. By a term here, we mean a term constructed without using auxiliary functions.

Def. 3.10 S *specifies* two ground terms e_1 and e_2 of type $D' \in \Delta'$ to be *observably equivalent* (or e_1 and e_2 are *observably equivalent by* S) iff e_1 and e_2 are observably equivalent in every data type in $\mathcal{D}(S)$ (i.e., the possible interpretations of e_1 in a model $A \in \mathcal{F}(S)$ are observably equivalent to the possible interpretations of e_2 in A). ∎

Def. 3.11 S *specifies* e_1 and e_2 to be *distinguishable* iff e_1 and e_2 are distinguishable in every data type in $\mathcal{D}(S)$ (i.e., the possible interpretations of e_1 in a model A in $\mathcal{F}(S)$ are distinguishable from the possible interpretations of e_2 in A). ∎

For example, **Insert(Insert(ø, 1), 1)** and **Insert(ø, 1)** are specified by the specification of **Set-Int** to be observably equivalent. **Insert(ø, 1)** and **Insert(ø, 2)** are distinguishable. However the specification in Figure 3.2 does not specify **Pop(Null)** and **Null** to be observably equivalent or distinguishable. If S is inconsistent, there are ground terms which are both observably equivalent as well as distinguishable by S, because $\mathcal{F}(S)$ is the empty set.

Since a specification S may leave the behavior of operations unspecified on certain inputs using the precondition and/or optional exception restrictions, there may in general exist ground terms of type $D' \in \Delta'$ which are neither specified by S to be observably equivalent nor distinguishable. For example, **Pop(Null)** is neither observably equivalent to **Null** nor distinguishable from **Null** by the specification of **Stk-Int** in Figure 3.2, as a data type in $\mathcal{D}$(**Stk-Int**) may have **Pop** return the empty stack itself when invoked on the empty

stack and another data type in $D(S)$ may have **Pop** signal on the empty stack. Ground terms involving nondeterministic operations may also be neither observably equivalent nor distinguishable by S; for example, the ground term **Choose(Insert(Insert(Null, 1), 3))** is neither observably equivalent nor distinguishable from 3. The above observable equivalence and distinguishability relations capture the common behavior of data types in $D(S)$.

3.5.2 Behavioral Completeness

In the definition of behavioral completeness, we must capture the intentional incompleteness of a specification. If a specification S associates a nontrivial precondition with an operation, different data types in $D(S)$ can have such an operation behaving differently on an input not satisfying the precondition. If an operation is specified to have an option to signal when its input satisfies a condition, different data types in $D(S)$ can have such an operation signalling the specified exception or terminating normally on an input satisfying the associated condition. If S specifies a nondeterministic operation, different data types in $D(S)$ can have such an operation having as much nondeterminism as desired. This incompleteness in S is intentional. Any other difference in the behavior of data types in $D(S)$ is unintentional.

The above means that for a specification S to be behaviorally complete, data types in $D(S)$ having maximal amount of nondeterminism allowed by S must have the same observable behavior on intended inputs, except that if there is an optional exception condition specified for an operation, then the operation has the option of signalling or terminating normally on an input satisfying the boolean condition in the optional exception condition.

We define three relations on the models in $F(S)$. The *partial isomorphic equivalence* relation formalizes the intentional incompleteness introduced due to the nontrivial preconditions specified for the operations in S. The *isomorphic embeddability* relation formalizes the intentional incompleteness due to the operations specified to have the option to signal exceptions. Later we combine them to define the *partial isomorphic embeddability* on reduced models in $F(S)$. We use the partial isomorphic embeddability

relation to define the behavioral completeness of a specification by relating the reduced models of data types in $\mathcal{D}(S)$ having the maximum amount of nondeterminism allowed by the specification S.

3.5.2.1 Partial Isomorphic Equivalence

Let P_σ be a precondition specified for σ in S. Let S' be the specification of a defining type $D' \in \Delta$ in S. The partial isomorphic equivalence relation relates models whose operations have the same behavior on inputs satisfying their preconditions. The definition is obtained by modifying the definition of isomorphic equivalence (Def. 2.13) given in Chapter 2. As in Chapter 2, we assume that the domains corresponding to each $D' \in \Delta$ in models A_1 and A_2 are defined by the isomorphically equivalent models in $\mathcal{F}(S')$ and that the isomorphic equivalence relation on these models in $\mathcal{F}(S')$ induces a bijection $\Phi_{D'} : V^1_{D'} \rightarrow V^2_{D'}$.

Def. 3.12 Given two algebras A_1 and A_2 in $\mathcal{F}(S)$

$$A_1 = [\{ V^1_{D'} \mid D' \in \Delta' \}, EXV_1 ; \{ f^1_\sigma \mid \sigma \in \Omega \}]$$
$$A_2 = [\{ V^2_{D'} \mid D' \in \Delta' \}, EXV_2 ; \{ f^2_\sigma \mid \sigma \in \Omega \}]$$

such that for each $D' \in \Delta$, $V^1_{D'}$ and $V^2_{D'}$ are the value sets defined by isomorphically equivalent models A'_1 and A'_2 in $\mathcal{F}(S')$, where S' is a specification of D', and $\Phi_{D'} : V^1_{D'} \rightarrow V^2_{D'}$ is a bijection induced due to the isomorphic equivalence of A'_1 and A'_2, A_1 and A_2 are *isomorphically equivalent w.r.t.* $\{ P_\sigma \mid \sigma \in \Omega \}$ (or w.r.t. S) iff there are bijections $\Phi_D : V^1_D \rightarrow V^2_D$ and $\Phi_{EXV} : EXV_1 \rightarrow EXV_2$ such that $\Phi = \{ \Phi_{D'} \mid D' \in \Delta' \} \cup \{ \Phi_{EXV} \}$ has the following properties:

(i) For each $ex : D_1 \times \ldots \times D_n$, and for every v_1 of type D_1, ..., v_n of type D_n,
$\Phi_{EXV}(ex(v_1, \ldots, v_n)) = ex(\Phi_{D_1}(v_1), \ldots, \Phi_{D_n}(v_n))$, and

(ii) for each $\sigma \in \Omega$, $\sigma : D_1 \times \ldots \times D_n \rightarrow D'$,
for every v_1 of type $D_1, \ldots, v_n$ of type D_n, if $P_\sigma(v_1, \ldots, v_n) = T$, then

(a) if neither f^1_σ nor f^2_σ signals, then
$\{ \Phi_{D'}(f^1_\sigma(v_1, \ldots, v_n)) \} = \{ f^2_\sigma(\Phi_{D_1}(v_1), \ldots, \Phi_{D_n}(v_n)) \}$; otherwise,

(b) $\Phi_{EXV}(f^1_\sigma(v_1, \ldots, v_n)) = f^2_\sigma(\Phi_{D_1}(v_1), \ldots, \Phi_{D_n}(v_n))$. ∎

We also call A_1 and A_2 *partially* isomorphically equivalent, when $\{ P_\sigma \mid \sigma \in \Omega \}$ is evident from the context.

The reason for requiring Φ_D to be a bijection (and not a partial one to one function) is the assumption that for the case when a constructor is specified to have a nontrivial precondition, if it terminates normally on an input not satisfying its precondition, the value returned can be constructed by the constructors using inputs satisfying their preconditions.

3.5.2.2 Isomorphic Embeddability

In the definition of isomorphic embeddability relation, we want to capture the intuition that if a specification S associates an optional exception condition with an operation σ, then on an input X satisfying the associated boolean condition $O(X)$, the function corresponding to σ either behaves the same in different algebras in $F(S)$ (i.e., it either returns the 'same' value or signals the 'same' exception value), or the function behavior differs in different algebras to the extent that in one algebra, the function signals the desired exception value and in the other, the function returns the desired normal value. The condition (iii) in the definition below captures this.

If any constructor σ is specified to optionally signal, then the value set of D defined by one algebra in $F(S)$ may be a subset of the value set of D defined by another algebra in $F(S)$. (In fact, one value set may have a value that is distinguishable from every value in the other value set.) That is why in the definition below, we do not require the mapping relating the value sets of D in two algebras to be a bijection; instead, it is required to be a one to one partial function.[9] However, the mapping must be defined for every value constructed by the function corresponding to a constructor σ using inputs which satisfy the associated precondition and do not satisfy any boolean condition stated in a required exception condition or an optional exception condition specified for σ. This constraint is captured in the condition (i) below.

---

9. That is also the reason for calling the relation isomorphically embeddable.

Def. 3.13 Given two algebras $\mathbf{A}_1$ and $\mathbf{A}_2$ in $F(S)$ satisfying the requirement about the domain corresponding to $D' \in \Delta$ stated in Def. 3.12, $\mathbf{A}_2$ is *isomorphically embeddable* in $\mathbf{A}_1$ w.r.t. S iff there exist 1-1 partial functions $\Phi_D : V_D^1 \rightarrow V_D^2$ and $\Phi_{EXV} : EXV_1 \rightarrow EXV_2$, with the following properties:

(i) for every set of values $v_1, \ldots, v_n$, for a constructor σ, if

(a) $P_\sigma[x_1/v_1, \ldots, x_n/v_n]$ holds,

(b) for every required exception condition specified for σ, its boolean condition $R_i[x_1/v_1, \ldots, x_n/v_n]$ does not hold, and

(c) for every optional exception condition specified for σ, its boolean condition $O_j[x_1/v_1, \ldots, x_n/v_n]$ does not hold,

then Φ_D is defined for every value $f_\sigma^1(v_1, \ldots, v_n)$,

(ii) for every exception name $ex : D_1' \times \ldots \times D_m'$,

$\Phi_{EXV}(ex(v_1', \ldots, v_m')) = ex(\Phi_{D_1'}(v_1'), \ldots, \Phi_{D_m'}(v_m'))$ if $\Phi_{D_i'}(v_i')$ is defined for each $1 \leq i \leq m$, and

(iii) for each $\sigma \in \Omega$, for every set of values $v_1, \ldots, v_n$ such that $\Phi_{D_i}(v_i)$ is defined for each $1 \leq i \leq n$,

(a) if on $v_1, \ldots, v_n$, f_σ^1 signals an exception value $ex(v_1', \ldots, v_m')$ specified to be optional by S, then the associated condition $O_j(x_1, \ldots, x_n)$ holds on $v_1, \ldots, v_n$, and $f_\sigma^2(\Phi_{D_1}(v_1), \ldots, \Phi_{D_n}(v_n))$ either signals $ex(\Phi_{D_1'}(v_1'), \ldots, \Phi_{D_m'}(v_m'))$ or returns $\Phi_D(v')$ for some v', or

(b) if $\Phi_{D_1'}(v_1'), \ldots, \Phi_{D_m'}(v_m')$ are defined and f_σ^2 signals an exception value $ex(\Phi_{D_1'}(v_1'), \ldots, \Phi_{D_m'}(v_m'))$ specified to be optional by S on input $\Phi_{D_1}(v_1), \ldots, \Phi_{D_n}(v_n)$, then the associated condition $O_j(x_1, \ldots, x_n)$ holds on $\Phi_{D_1}(v_1), \ldots, \Phi_{D_n}(v_n)$, and $f_\sigma^1(v_1, \ldots, v_n)$ either signals $ex(v_1', \ldots, v_m')$ or returns v'; otherwise,

(c) $\{ f_\sigma^1(v_1, \ldots, v_n) \} = \{ f_\sigma^2(\Phi_{D_1}(v_1), \ldots, \Phi_{D_n}(v_n)) \}$. ∎

For example, let us modify the model $\mathbf{A}_{stk}$ discussed in Subsection 2.3.2 so that the function corresponding to **Push** signals **overflow** if sequence size is 128, instead of 100, and call the modified model $\mathbf{A}'_{stk}$. It can be shown that $\mathbf{A}_{stk}$ is isomorphically embeddable in $\mathbf{A}'_{stk}$. $\mathbf{A}'_{stk}$ is 'bigger' than $\mathbf{A}_{stk}$ because the value set corresponding to

Stk has more elements in A'_{stk} than in A_{stk}. When optional exception conditions for constructors are specified to state a least upper bound on the size of the values of the data type, as in case of the specification of Stk·Int in Figure 3.2, different algebras in $F(S)$ may have different upper bounds on the size of the values in their value sets.

3.5.2.3 Partial Isomorphic Embeddability

We combine the notions of partial isomorphic equivalence and isomorphic embeddability to define another relation. The new relation captures both kinds of intentional incompleteness, due to preconditions as well as due to optional exception conditions.

Def. 3.14 A_1 is *partially isomorphically embeddable* w.r.t. S in A_2 if and only if there exists a model A' in $F(S)$ such that A' is partially isomorphically equivalent to A_1 and A' is isomorphically embeddable in A_2. ∎

3.5.2.4 Definition of Behavioral Completeness

We define behavioral completeness of a specification by relating the reduced models of the data types having maximal amount of nondeterminism allowed by S in $D(S)$ using the partial isomorphic embeddability relation. The definition of behavioral completeness is a single level definition in the sense that a specification S can be behaviorally complete irrespective of whether a specification of a defining type in S is behaviorally complete. If the specification of a defining type is behaviorally incomplete, the incompleteness will be reflected in the semantics of a behaviorally complete S. So, in the definition, we consider only reduced models in $F(S)$ that have the domains corresponding to each $D' \in \Delta$ defined by the isomorphically equivalent models in $F(S')$, where S′ is a specification of D′.

Def. 3.15 A specification S is *behaviorally complete* iff (i) S is inconsistent, or (ii) for any two reduced models $\mathbf{A}_1$ and $\mathbf{A}_2$ in $\mathcal{F}(S)$ having maximum amount of nondeterminism allowed by S and whose domains corresponding to each $D' \in \Delta$ are defined by the isomorphically equivalent models in $\mathcal{F}(S')$, where S' is a specification of D', $\mathbf{A}_1$ is partially isomorphically embeddable in $\mathbf{A}_2$ or vice versa. ∎

The reasons for having the first case this way in the above definition are that for an inconsistent S, $\mathcal{F}(S) = \emptyset$, so any relation among algebras in $\mathcal{F}(S)$ holds, and that we want our definitions to be compatible with the definitions of consistency and completeness in logic, in which an inconsistent theory is complete.

For examples, the specifications of **Set-Int**, **Stk-Int**, and **Bool** in Figures 3.1, 3.2, and 3.4 respectively can be shown to be behaviorally complete. Note that any specification not specifying any observers is trivially behaviorally complete. We can show the following:

Thm. 3.3 For a specification S specifying only deterministic operations and not specifying any precondition or an optional exception condition for an operation, a consistent S is behaviorally complete iff S precisely specifies a data type D assuming that the specification S' of every $D' \in \Delta$ precisely specifies D'.

Proof The above definition of behavioral completeness reduces under the stated conditions to requiring that the reduced models in $\mathcal{F}(S)$ are isomorphically equivalent.[10] This means that $\mathcal{F}(S) = \mathbf{M}_D$.

Hence the theorem. ∎

The behavioral completeness property guarantees that the behavior of the operations has not been left unintentionally unspecified. However, there are situations when the behavioral completeness requirement on specifications is restrictive [31, 51]. For example, consider a modified version of the specification of **Set-Int** in Figure 3.1 in which **Choose** is not specified to nondeterministic. In such a specification also, we do not wish to

---

10. If a specification does not specify a nontrivial precondition for an operation and also does not specify any optional exception condition, the partial isomorphic embeddability relation reduces to isomorphic equivalence.

commit to the value **Choose** may return on an nonempty set, so the axiom specifying **Choose** is still

Choose(s) $\in$ **s** $\equiv$ **T.**

This specification is not behaviorally complete. We would want such a specification to be behaviorally incomplete, as otherwise **Choose** must be completely specified. The behavioral completeness requirement is restrictive in such a case because the reduced algebras in the semantics of the modified specification are not isomorphically equivalent. For example, in one reduced algebra, the function corresponding to **Choose** when applied on { 1, 3 } may return 1, while in another reduced algebra, the corresponding function may return 3. For most specifications specifying nondeterministic operations, if we modify such a specification so that an operation specified originally to be nondeterministic is instead specified to be deterministic, then we would often want the modified specification to be behaviorally incomplete.

3.6 Comparison With Related Works

We compare our specification language with those of Guttag et al. [29] with extensions proposed in [31], Zilles [77], the ADJ group [22, 23], Burstall and Goguen [7], Goguen and Tardo [21], and Nakajima et al. [62]. We first discuss the capabilities of these specification languages and the approach used to give their semantics. Later, we compare the semantics of a specification in these languages.

Zilles [77] and ADJ [23] do not allow auxiliary functions in a specification, so their languages have a limited expressive power. Zilles [77] assumes that the operations of a data type are deterministic and that they do not signal exceptions. The ADJ [23] do not allow nondeterministic operations either; they adopt the simpler approach discussed in Subsection 2.3.3 for modeling exceptions, and discuss a specification language embodying this approach. Goguen [20] extended the ADJ method of modeling exceptions, which we compared with our approach in Subsection 2.3.2. His approach for specifying exceptional behavior of the operations is different from our approach; it is motivated by the view that exception values are like normal values (and so they are typed). The exceptional behavior of the operations is specified using equations. Our language is richer than his language because of the preconditions and the distinction made between optional exception conditions and required exception conditions. His semantics of the specification method is complex.

Burstall and Goguen's [7] CLEAR language and its extension, the OBJ language, support hierarchical structure and modularity like our language. However, Burstall and Goguen have ambitious goals; they are attempting to develop a general purpose specification language based on algebraic semantics in which the semantics of a programming language can be specified. So they are forced to introduce complex mechanisms, for instance, procedures operating on theories, which make the specification language hard to understand. The category-theoretic semantics of their language is also complex [30]. Our approach instead has been to concentrate on the data component of programs, and develop a specification language and a formalism for data types. Our semantic method is simpler.

Guttag et al.'s work [29] is the closest to our work. Their language is limited as it

cannot specify data types with nondeterministic operations. As was said in Section 3.1, our specification language is an enrichment of the specification language in [31]. Our formalism can provide a semantics for their specification language. Our formalism can also be used to provide a mathematical basis of the AFFIRM system [60, 61]. In this sense, our formalism places their work on a firm basis.

Nakajima et al. [62] specify a data type, as discussed in Chapter 1, as a first order theory. Their method differs from other methods including our method because they allow any first order formula to be an axiom in a specification. Auxiliary functions are not allowed in a specification. Operations are assumed to be deterministic; they do not signal exceptions. We have not yet seen the semantics of their specification language. If we assume that a first order theory is interpreted in a standard way as in Logic [16], the problems with this approach are discussed in the related work section of the first chapter. We further comment on their specification method in the next chapter from the point of view of deducing properties from a specification.

Burstall and Goguen, Nakajima et al., and Guttag [31] can specify a *type scheme* (also called a *parameterized type*) in their languages. Recently, the ADJ group [71] has given a category theoretic semantics of a parameterized type. Our specification language, as it is, cannot express a parameterized type. However it should be evident from the discussion that our formalism as well as specification language can be easily extended to parameterized types. We discuss these extensions in the last chapter of the thesis.

There are differences between our semantics of a specification, and those of Zilles, the ADJ group, and Guttag et al. [28], which are motivated by different definitions of a data type used in various formalisms. Zilles and the ADJ assume that values not specified to be related by the axioms are different, even if they are observably equivalent. Guttag et al. on the contrary assume that the values are equivalent unless specified to be different. We have taken a different approach; we consider the axioms as specifying the observably equivalence relation. Our approach towards the semantics of a specification is similar to the one adopted in logic; we consider all models of the axioms to be the semantics of the specification. (Of course, we consider only the algebras satisfying the minimality property for modeling data types, and rule out nonstandard models.) Our

semantics thus subsumes Zilles's and the ADJ's definitions, as well as Guttag et al.'s definition in the following way.

To understand the semantics of a specification in the ADJ group formalism as well as in Zilles's formalism, we introduce the following definition. As is stated in Subsection 2.2.6, the models in $\mathcal{F}(S)$ can be partially ordered using the onto homomorphism relation, i.e., $\mathbf{A}_1 \leq \mathbf{A}_2$ if and only if $\mathbf{A}_1$ is a homomorphic image of $\mathbf{A}_2$.

Def. 3.16 A model $\mathbf{A}$ in $\mathcal{F}(S)$ is called *initial* if $\mathbf{A}$ is a maximal model with respect to the homomorphism relation, and $\mathbf{A}$ identically satisfies S. ∎

In an initial model $\mathbf{A}$, $V_{D'}$ for each $D' \in \Delta$ is a value set defined by an initial model in $\mathcal{F}(S')$, where S' is a specification of D'. Two members in V_D are not the same unless they are related by the axioms and restrictions. The ADJ group and Zilles define the semantics of a specification S to be the set of initial models in $\mathcal{F}(S)$. Guttag et al., on the other hand, define the semantics of a specification S to be the set of reduced models in $\mathcal{F}(S)$.

4. Deductive System

In this chapter, we develop a deductive system for abstract data types. The deductive system embodies general properties of data types which are not explicitly stated in a specification but assumed in the semantics of the specification language. We construct a theory of a data type, which is a collection of properties of the data type, from its specification. The theory of a data type can be used in reasoning about programs and designs that use the data type in the same way as the properties of natural numbers are used in reasoning about programs operating on natural numbers. In particular, the correctness proof of an implementation of a data type with respect to its specification as discussed in the next chapter, involves the use of the theories of its defining types and the theory of its rep, the data type whose values are used to represent the values of D in the implementation. We can pose questions about the behavior of a data type and check whether they can be answered from its specification according to our intentions using the deductive system. In this sense, constructing the theory of a data type can enhance our confidence in its specification.

The construction of the theory of a data type from its specification has an important advantage that the theory does not depend on any particular implementation of the data type. The correctness criterion used for implementations in Chapter 5 guarantees that every property in the theory is satisfied by every correct implementation. We can thus reason about programs using a data type abstractly without referring to any particular implementation of the data type. This separation between the theory of a data type and its implementations via the specification factors the proof process in to two independent parts: (i) Proof of use of a data type, and (ii) proof of correctness of implementation of a data type [37]. In this chapter, we discuss the first part; we discuss the second part in the next chapter.

The theory of a data type is constructed hierarchically from its specification, using the theories of the types used in the specification, just like the specification of a data type is designed. The design of our specification language has been influenced by the goal that a specification should not have to state more than what is required and that it be structured

in the sense that different components of the data type behavior are separately specified. To construct the theory of a data type from its specification, we combine these components. For instance, as is discussed in the previous chapter, an axiom in the axioms component has a restricted interpretation: A variable of type D' in the axiom cannot be freely substituted; instead, the substitution should be such that the input to every operation symbol satisfies its precondition as specified by the restrictions component, and no operation invocation should signal. We first construct the unrestricted axioms from the restricted axioms in the axioms component of a specification using the restrictions; these unrestricted axioms are used to construct the theory. Henceforth, we refer to a (restricted) axiom in the axioms component of a specification as a formula and to an unrestricted axiom as an axiom to avoid confusion.

The proposed deductive system is used to prove properties manually. We have not investigated the possibilities of automating the deductive system, but we relate our work to Musser's work [60, 61] on automating the proof theory of data types from their algebraic specifications.

Instead of discussing the complete deductive system and the construction of a theory from a specification specifying nondeterministic operations and operations exhibiting exceptional behavior in a single shot, we do so step by step. We first discuss the theory of a data type with deterministic operations and without considering their exceptional behavior. We then incorporate the exceptional behavior of data types into their theory. Finally, we discuss data types with nondeterministic operations to exhibit the extra machinery needed for introducing nondeterminism.

For specifications specifying only deterministic operations, we discuss various subtheories, namely, the *equational* subtheory, *distinguishability* subtheory, *inductive* subtheory, constructed using different fragments of the deductive system. We define three structural properties of a specification, namely, *sufficient completeness*, *well definedness*, and *completeness*. Checking for these properties for a specification is a step towards ensuring the correctness of the specification. 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 discussed in the previous chapter, 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. We show that sufficient completeness is stronger than behavioral completeness.

The completeness property is even 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.

The well definedness property constrains that a specification be modular in the sense that it preserve the specifications of defining types and auxiliary types in it. This property is stronger than the consistency property.

In the last section, we define a stronger equivalence on specifications than the equivalence defined in Section 3.3. The stronger equivalence of specifications requires that not only the two specifications have the same semantics, but their theories must also be the same.

4.1 Preliminaries

A data type can have many different but equivalent specifications (see Section 3.3 and Section 4.5). These specifications may differ because

(i) they may specify the properties of constructors to different extents,

(ii) the properties of the operations are specified in different ways, and

(iii) they may use different sets of auxiliary functions.

Theories constructed from different equivalent specifications can be different, as will be clear from the following discussion. Unless stated otherwise, we assume that a data type has a single fixed specification; in the last section of the chapter, we discuss theories constructed from different but equivalent specifications of a data type.

If a specification S specifies only a single data type **D**, then the theory constructed from S is the theory of **D**. If S specifies a set of related data types, then the theory constructed from S is the theory of the set of related data types. The theory constructed from S consists of properties characterizing the behavior of the algebras in $\mathcal{F}$(S), the semantics of S. Let **Th**(S) stand for the theory constructed from S.

The deductive system uses multi-sorted (or many sorted) first order predicate calculus with identity [16] as the underlying logic. Though a first order theory cannot completely characterize the 'infinite' models in $\mathcal{F}$(S), we prefer first order logic over second order logic because of the following reasons:

(i) First order logic is well studied, and is better understood than second order logic,

(ii) most of the programming logics developed for reasoning about the control structures of programming languages are first order,

(iii) the recent work of Cartwright and McCarthy [8] has established that even the termination proof, which was believed to employ second order reasoning, can be adequately done in first order logic,

(iv) most of the work in automatic verification uses first order logic as the underlying basis, and

(v) we believe that the most of the interesting properties of programs can be expressed in first order logic.

Multi-sorted logic is more convenient than single-sorted logic as it avoids the use of type

predicates, which must be introduced in a single-sorted logic to differentiate among terms of different types. We use an induction rule having infinitely many premises which is some what unusual; the proofs using this rule are infinitary. We interpret the formulas in Th(S) in the algebras in $\mathcal{F}$(S); we do not consider uncountable structures because they are not type algebras and so they are of no interest.

As was discussed in the previous chapter, a formula is interpreted in a type algebra in the same way as a formula in a structure in Logic [16], except that the symbol $\equiv$ is interpreted as the observable equivalence relation (see the definition in Sections 2.2 and 2.3) on a domain instead of the identity relation. Because the observable equivalence relation is an equivalence relation and is preserved by every function in a type algebra, the standard rules for identity hold (i.e., the rules for identity are sound under this interpretation).

We now discuss the structure of formulas expressing properties of the models in $\mathcal{F}$(S). Following Enderton [16], we define the *language* of Th(S) as the set of nonlogical symbols; the nonlogical symbols are used with the logical symbols to construct formulas. [1] Let L(S) stand for the language of Th(S). Instead of defining the complete language of Th(S) here, we introduce it incrementally. We discuss here L(S) for a specification neither specifying nondeterministic operations nor the exceptional behavior of the operations. L(S) includes the operation symbols of D specified by S as well as the auxiliary function symbols used in S. Since Th(S) is constructed using the theories of the defining types and the theories of the auxiliary types used in S, L(S) includes L(S'), where S' is a specification of a data type D', for each $D' \in \Delta \cup A_t$.

In Section 4.3 on specifications specifying exceptional behavior of the operations, we include the exception names in L(S). In Section 4.4 on specifications specifying nondeterministic operations, L(S) includes additional symbols needed for expressing

---

1. A symbol (or an axiom or a rule of inference) is called *nonlogical* if it is specific to a particular domain of discourse whose theory is being constructed. This is in contrast to logical symbols, which are determined by the underlying logic used to develop the theory. For instance, a logical axiom characterizes the logical reasoning available in the underlying logic, whereas a nonlogical axiom characterizes a property about the domain of discourse.

properties about nondeterministic operations.

Terms of various types can be constructed using the symbols in L(S) and variables of various types as discussed in the previous chapter. An atomic formula is an equation of the form '$e_1 \equiv e_2$', where e_1 and e_2 are terms of the same type. Compound formulas are constructed from atomic formulas using the standard rules of construction for first order predicate calculus with the help of logical symbols.

We consider a boolean term as a term rather than an atomic formula; in this sense, we adopt a uniform view about the symbols in L(S), considering each as a function symbol. This view is especially convenient when we incorporate the exceptional behavior of the operations. In case we use a boolean term b as a formula, b is considered as the abbreviation for the equation '$b \equiv$ T.'

Recall that '$e_1 \equiv$ **if** b **then** e_2' is an abbreviation for '$e_1 \equiv$ **If-then-else**(b, e_2, e_1)' and '$e_1 \equiv$ **if** b **then** e_2 **else** e_3' stands for the following two conditional equations

'$e_1 \equiv$ **if** b **then** e_2,'

'$e_1 \equiv$ **if** $\sim b$ **then** e_3.'

In the simple case when exceptional behavior is not considered, '$e_1 \equiv$ **if** b **then** e_2' is equivalent to '$(b \equiv \text{T}) \Rightarrow (e_1 \equiv e_2)$.' When we incorporate exceptional behavior, the above equivalence does not always hold, because b could possibly signal an exception. However, if b is guaranteed not to signal, then the above equivalence holds in that case also.

We use the abbreviation '$e_1 \not\equiv e_2$' for the formula '$\sim (\forall\, x_1, \ldots, x_n)\, [\, e_1 \equiv e_2\,]$,' where $x_1, \ldots, x_n$ are the only variables in e_1 and e_2. Note that if e_1 and e_2 are ground terms, then '$e_1 \not\equiv e_2$' is equivalent to '$\sim (e_1 \equiv e_2)$.' In fact, it is easy to see that

$(\forall\, x_1, \ldots, x_n)\, [\sim e_1 \equiv e_2\,] \Rightarrow (\, e_1 \not\equiv e_2)$.

Only a subset of Th(S) is useful in reasoning about programs and designs using D. This subset consists of formulas in Th(S) expressed using only the operation symbols. Formulas expressed using auxiliary functions are not directly useful because the auxiliary functions are not available to the users of the data type(s) being specified, but these formulas help in proving formulas without auxiliary functions. The correctness criterion for implementations with respect to a specification S discussed in the next chapter does not require a correct implementation to include implementations of auxiliary functions used in

S. Even if an auxiliary function is implemented, it is not available to the users of a data type.

Let L(D) stand for the language of a data type D, which is a subset of L(S) consisting only of the operation symbols. L(S) - L(D) is then the set of auxiliary functions used in specifications of various data types. Let Th(D) stand for the subset of Th(S) consisting of formulas in Th(S) expressed using the nonlogical symbols in L(D). We are primarily interested in formulas in Th(D). The correctness criterion used in the next chapter ensures that Th(D) holds for all correct implementations with respect to S. Th(D) serves as the interface between programs using D and the correct implementations of D. Note that Th(D) does not include those nonlogical axioms of Th(S) which are expressed using auxiliary functions.

4.2 Theory of Data Types without Nondeterminism and without Exceptional Behavior

We start with the simple case of specifications that do not specify nondeterministic operations and the exceptional behavior of the operations. The restrictions component of such a specification may specify the nontrivial preconditions for the operations. For illustration, we modify the data type **Set-Int** so that **Choose** is deterministic; let **Set-Int'** stand for the modified **Set-Int**. The specification of **Set-Int'** is given in Figure 4.1, which is obtained by modifying the specification of **Set-Int** given in Figure 3.1. The syntactic specification of the operation **Choose** does not have the identifier *nondeterministic*. Instead of the required exception condition for **Choose** on the empty set, we specify '**~ #(s) = 0**' as its precondition in the restriction component of the specification of **Set-Int'**.

We first discuss how to construct unrestricted nonlogical axioms of **Th**(S) from

---

Figure 4.1. Specification of Set-Int'

Operations

Null	**: → Set-Int'**	*as* **Ø**
Insert	**: Set-Int' X Int → Set-Int'**	
Remove	**: Set-Int' X Int → Set-Int'**	
Has	**: Set-Int' X Int → Bool**	*as* $x_2 \in x_1$
Size	**: Set-Int' → Int**	*as* $\#(x_1)$
Choose	**: Set-Int' → Int**	

Restrictions

Pre(**Choose(s)**) :: **~ (# (s) = 0)**

Axioms

1. **Remove(Ø, i) ≡ Ø**
2. **Remove(Insert(s, i1), i2) ≡ if i1 = i2 then Remove(s, i1) else Insert(Remove(s, i2), i1)**
3. **i ∈ Ø ≡ F**
4. **i1 ∈ Insert(s, i2) ≡ if i1 = i2 then T else i1 ∈ s**
5. **#(Ø) ≡ 0**
6. **#(Insert(s, i)) ≡ if i ∈ s then #(s) else #(s) + 1**
7. **Choose(s) ∈ s ≡ T**

the formulas in the axioms component and the preconditions specified in S. We then discuss how to construct Th(S) from the nonlogical axioms thus obtained. We do so step by step exhibiting the power of various fragments of the deductive system. This will also help in investigating how easily these fragments can be automated. We first discuss a simple but useful subset of Th(S), called the *equational* subtheory and written as EQ(S). Formulas in EQ(S) are proved using the rules of $\equiv$ and the substitution rule of $\forall$. Most of the work on developing the proof theory of data types from their algebraic specifications has focused on this subtheory [23, 71, 7, 21, 29].

We discuss later a richer subtheory, called the *distinguishability* subtheory and written as DS(S), having inequalities '$e_1 \not\equiv e_2$' in addition to equations. The inability to prove an inequality has been a major limitation of the recent works on proof theories based on algebra specifications. For instance, both in Zilles's method as well as in ADJ's method, two terms e_1 and e_2 are unequal, i.e., '$e_1 \not\equiv e_2$' is provable, if and only if '$e_1 \equiv e_2$' is not in the equational subtheory, so the proof of inequality becomes meta. Zilles [76] recognizes this limitation and suggests also using inequalities as axioms. In our deductive system, inequalities can be proved from equations by the method of proof by contradiction. We have this advantage because we view two abstract values (i.e., ground terms) of a data type to be distinguishable (so unequal) if and only if a sequence of operations can distinguish them. This is in contrast to the view taken by the ADJ group and Zilles that two abstract values are distinguishable if and only if they are not specified to be equal.

We later include an induction rule which captures the minimality property of a data type. This rule is 'infinite' and is derived from the syntactic specifications of the operations and the restrictions components of the specification. More properties of a data type can be proved using the induction rule than without it. We discuss how the rule is used to prove other rules using the nonlogical axioms derived from the specification, which simplify the proof of properties of the data type. The subset of equations and inequalities provable using the induction rule and the rules of the distinguishability subtheory is called the *inductive* subtheory and written as IND(S).

We finally construct the full theory Th(S) using the whole machinery of first order predicate calculus and the 'infinite' induction rule. We demonstrate the use of Th(S)

in verifying properties of programs. Every subtheory (as well as the full theory **Th(S)**) is constructed hierarchically from the corresponding subtheory (or the full theory) constructed from the specifications of the defining types and the auxiliary types used in S. For instance, **IND(S)** is constructed from **IND(S')**, where S' is a specification of $D' \in \Delta \cup A_t$.

In the last subsection, we define sufficient completeness, completeness, and well definedness properties of a specification, and relate them to behavioral completeness and consistency properties discussed in Section 3.5.

4.2.1 Derivation of Nonlogical Axioms

The unrestricted nonlogical axioms for a specification S can be derived in a straightforward way. If S specifies a nontrivial precondition for some operations, then the nonlogical axioms are generally conditional equations. Let PC_e stand for a conjunction of conditions of the form '$P_\sigma(e_1, \ldots, e_n) \equiv T$' for every occurrence of σ having the input $e_1, \ldots, e_n$ in e. If an equation '$e_1 \equiv e_2$' is in the axioms component of S, the corresponding nonlogical axiom of Th(S) is the formula

$$(PC_{e_1} \wedge PC_{e_2}) \Rightarrow (e_1 \equiv e_2).$$

For example, the formula

$$\mathbf{Choose(s) \in s \equiv T}$$

has an occurrence of the operation **Choose**, which is specified to have the nontrivial precondition, so the corresponding unrestricted nonlogical axiom is

$$\mathbf{(\sim \#(s) = 0 \equiv T) \Rightarrow (Choose(s) \in s \equiv T).}$$

If a formula in the axioms component does not have any operation specified to have a nontrivial precondition, then the formula itself serves as the nonlogical axiom. For example, the formula

$$\mathbf{\#(Insert(s,i)) \equiv if\ i \in s\ then\ \#(s)\ else\ \#(s) + 1}$$

itself serves as a nonlogical axiom.

For any restricted quantifier-free formula f, the corresponding unrestricted formula is '$PC \Rightarrow f$', where PC is a conjunction of the formulas PC_{e_i} for every term e_i in the formula f.

4.2.2 Equational Subtheory

The equational subtheory EQ(S) consists of equations derived from the nonlogical axioms of S. An equation '$e_1 \equiv e_2$' is in EQ(S) if and only if it is provable from the nonlogical axioms of S and EQ(S'), where S' is a specification of D', for each $D' \in \Delta \cup A_t$, using the four rules of $\equiv$, namely,

(i) reflexivity,

(ii) symmetry,

(iii) transitivity,

(iv) substitution property of every function symbol,

and,

(v) the substitution rule for the universal quantifier $\forall$ (i.e., substituting an appropriate term for every occurrence of a free variable in a nonlogical axiom).

All five of the above rules are not necessary; some of them can be derived from the others [16]. As an illustration, we give a proof of the equation '**#(Insert(Insert(Null, i), i)) ≡ 1.**' in Figure 4.2.

EQ(S) defines a relation on ground terms of different types; let $EQ_{D'}$ stand for this relation on ground terms of type D'. For any ground terms e_1 and e_2, $\langle e_1, e_2 \rangle \in EQ_D$ if and only if '$e_1 \equiv e_2$' $\in$ EQ(S).

If the nonlogical axioms are equations (possibly using if-then-else functions), they can be considered as unidirectional rewrite rules by defining an appropriate ordering on terms. If a decision procedure for EQ(S) exists (i.e., the relation $EQ_{D'}$ for each $D' \in \Delta \cup A_t \cup \{D\}$ is decidable), then it is often possible to generate a convergent set of rewrite rules from the nonlogical axioms using the Knuth-Bendix algorithm [44], which

---

Figure 4.2. Proof of '#(Insert(Insert(Null, i), i)) ≡ 1'

1. i ∈ Insert(Null, i) ≡ T	Substitution in Axiom 4 of Set-Int' and the theorem of Int.
2. #(Insert(Insert(Null, i), i)) ≡ #(Insert(Null, i))	Step 1, substitution in Axiom 6 of Set-Int'
3. ≡ #(Null) + 1	Axiom 3, substitution in axiom 6 of Set-Int', and transitivity.
4. ≡ 0 + 1	Axiom 5 of Set-Int.
5. ≡ 1	Theorem of Int.

constitutes the decision procedure for **EQ(S)**. The AFFIRM system [60] is designed in part around this result. Though nonlogical axioms using **if-then-else** functions have been studied [60, 21, 5], there appears to be some difficulties in using the Knuth-Bendix algorithm on them [61].

For automating the process of proving properties from the nonlogical axioms of **S** using the above five rules, it may be helpful to view a formula of the form

$$\mathbf{PC} \Rightarrow (e_1 \equiv e_2),$$

where **PC** is a conjunction '$b_1 \equiv \mathbf{T} \wedge \ldots \wedge b_n \equiv \mathbf{T}$' as the formula

$$e_1 \equiv \textbf{if } b_1 \wedge \ldots \wedge b_n \textbf{ then } e_2,$$

as the two formulas are equivalent and the second formula can be considered as a rewrite rule. For example,

$$(\sim \#(s) = 0 \equiv \mathbf{T}) \Rightarrow \mathbf{Choose}(s) \in s \equiv \mathbf{T}$$

can be viewed as

$$\mathbf{Choose}(s) \in s \equiv \textbf{if} \sim \#(s) = 0 \textbf{ then } \mathbf{T}.$$

4.2.3 Distinguishability Subtheory

The distinguishability subtheory **DS(S)** is richer than **EQ(S)**; it has two kinds of formulas: (i) '$e_1 \equiv e_2$,' and (ii) '$e_1 \not\equiv e_2$.' Our approach for proving inequalities is simple; it is based on the definition of distinguishability discussed in Sections 2.2 and 2.3. The distinguishability theory of **Bool** serves as the basis; since '$\mathbf{T} \not\equiv \mathbf{F}$' is a formula in the specification of **Bool**, '$\mathbf{T} \not\equiv \mathbf{F}$' $\in$ **DS(Bool)**. (Recall that only the specification of **Bool** includes an inequality as an axiom.) '$\mathbf{T} \not\equiv \mathbf{F}$' obviously holds in every model of **Bool**. This inequality is used to prove inequalities of terms of type **D** by *reductio ad absurdum* (proof by contradiction); this is the sixth logical rule, besides the five rules discussed in the previous subsection, which is used to construct the subtheory **DS(S)**. We of course use inequalities in **DS(S')**, where **S'** is a specification of $\mathbf{D}' \in \Delta \cup \mathbf{A_t}$.

Given two terms e_1 and e_2, we prove '$e_1 \not\equiv e_2$' as follows:

We assume on the contrary that '$e_1 \equiv e_2$;'

we then derive '$e_1' \equiv e_2'$', where '$e_1' \not\equiv e_2'$' is already provable, i.e., either '$e_1' \not\equiv e_2'$' $\in$ **DS(S')**, or '$e_1' \not\equiv e_2'$' $\in$ **DS(S)**.

We illustrate the above rule to prove the inequality '**Null ≢ Insert(s, i)**' in Figure 4.3. For any ground terms e_1 and e_2, the formula '$e_1 \not\equiv e_2$' interprets in a model in F(S) to whether the interpretation of e_1 is distinguishable from the interpretation of e_2.

The method of proof by contradiction can be integrated into a rewrite rules system like AFFIRM. If an inequality '$e_1 \not\equiv e_2$' is to be proved, we assume '$e_1 \equiv e_2$' as an axiom and add it to the set of nonlogical axioms. We get the rewrite rules corresponding to the new set of axioms and run them to check whether a contradiction, i.e., one of the rules 'T→F' and 'F→T' or '$e_1' \rightarrow e_2'$,' is generated, where the inequality '$e_1' \not\equiv e_2'$' is already proved.

4.2.4 Inductive Subtheory

The subtheory DS(S) is still not rich enough because there are many useful equational formulas which hold for every data type in D(S), but cannot be proved using the logical rules of DS(S). For example, the equation

Has(Remove(s, i), i) ≡ F

cannot be proved because

(i) there is no nonlogical axiom directly expressing the behavior of **Has** on a set argument having the structure **Remove(s, i)**, and

(ii) **Remove(s, i)** is not equivalent to **Null** or an expression of the form **Insert(s', i')** unless some conditions are placed on **s**.

But, '**Has(Remove(s, i), i) ≡ F**' holds in every model in F(**Set-Int'**). Even if we use the whole deductive system of first order predicate calculus, this formula cannot be proved from the nonlogical axioms of **Set-Int'**.

Figure 4.3. Proof of Null ≢ Insert(s, i)

To prove Null ≢ Insert (s, i)
assume Null ≡ Insert (s, i)
Has (Null, i) ≡ Has(Insert(s, i), i), — substitution property of Has
F ≡ T, — the axioms 3 and 4 of Set-Int',
which is a contradiction.
so Null ≢ Insert(s, i) ∈ DS(Set-Int').

The above limitation is due to the fact that the minimality property of data types, which is captured in the definition of a type algebra, is neither captured in the underlying logic nor expressed as a nonlogical axiom (see the discussion of the minimality property in Section 2.1). We discuss below an induction rule which captures this property. The rule can be constructed from the syntactic specifications of the operations in **S**. We compare our rule with other similar rules proposed in the literature, and demonstrate the inadequacy of some of these rules. We discuss how the 'infinite' rule can be used in proofs. For better exposition, we first assume that no constructor of **D** is specified to have a nontrivial precondition by **S**; we later relax this restriction.

4.2.4.1 Infinite Induction Rule

Def. 4.1 A ground term e is called a *constructor* ground term if e is expressed only using constructor symbols. ∎

(†) **Induction Rule**

Given a formula $\Phi(x)$ with a free variable x of type **D**.

For every constructor ground term e of type **D**, $\Phi[x/e] \vdash (\forall\ x)\ \Phi(x)$.

The above inference rule is infinitary, as there are usually infinitely many constructor ground terms of type **D** and so, the rule requires infinitely many premises. The notion of a proof is infinitary whenever the induction rule is used. Intuitively, the above rule states that if a formula $\Phi(x)$ holds in every case when a value of type **D** is substituted for x, then we can deduce the formula '$(\forall\ x)\ \Phi(x)$.' It is easy to see that the above rule is sound because every type algebra by definition has the minimality property, which states that every value of **D** is represented by some constructor ground term of type **D**. It is sufficient to consider only constructor ground terms because these represent every value in a type algebra.

Burstall and Goguen [7] also realized the limitation of the proof theory based on

the rules of $\equiv$.[2] They introduced the *induce* operator on theories; the induced theory is equivalent to the original theory with the above induction rule. The above induction rule is a generalization of the structural induction rule of Burstall [6]. The structural induction rule is based on identifying a minimal set of constructors (instead of all constructors) which generates the values of D and has the property that every finite sequence of constructors in the subset generates a distinguishable value. To our knowledge, Wegbreit and Spitzen [72] were the first to generalize the structural induction rule, but they presented it informally. The data induction rule of Guttag et al. [29] is the same as the induction rule of Wegbreit and Spitzen. Recently, Musser [61] has suggested a formalization similar to our formulation of the rule.

4.2.4.2 Rationale for an Infinite Induction Rule

Below, we discuss the rationale for using an infinite rule to capture the minimality property of a data type. We demonstrate the inadequacy of an induction scheme seemingly suggested by Wegbreit and Spitzen [72], Guttag et al. [29], and Nakajima et al. [62]. For illustration, we use a simple version of the data type *natural number*, denoted by N_2. N_2 has four operations: 0, the constant zero; S, the successor operation; P, the predecessor operation; and, =, the equality operation. Its specification is given in Figure 4.4. The constructor P is derived in the sense that the values returned by P can be constructed using 0 and S. We would like to prove from the nonlogical axioms of N_2 and the induction rule, the following normal form lemma in the full theory:

(1) $(\forall x)[x \equiv 0 \vee (\exists y)[x \equiv S(y)]]$.

In general, we would like to have in $Th(N_2)$ the scheme

(2) $(\Phi(0) \wedge (\forall x)[\Phi(x) \Rightarrow \Phi(S(x))]) \Rightarrow (\forall x)\,\Phi(x)$,

where Φ is a first order formula with at least one free variable.

If we express the minimality property of N_2 with the following scheme:

(3) $(\Phi(0) \wedge (\forall x)[\Phi(x) \Rightarrow (\Phi(P(x)) \wedge \Phi(S(x))]) \Rightarrow (\forall x)\,\Phi(x)$,

---

2. However, ADJ [71] do not seem to agree that properties provable using the induction rule are relevant.

Figure 4.4. Specification of Data Type N_2

Operations

$0 : \rightarrow N_2$
$S : N_2 \rightarrow N_2$
$P : N_2 \rightarrow N_2$
$= : N_2 \times N_2 \rightarrow Bool$

Axioms

$P(0) \equiv 0$
$P(S(x)) \equiv x$
$x = x \equiv T$
$x = y \equiv y = x$
$S(x) = 0 \equiv F$
$S(x) = S(y) \equiv x = y$

---

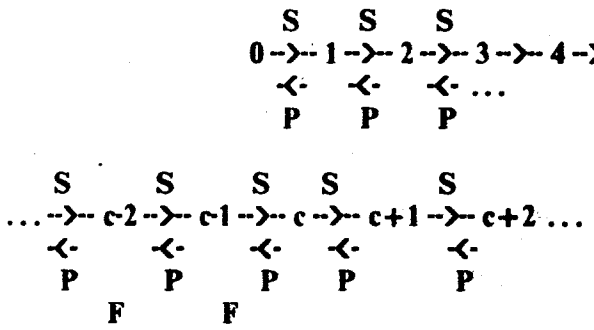
where Φ is a first order formula, we can neither prove (1) nor (2). This is because there are nonstandard models of the nonlogical axioms given in Figure 4.4 and the scheme (3), in which the scheme of formulas (2) and/or the formula (1) do not hold. Figure 4.5 is one such model in which the nonlogical axioms as well as the scheme (3) holds but the formula scheme (2) does not hold. The model has an infinite chain going from a constant symbol c in both directions in addition to the chain of natural numbers, and there is a unary predicate symbol M whose interpretation in the model is the predicate which is false on all constants on the negative side of c, and true otherwise. The figure shows the values in the models on which the interpretation of M is false.

The scheme (3) does not capture the property that the operation P when applied on any natural number will hit in finitely many steps either 0 or a number that behaves like 0 (in nonstandard models). This property is needed to derive (2) or (1).

It should be obvious that the scheme (2) as well as the formula (1) hold in every model in $\mathcal{F}(N_2)$. Formulas of the kind (2) and the formula (1) are very useful in proving properties of programs using N_2. For example, using the formula scheme (2), the proof by induction amounts to checking for the basis condition and a single case in the inductive step, where as (3) requires two cases in the inductive step.

We would like the induction rule to be constructible from the syntactic

Figure 4.5. A Nonstandard Model of the Axioms in N with the Scheme (3)



specification so that the rule does not have to be stated explicitly for every data type in its specification. In addition, the induction rule should be strong enough so that, for example, the formula scheme like (2) and the normal form theorem (1) can be derived in case of N_2. The above discussion shows that the scheme (3) is not powerful enough. However, the infinite induction rule (†) for N_2 does the job. It can be shown that the scheme (2) and the formula (1) are derivable from that rule.

Another alternative for characterizing the minimality property is to use multisorted second order predicate calculus as the underlying logic and express the minimality property as a second order formula. But, this approach is not attractive because of the reasons discussed in the first section.

4.2.4.3 Use of the Induction Rule

For using the induction rule (†), we must establish infinitely many premises. This can be done by imposing a partial ordering on the set of constructor ground terms and using induction on ground terms. We discuss below a technique for doing this. We start with an instantiation of this technique which uses the structure of the ground terms; this method is known as the structural induction [6]. We show that

(i) for each basic constructor $\sigma : D_1 \times \ldots \times D_n \rightarrow D$, which does not take any argument of type D, $\Phi[x/\sigma(e_1, \ldots, e_n)]$ is provable, and

(ii) for every other constructor $\sigma \in \Omega$, $\Phi[x/\sigma(e_1, \ldots, e_n)]$ is provable assuming $\Phi[x/e_i]$ for

every $\mathbf{D}_i = \mathbf{D}$.

However, there are situations when the structural induction is not useful or convenient; instead, a different partial ordering on ground terms is preferable.

We present below a generalized technique. Let **G** stand for the set of all constructor ground terms of type **D**. We can define an ordering relation (non-reflexive, antisymmetric, and transitive) $<$ on **G** such that $(\mathbf{G}, <)$ satisfies the minimum condition. Defining $<$ on **G** gives a generalized (Noetherian) induction rule [10] on **G**.

Def. 4.2 $(\mathbf{G}, <)$ satisfies the *minimum condition* iff for every nonempty subset **A** of **G**, **A** has a minimal element with respect to $<$.[3] ∎

Generalized Induction Rule:

If for every $e \in \mathbf{G}$ such that for every element $e' \in \mathbf{G}$ that is $< e$, $\Phi[x/e'] \Rightarrow \Phi[x/e]$,

then $(\forall\, e \in \mathbf{G})\ \Phi[x/e]$.

So, in order to establish the infinitely many premises of the 'infinite' induction rule (†), we define a partial ordering $<$ on the constructor ground terms in **G** such that $(\mathbf{G}, <)$ has the minimum condition and use the generalized induction rule.

Using the nonlogical axioms of **S**, one can identify a subset **C** of **G** such that for every constructor ground term $e \in \mathbf{G}$, there is a ground term e' in **C** such that '$e \equiv e'$' $\in$ **EQ(S)**. We can then simplify the induction rule using the following rule of first order predicate calculus:

$$(e \equiv e') \vdash \Phi[x/e] \Leftrightarrow \Phi[x/e']$$

We need to show only that for every ground term $e \in \mathbf{C}$, $\Phi[x/e]$. For example, it can be shown in case of **Bool**, that for every boolean ground term e, either '$e \equiv$ T' $\in$ **EQ(Bool)** or '$e \equiv$ F' $\in$ **EQ(Bool)**. So to prove a property having a free variable of type **Bool** by induction, it suffices to show that the property holds in case of **T** and **F**.

Let us consider the example of **Set-Int'**. The induction rule (†) for **Set-Int'** is:

---

3. The property of a set **A** satisfying the minimum condition with respect to an ordering relation $<$ is related to the *well foundedness* property of **A** with respect to $<$. It can be shown that **A** is well founded with respect to $<$ if and only if $(\mathbf{A}, <)$ satisfies the minimum condition.

For every constructor ground term e of type **Set-Int'**, $\Phi[x/e] \vdash (\forall\ x)\ \Phi(x)$.

The following theorem establishes that the constructor **Remove** is derived in the sense that it does not construct any value of **Set-Int'** distinguishable from the values constructed by **Null** and **Insert**.

Thm. 4.1 Every constructor ground term e of type **Set-Int'** is equivalent by equational reasoning to a ground term e' not having any occurrence of **Remove**, i.e., the equation '$e \equiv e'$' $\in$ **EQ(Set-Int')**.

Proof Using induction on the number of **Remove** (and subsequently the number of **Insert**) in a constructor ground term, we show the above with the help of the axioms 1 and 2 of **Set-Int'**. For details, see Appendix III. ∎

Using this theorem, we get a simpler induction rule for **Set-Int'**:

(4) For every constructor ground term e of type **Set-Int'** having only the occurrences of **Null** and **Insert**, $\Phi[x/e] \vdash (\forall\ x)\ \Phi(x)$.

We can define an ordering generated by the following relation on ground terms constructed using **Null** and **Insert**.

$$\textbf{Null} < \textbf{Insert}(x, i), \text{ and } x < \textbf{Insert}(x, i)$$

for any constructor ground term x and integer constructor ground term i. Using the induction rule (4), we can prove for any formula Φ,

(5) $(\ \Phi[x/\textbf{Null}] \wedge (\forall\ x)[\ \Phi(x) \Rightarrow (\forall\ i)\ \Phi(\textbf{Insert}(x, i))\ ]\ ) \Rightarrow (\forall\ x)\ \Phi(x)$.

We also get the following normal form theorem for **Set-Int'** using (5)

$$(\forall\ s)[\ s \equiv \textbf{Null}() \vee (\exists\ s', i')\ \ s \equiv \textbf{Insert}(s', i')\ ].$$

Note that the above formula is different from Theorem 4.1. (The above formula is not in **IND(S)** because of the use of the existential quantifier $\exists$ in it, but it is in **Th(S)** as discussed later.) Theorem 4.1 cannot be expressed in first order predicate calculus. Using the scheme (5) and the nonlogical axioms of **Set-Int'**, we prove '**Has(Remove(s, i), i)** $\equiv$ **F**' in Figure 4.6. Recall that this formula could not be proved in **DS(Set-Int')**.

The inductive subtheory **IND(S)** consists of equations and inequalities, and is defined to be the set of formulas derived from the nonlogical axioms using the six rules discussed in the last subsection (meaning **DS(S)** $\subseteq$ **IND(S)**) and the infinite induction rule

Figure 4.6. Proof of 'Has(Remove(s, i), i) ≡ F'

We use the formula scheme (5) above.
Basis: Has(Remove(Null, i), i) ≡ Has(Null, i) ≡ F — Axioms 1, 3.
Inductive Step Assume Has(Remove(s, i), i) ≡ F,
to show (∀ il)[Has(Remove(Insert(s, il), i), i) ≡ F]

Case 1: i = il
Has(Remove(Insert(s, il), i), i) ≡ Has(Remove(s, i), i) ≡ F, Axiom 2, and the assumption.

Case 2: ~ (i = il)
Has(Remove(Insert(s, il), i), i) ≡ Has(Insert(Remove(s, i), il), i) — Axiom 2.
≡ Has(Remove(s, i), i) ≡ F — Axiom 4 and the assumption.

Using the scheme (5), we get Has(Remove(s, i), i) ≡ F.

(†). We later discuss the conditions under which formulas in **IND(S)** can be proved using the Knuth-Bendix algorithm (Subsection 4.2.7).

4.2.4.4 Specifications with Nontrivial Preconditions for Constructors

The induction rule (†) is also applicable to specifications specifying nontrivial preconditions for the constructors as it captures a general property of data types and not a property of specifications. It can be simplified depending on the semantics used for a constructor σ on inputs not satisfying its precondition.

If nontrivial preconditions are specified for constructors, we are interested in constructor ground terms in which the input to every constructor invocation satisfies the specified precondition. This is so because a constructor is not likely to be invoked with an input not satisfying the specified precondition. Even if the constructor is invoked on such an input, we are not interested in its behavior.

Def. 4.3 A constructor ground term e is called *legal* if and only if (i) e does not have any occurrence of an auxiliary function, and (ii) for every subterm of e of the form $e_1 = \sigma(e_{11}, \ldots, e_{1n})$, where σ is a constructor, '$\mathbf{P}_\sigma(e_{11}, \ldots, e_{1n}) \equiv$ T' $\in$ **EQ(S)**. ∎

The restriction that '$\mathbf{P}_\sigma(e_{11}, \ldots, e_{1n}) \equiv$ T' $\in$ **EQ(S)** is for convenience; we could have required the formula to be in **Th(S)**, the full theory constructed from S. (Recall that $\mathbf{P}_\sigma(X)$

is a boolean term without involving any quantifier.) We are mostly interested in formulas involving legal ground terms.

Assuming the semantics used in Chapter 3 (i.e., on an input not satisfying its precondition, σ returns a value of D constructible by the constructors of D using inputs

---

Figure 4.7. Specification of Stk-Int

Stk-Int *as* **Stk**

Operations

```
Null    : → Stk
Push    : Stk X Int → Stk
                    → overflow(Stk, Int)
Pop     : Stk → Stk
Top     : Stk → Int
                → no-top()
Replace : Stk X Int → Stk
Empty   : Stk → Bool
```

Auxiliary Functions

```
Size    : Stk → Int          as #(x)
```

Restrictions

Pre(Pop(s)) :: ~ Empty(s)
Pre(Replace(s, i)) :: ~ Empty(s)

Empty(s) ⇒ Top(s) *signals* no-top()
Push(s, i) *signals* overflow(s, i) ⇒ #(s) ≥ 100

Axioms

1. Pop(Push(s, i)) ≡ s
2. Top(Push(s, i)) ≡ i
3. Replace(s, i) ≡ Push(Pop(s), i)
4. Empty(Null) ≡ T
5. Empty(Push(s, i)) ≡ F
6. #(Null) ≡ 0
7. #(Push(s, i)) ≡ #(s) + 1

satisfying their preconditions[4]), the induction rule (†) gets simplified to

for every legal constructor ground term e of type **D**, $\Phi[x/e] \vdash (\forall\, x)\, \Phi(x)$.

This is so because every constructor ground term that is not legal is equivalent to some legal constructor ground term by the above assumption.

If the above assumption about the behavior of σ is dropped and nothing is assumed about its behavior on inputs not satisfying the preconditions, then we have

for every legal constructor ground term e of type **D**, $\Phi[x/e] \vdash$

$$(\forall\, x)\Big(\bigvee_{i=1,\,m} (\exists\, x_{i_1}, \ldots, x_{i_{n_i}})\,[\, x \equiv \sigma_i(x_{i_1}, \ldots, x_{i_{n_i}}) \wedge \mathbf{P}_{\sigma_i}(x_{i_1}, \ldots, x_{i_{n_i}}) \equiv \mathbf{T}\,]\Big) \Rightarrow \Phi(x),$$

where $\{\, \sigma_1, \ldots, \sigma_m \,\}$ is the set of constructors of **D**. The condition in the matrix of the consequence of the above rule ensures that x ranges over values serving as the interpretations of the legal ground terms of **D**. This is the strongest consequence we can have because the interpretation of illegal constructor ground terms is not known. For example, if we drop the restrictions in the specification of **Stk-Int** repeated in Figure 4.7 specifying the exceptional behavior of the operations, the modified specification associates preconditions with the constructors **Pop** and **Replace**. The induction rule would then be

for every legal constructor ground term e of type **Stk-Int**, $\Phi[s/e] \vdash$

$(\forall\, s)\,(\, s \equiv \mathbf{Null}() \vee (\exists\, s', i')\, s \equiv \mathbf{Push}(s', i') \vee (\exists\, s')\,[\, \sim \mathbf{Empty}(s') \equiv \mathbf{T} \wedge s \equiv \mathbf{Pop}(s')\,]$

$\vee\, (\exists\, s', i')\,[\, \sim \mathbf{Empty}(s') \equiv \mathbf{T} \wedge s \equiv \mathbf{Replace}(s', i')\,]\,) \Rightarrow \Phi(s).$

We have discussed in Chapter 3 the reasons for assuming that a constructor σ on an input not satisfying its precondition can either signal an exception or return a value constructible by the constructors using inputs satisfying their preconditions. An additional reason for this assumption is that otherwise the induction rule gets complex, as should be evident from the above discussion.

---

4. σ can also signal on such an input; since we are considering data types without exceptional behavior, this choice is ruled out.

4.2.5 The Full Theory

In proving properties of programs, one often uses properties of data types other than equations and inequalities. For example, we often need to prove properties of the form '$(e_{11} \equiv e_{21} \wedge \ldots \wedge e_{1n} \equiv e_{2n}) \Rightarrow (f_1 \equiv f_2)$.' Or, we may need a formula involving existential quantifiers. For example, consider the **union** procedure on sets of integers written in a CLU-like language and given in Figure 4.8. The integer variable i inside the loop defines the range (-i+1, i-1) of integers which have been checked to be members of the first argument and if so, have been inserted into the result being computed. The variable i is incremented every time the loop is executed. To prove the termination of **union**, we need to show that a set is either empty or there is an integer k such that every element of the set lies in the range (-k, k). The following formula expresses this property:

(6) $(\forall\, s)\,[\,s \equiv \mathbf{Null} \vee (\exists\, k)\ (\forall\, j)\,[\,\mathbf{Has}(s, j) \equiv \mathbf{T} \Rightarrow (\,j \leq k \wedge j \geq -k\,)\,]\,]$

To prove such properties, we need the whole machinery of first order predicate calculus with identity. The proof of (6) is given in Figure 4.9.

The full theory **Th(S)** is the set of formulas derivable from the nonlogical axioms of S and **Th(S')**, where S' is a specification of a defining type or an auxiliary type used in S, using the logical axioms and rules of inference of multi-sorted first order predicate calculus

---

Figure 4.8. Procedure Union - I

```
union = proc(s1, s2 : Set-Int') returns (Set-Int')
        i : Int := 0
        r1 : Set-Int' := s1
        r2 : Set-Int' := s2
        while ~ Set-Int'$Size(r1) = 0 do
         if Set-Int'$Has(r1, i) then r1 := Set-Int'$Remove(r1, i)
                                     r2 := Set-Int'$Insert(r2, i)
         end
         if Set-Int'$Has(r1, -i) then r1 := Set-Int'$Remove(r1, -i)
                                      r2 := Set-Int'$Insert(r2, -i)
         end
         i := i+1
        end
        return (r2)
        end union
```

Figure 4.9. Proof of the Formula (6)

To prove $(\forall s)[s \equiv Null() \vee (\exists i)(\forall j)[Has(s, j) \equiv T \Rightarrow (j \leq i \wedge j \geq -i)]]$

Using the scheme (5),
$\Phi(s) = [s \equiv Null \vee (\exists i)(\forall j)[Has(s, j) \equiv T \Rightarrow (j \leq i \wedge j \geq -i)]]$
Basis $\Phi(Null) <=> T$
Inductive Step Assume $\Phi(s)$, to show $(\forall k)\,\Phi(Insert(s, k))$
Since $\Phi(s) \Leftrightarrow T$, we have two cases,
Case 1 $s \equiv Null()$
$\Phi(Insert(Null, k)) \Leftrightarrow T$, because i is $|k|$, the absolute of k
Case 2 $(\exists i)(\forall j)[Has(s, j) \equiv T \Rightarrow (j \leq i \wedge j \geq -i)]$
Subcase 1 $-i \leq k \leq i$,
i itself serves to prove that $\Phi(Insert(s, k)) \Leftrightarrow T$ from $\Phi(s)$
Subcase 2 $k > i \vee k < -i$
$|k|$ serves as i to prove that $\Phi(Insert(s, k)) \Leftrightarrow T$ from $\Phi(s)$.

Using the scheme (5), we have $(\forall s)\,\Phi(s)$.

---

with identity, as well as the infinitary induction rule (†).

The following diagram summarizes the relationships among different subtheories and the full theory:

Th(S)	First Order Predicate Calculus + Infinite Induction Rule
∪	
IND(S)	+ Infinite Induction Rule
∪	
DS(S)	+ Proof by Contradiction
∪	
EQ(S)	Four Rules of ≡ and the Substitution Rule of ∀

The following theorem shows that the above deductive system is sound.

Thm. 4.2 For any two ground terms e_1 and e_2,

(i) if '$e_1 \equiv e_2$' $\in$ Th(S), then e_1 and e_2 are observably equivalent by S (i.e., observably equivalent in the models in F(S)), and

(ii) if '$e_1 \not\equiv e_2$' $\in$ Th(S), then e_1 and e_2 are distinguishable by S.

Proof The theorem follows from the facts that (a) the nonlogical axioms hold in the models in F(S) with $\equiv$ interpreted as the observable equivalence relation, (b) the observable equivalence relations are preserved by the functions in the models in F(S). ∎

4.2.6 Properties of a Specification

We can define properties desirable of a specification by requiring that various subtheories and the full theory derived from the specification satisfy certain conditions. Guttag and Horning [28] have discussed the sufficient completeness property for a restricted class of specifications, which has been found useful. We state that property in our framework. We extend it to specifications using auxiliary functions and specifying preconditions for the operations. The sufficient completeness property captures the intuitive notion that the behavior of the observers is completely specified on intended inputs and that the result of an observer on an intended input can be deduced by equational reasoning. We relate this property to the behavioral completeness property defined in the previous chapter and show that sufficient completeness is stronger than behavioral completeness (Theorem 4.4) because behavioral completeness only requires that the behavior of the observers be completely specified on intended inputs and it does not say anything about what can be deduced from the specification.

When specifications are used to prove properties of programs using the data types being specified, we often need to relate different constructor sequences. In that case, it is desirable to have a specification satisfy a stronger property than sufficient completeness, which in addition to the requirement that the behavior of the observers can be deduced by equational reasoning on any intended input, also requires that the equivalence of the observable effect of different constructors can be deduced by equational reasoning. We call this property the completeness property of a specification and define it precisely. We

later see that for a complete and consistent specification S, formulas in **IND(S)** can be proved using the Knuth-Bendix algorithm (see Subsection 4.2.7).

Recall from Section 3.5 that for a consistent and behaviorally complete specification S, the models in $F(S)$ are behaviorally equivalent w.r.t. $\{ P_\sigma \mid \sigma \in \Omega \}$. Furthermore, if S does not specify any nontrivial precondition for the operations, the semantics of a specification S is a single data type, a set of behaviorally equivalent algebras. In that case, for any two ground terms of type D, they are either observably equivalent by S or distinguishable by S An obvious question is whether the proposed deductive system is powerful enough to deduce this from a consistent and behaviorally complete specification. We show that it is not the case. But if a specification is consistent and complete, then the deductive system has this property.

Since S is hierarchical, S should preserve the specifications of the types used in S. S should only specify the behavior of the operations of D, and it should not specify the behavior of a type D' used in S that is not captured by its specification S'. Specifications so designed are modularly structured; they support the factoring and hierarchical structuring of the proof of correctness of a hierarchically designed implementation. We define the well definedness property of a specification which captures this modularity requirement.

Before we discuss these properties, we prove

Thm. 4.3 For a consistent S, for any two ground terms e_1 and e_2 of the same type, both '$e_1 \equiv e_2$' and '$e_1 \not\equiv e_2$' cannot be in **Th(S)**.

Proof If S is consistent, then $F(S) \neq \emptyset$.

Suppose for some e_1 and e_2, both '$e_1 \equiv e_2$' and '$e_1 \not\equiv e_2$' are in Th(S). '$e_1 \equiv e_2$' $\in$ Th(S) implies that e_1 and e_2 are observably equivalent by S. Similarly, '$e_1 \not\equiv e_2$' $\in$ Th(S) implies that e_1 and e_2 are distinguishable by S, which is a contradiction. ∎

4.2.6.1 Sufficient Completeness

As was said earlier for constructors, for a specification specifying nontrivial preconditions for the operations, one is interested in ground terms in which the input to every occurrence of an operation symbol satisfies the associated precondition. This is so because an operation is not likely to be invoked with an input not satisfying the specified precondition. Even if the operation is invoked on such an input, we are interested in its behavior. Furthermore, if a specification uses auxiliary functions, ground terms in which auxiliary functions appear are also not of interest because they are not used in programs using the data type. Earlier we defined a legal constructor ground term (Def. 4.3); below, we extend the definition to a ground term.

Def. 4.4 A ground term e is called *legal* if and only if (i) e does not have any occurrence of an auxiliary function, and (ii) for every subterm of e of the form $e_1 = \sigma(e_{11}, \ldots, e_{1n})$, where $\sigma \in \Omega$, '$\mathbf{P}_\sigma(e_{11}, \ldots, e_{1n}) \equiv \mathbf{T}' \in \mathbf{EQ(S)}$. ∎

For a specification using auxiliary functions and specifying nontrivial preconditions, only legal ground terms are interesting. If such a specification is consistent and behaviorally complete, any two legal ground terms are either observably equivalent by S or distinguishable by S (see Section 3.5).

In [28], Guttag and Horning define the sufficient completeness property of specifications which do not specify a nontrivial precondition for the operations and do not use auxiliary functions. We state their definition in our framework.

Def. 4.5 A specification S is *sufficiently complete* if and only if for every ground term e of type $\mathbf{D}' \in \Delta$, there exists a theorem derivable from S of the form '$e \equiv e'$', where e' is a ground term of type D' without any occurrence of an operation symbol of D. ∎

In [28], the deductive system to be used to derive a theorem is not specified. Guttag [33] requires that the equation '$e \equiv e'$' be in the equational subtheory EQ(S).

The sufficient completeness property can be extended to specifications using auxiliary functions and specifying nontrivial preconditions for the operations. For auxiliary functions, there are two possible extensions:

(i) Consider only the ground terms expressed using the operation symbols, because only these terms can be used in a program, or

(ii) consider all ground terms, thus requiring that auxiliary functions also be completely specified.

We take the former approach; however, we recommend that whenever an auxiliary function is used, it be completely specified.

Def. 4.6 A specification is *sufficiently complete* if and only if for every legal ground term e of type $D' \in \Delta$, a formula '$e \equiv e'$' $\in$ EQ(S), where e' is a legal ground term of type D' without having any operation symbol of D or any auxiliary function. ∎

For example, the specification of **Set-Int'** is not sufficiently complete, because for instance, a legal ground term **Choose(Insert(Insert((Null, 1), 2))** cannot be related to any ground term of type **Int** that does not have any occurrence of an operation symbol of **Set-Int'**.

The following theorem relates sufficient completeness to behavioral completeness. The intuition behind this result is that if the behavior of observers on intended inputs can be deduced by equational reasoning from S, then the observers must be completely specified by S.

Thm. 4.4 If a specification S is sufficiently complete, then S is behaviorally complete.

Proof: See Appendix III. ∎

The converse of the above theorem however does not hold. So, the sufficient completeness property is strictly stronger than behavioral completeness, as there are specifications which are behaviorally complete but are not sufficiently complete. This is so because in the definition of sufficient completeness, only a fragment of the deductive system of first order predicate calculus is used to derive properties from the specification. There can exist a legal ground term e of type $D' \in \Delta$ such that we cannot derive '$e \equiv e'$' for any e' of type D' not having any occurrence of an operation symbol of D in the equational subtheory EQ(S). However, we can derive the above equation in Th(S) using other rules in addition to the rules of the equational subtheory. We illustrate this point using the specification of **Set-Int'**. We add another axiom defining **Choose** on sets of size > 1 as

returning the maximum integer in the set.

8. Choose(Insert(Insert(s, i1), i2)) ≡ if Size(s) = 0 then (if ~ i1 = i2 then Max(i1, i2)) else (if ~ i1 = i2 then Max(Choose(Insert(s, i1)), i2) else Choose(Insert(s, i1))).

The modified specification is not sufficiently complete, because **Choose (Insert(Null, i))** is not directly specified. Nor can we deduce by equational reasoning that '**Choose(Insert(Null, i)) ≡ i.**' However, using the theorem of **Int**, '**(i = j ≡ T) ⇒ i ≡ j**' derived using the induction rule for integers, the axioms 3, 4, and 7 of **Set-Int'**, and case analysis, we can prove by contradiction that

Choose(Insert(Null, i)) ≡ i

It should be obvious that with a minor modification of the proof of Theorem 4.4, we can prove the following generalization of Theorem 4.4:

Thm. 4.5 If for every legal ground term e of type $D' \in \Delta$, there exists a ground term e' of type D' not having any operation symbol of D and auxiliary function such that '$e \equiv e'$' $\in$ **Th(S)**, then S is behaviorally complete. ∎

Theorem 4.4 can be derived as a corollary of the above theorem. We conjecture that the converse of the above theorem is also true, which says that the deductive system is complete with respect to deducing the behavior of an observer on an intended input.

Conjecture 4.1 If S is behaviorally complete, then for every legal ground term e of type $D' \in \Delta$, there exists a ground term e' of type D' not having any operation symbol and auxiliary function such that '$e \equiv e'$' $\in$ **Th(S)**.

We can prove the following partial completeness result about the deductive system in proving the distinguishability of legal ground terms of type D', $D' \in \Delta \cup \{ D \}$.

Thm. 4.6 For a consistent and sufficiently complete S, if any two legal ground terms e_1 and e_2 of type D are distinguishable by S, then '$e_1 \not\equiv e_2$' $\in$ **DS(S)**.

Proof See Appendix III. ∎

If conjecture 4.1 is true, then we can prove a similar result about behaviorally complete specifications: For a consistent and behaviorally complete specification S, if any two legal

ground terms e_1 and e_2 of type D are distinguishable by S, then '$e_1 \not\equiv e_2$' $\in$ Th(S).

4.2.6.2 Completeness

We cannot prove a similar result about the observable equivalence of legal ground terms of type D, because we do not have a rule analogous to proof by contradiction in the deductive system that enables us to prove the observable equivalence of ground terms unless explicitly specified by the nonlogical axioms. Different but equivalent specifications of the same data type can differ in the extent to which the observable equivalence relation of legal ground terms of D can be proved from the nonlogical axioms. For example, the terms **Insert(Insert(Null, 2), 2)** and **Insert(Null, 2)** are observably equivalent by **Set-Int'**, but '**Insert(Insert(Null, 2), 2) ≡ Insert(Null, 2)**' $\notin$ Th(Set-Int'). If we add the following axiom to the specification of **Set-Int'**:

9. **Insert(Insert(s, i1), i2) ≡ if i1 = i2 then Insert(s, i1) else Insert(Insert(s, i2), i1),**

then '**Insert(Insert(Null, 2), 2) ≡ Insert(Null, 2)**' $\in$ EQ(Set-Int'). The semantics of the modified specification is the same as the semantics of the original specification of **Set-Int'**. The more a specification of D captures the observable equivalence relation on terms of type D, the more useful it is in deriving the theory of D and hence in proving properties of programs using D. We define below a property of a specification requiring it to completely specify the observable equivalence relation. We put a stronger requirement: We want EQ(S), instead of Th(S), to have a formula '$e_1 \equiv e_2$' for two legal ground terms e_1, e_2 if and only if e_1 and e_2 are observably equivalent by S, so that such formulas can be derived by purely equational reasoning (i.e., using the rules of ≡ and the substitution rule for ∀).

Def. 4.7 A sufficiently complete specification S is *complete* if and only if assuming that the specification S' of each D' $\in \Delta \cup A_t$ is complete, for any two legal grounds terms e_1 and e_2 of the same type, '$e_1 \equiv e_2$' $\in$ EQ(S) if and only if e_1 and e_2 are observably equivalent by S. ∎

The completeness property of a specification should not be confused with the completeness property of a theory of an algebraic structure as defined in Logic [16]. Using Theorems 4.4 and 4.6, and the fact that for a consistent and behaviorally complete specification, any two

legal ground terms are either observably equivalent or distinguishable by S, we have

Thm. 4.7 For a consistent and complete specification S, for any legal ground terms e_1 and e_2 of the same type, either '$e_1 \equiv e_2$' $\in$ DS(S) or '$e_1 \not\equiv e_2$' $\in$ DS(S). ∎

Musser [61] has called a specification from which either '$e_1 \equiv e_2$' or '$e_1 \not\equiv e_2$' can be derived in DS(S) to be *fully specified*, though his view of a specification is somewhat different. He views the operator '$\equiv$' as another operation of a data type, whereas we consider '$\equiv$' as a predicate in the underlying logic used to construct formulas.

4.2.6.3 Well Definedness

We would like a specification S to be modular, i.e., for the specification S′ of each D′ $\in \Delta \cup A_t$, Th(S) $|_{L(S')}$ = Th(S′). This means that Th(S) does not have a formula expressed using symbols in L(S′) that is not in Th(S′). Only those properties which involve an operation symbol of D and/or auxiliary functions used in S can be proved from S; a formula not having any operation symbol of D or an auxiliary function in S and not in Th(S′) cannot be proved from S.

For a consistent and sufficiently complete specification, the following holds:

Thm. 4.8 For a consistent and sufficiently complete S, for any legal ground terms e_1', e_2' of type D′ $\in \Delta$ constructed using the symbols in L(S′), if neither '$e_1' \equiv e_2'$' $\in$ Th(S′) nor '$e_1' \not\equiv e_2'$' $\in$ Th(S′), where S′ is a specification of D′, '$e_1' \not\equiv e_2'$' $\notin$ Th(S).

Proof By contradiction.

Suppose '$e_1' \not\equiv e_2'$' $\in$ Th(S) meaning that e_1' and e_2' are distinguishable by S (as well as by S′) (by Theorem 4.2). By Theorem 4.6, '$e_1' \not\equiv e_2'$' $\in$ Th(S′), which is not the case. So the theorem. ∎

However, we could have a specification S such that '$e_1' \equiv e_2'$' $\in$ Th(S) in the above case. The following property of a specification rules out such cases.

Def. 4.8 A specification S is *well defined* if and only if for every $D' \in \Delta \cup A_t$, assuming that S' of D' is well defined, $Th(S)|_{L(S')} = Th(S')$. ∎

We are usually interested in well defined and complete specifications. Behaviorally incomplete specifications are occasionally of interest. **Set-Int'** is such an example.

4.2.7 Automation of IND(S)

Recently Musser [61] has discussed how to automate **IND(S)** when S satisfies certain conditions. If (i) S is consistent and complete, and (ii) the nonlogical axioms derived from S can be written as equations (possibly using **if-then-else** operator), then the Knuth-Bendix algorithm, which treats equational axioms as rewrite rules, can be used to derive an equational formula '$e_1 \equiv e_2$' in the inductive subtheory **IND(S)**. The equation '$e_1 \equiv e_2$' is input to the algorithm as a rewrite rule to get a new convergent set of rules having the added rewrite rule. There are three possibilities:

(i) The algorithm succeeds implying that the new equation is consistent with the nonlogical axioms and thus provable,

(ii) an inconsistency, such as '$e_1' \rightarrow e_2'$' where e_1' and e_2' can be proved to be not equal, in particular '**T** → **F**' or '**F** → **T**,' is generated as a rule, implying that the equation is not a theorem, and

(iii) the algorithm does not terminate implying that (a) an additional lemma be proved first, which could be guessed from the set of new rules generated, (b) the specified ordering on terms used by the algorithm does not work, and some other ordering needs to be tried, or (c) there does not exist a finite convergent set of rules to express **IND(S)**.

The basis of deducing from (ii) that '$e_1 \equiv e_2$' is not a theorem is the consistency of S and the method of proof by contradiction; in fact '$e_1 \not\equiv e_2$' is a theorem in **IND(S)** in this case. The basis of deducing from (i) that '$e_1 \equiv e_2$' is a theorem in **IND(S)** is the completeness of the specifications: For a substitution of all variables in e_1 and e_2 by ground terms, the resulting ground terms e_1' and e_2' have the property that either '$e_1' \equiv e_2'$' ∈ **IND(S)** or '$e_1' \not\equiv e_2'$' ∈ **IND(S)**.

4.3 Theory of Exceptions Without Nondeterminism

We now incorporate the exceptional behavior of data types into their theories with the assumption that specifications do not specify nondeterministic operations. New atomic formulas are introduced to express the exceptional behavior of the operations. We describe how the nonlogical axioms of **Th(S)** can be derived in this case from a specification S. We discuss how to construct **EQ(S)**, **DS(S)**, **IND(S)**, and **Th(S)**. New 'logical' axioms characterizing the exceptional behavior of the operations are presented. We extend the properties of a specification discussed in the previous section to specifications specifying the exceptional behavior. For illustration, we modify the specification of **Set-Int'** so that the operation **Choose** is required to signal **no-element()** on the empty set; let **Set-Int"** stand for the modified **Set-Int**. So, instead of the Restrictions component specifying a precondition for **Choose**, it specifies a required exception condition as follows:

#(s) = 0 ⇒ Choose(s) signals no-element().

We also use the specification of **Stk-Int.**

Besides the operation symbols and auxiliary function symbols, the language **L(S)** also includes the names of exceptions signalled by the operations as specified in S. Exception terms are constructed as discussed in Chapter 2, using terms and exception names. There are two new sets of atomic formulas in addition to equations:

(a) **e signals** *ext*,

where e is a term, *ext* is an exception term, and every variable in *ext* is also in e; and

(b) $ext_1 \equiv ext_2$,

where ext_1 and ext_2 are exception terms. The predicate '**signals**' is similar to ≡ but its arity is **(D ∪ EXV) x EXV.**

As in the previous section, we first discuss the derivation of the nonlogical axioms of **Th(S)** from S. Then, we discuss the subtheories **EQ(S)**, **DS(S)**, and **IND(S)**, and the full theory **Th(S)**. In the last subsection, we extend sufficient completeness, completeness, and well definedness properties.

4.3.1 Derivation of Nonlogical Axioms

The nonlogical axioms of **Th(S)** are derived from the restrictions and axioms components of the specification S in a slightly different way than discussed in Subsection 4.2.1. We first discuss the restrictions, and later the formulas in the axioms component.

4.3.1.1 Restrictions Component

From a restriction specifying a required exception signalled by an operation σ,

$$\mathbf{R}_i(X) \Rightarrow \sigma(X) \textbf{ signals } ext,$$

we get the following nonlogical axiom:

$$\mathbf{P}_\sigma(X) \Rightarrow (\mathbf{R}_i(X) \Rightarrow \sigma(X) \textbf{ signals } ext),$$

because the restriction holds only if the input X satisfies the precondition associated with σ.[5] For example, the restriction on the operation **Top** in the specification of **Stk-Int**,

Empty(s) ⇒ Top(s) signals no-top( ),

is a nonlogical axiom of **Th(Stk-Int)**, as the precondition for **Top** is T. Similarly, from a restriction specifying an optional exception signalled by an operation σ,

$$\sigma(X) \textbf{ signals } ext \Rightarrow \mathbf{O}_j(X),$$

we get

$$\mathbf{P}_\sigma(X) \Rightarrow (\sigma(X) \textbf{ signals } ext \Rightarrow \mathbf{O}_j(X)),$$

as a nonlogical axiom. For example, the restriction on **Push**,

Push(s, i) signals overflow(s, i) ⇒ #(s) ≥ 100,

is a nonlogical axiom of **Th(Stk-Int)**.

---

5. Recall that the boolean term $\mathbf{R}_i(X)$ is an abbreviation for the formula $\mathbf{R}_i(X) \equiv \mathbf{T}$.

4.3.1.2 Axioms Component

The preconditions in the restrictions component are also used in constructing the nonlogical axioms from the formulas in the axioms component of S. As discussed in Chapter 3, a variable in a formula in the axioms component cannot be freely substituted. When the exceptional behavior was not considered in Subsection 4.2.1, the substitution was conditional: The arguments to every operation in the axiom must satisfy the associated precondition. Now, there is an additional requirement: The substitution should not result in an operation signalling on its arguments.

To express the second condition, we introduce a unary auxiliary function $N?_{D'} : D' \cup EXV \rightarrow \textbf{Bool}$ for every $D' \in \Delta \cup \{ D \} \cup A_t$. These auxiliary functions are not used in a specification. Informally, N? separates a normal value of D' from an exception: It returns T if its argument interprets to a normal value of D'; it returns F if its argument signals an exception. Furthermore, $N?_{D'}(\sigma(e_1, \ldots, e_n))$ is F if $N?_{D_i}(e_i)$ is false for any e_i; this constraint on the behavior of $N?_{D'}$ enables us to get a simpler transformation of the restricted formulas in the axioms component of S.

Using $N?_{D'}$, we transform a restricted formula in the axioms component to an unrestricted formula which serves as a nonlogical axiom of Th(S). If an equation '$e_1 \equiv e_2$' is in the axioms component, where e_1 and e_2 are of type D', then the corresponding unrestricted axiom is

$$(N?_{D'}(e_1) \wedge N?_{D'}(e_2)) \Rightarrow ((PC_{e_1} \wedge PC_{e_2}) \Rightarrow e_1 \equiv e_2),$$

where PC_e is a conjunction of conditions expressing the constraint that the input to every operation invocation in a term e satisfies the associated precondition. Similarly, if a restricted formula is '$e_1 \equiv$ **if** b **then** e_2,' then the corresponding unrestricted formula is

$$(N?_{Bool}(b) \wedge N?_{D'}(e_1) \wedge N?_{D'}(e_2)) \Rightarrow ((PC_b \wedge PC_{e_1} \wedge PC_{e_2}) \Rightarrow (b \Rightarrow e_1 \equiv e_2)).$$

If a restricted formula is '$e_1 \equiv$ **if** b **then** e_2 **else** e3,' then the corresponding unrestricted formulas are obtained using the fact that this formula is equivalent to two conditional equations

$e_1 \equiv$ **if** b **then** e_2

$e_1 \equiv$ **if** $\sim b$ **then** e_3.

We illustrate the above transformation on the following equation in the axioms component of the specification of **Stk-Int**:

Replace(s, i) ≡ Push(Pop(s), i).

The corresponding unrestricted axiom is

$(N?_{Stk\text{-}Int}(Replace(s, i)) \wedge N?_{Stk\text{-}Int}(Push(Pop(s), i))) \Rightarrow$
$(\sim Empty(s) \Rightarrow Replace(s, i) \equiv Push(Pop(s), i))$.

4.3.1.3 Definition of $N?_{D'}$

A specification of D implicitly defines $N?_D$ and extends $N?_{D'}$ for every defining type D′ of D as well as any auxiliary types D′ used in S. $N?_{D'}$ is defined by the specification of D′. Since an operation σ has the arity $D_1 \times \ldots \times D_n \rightarrow D' \cup EXV$, and $N?_{D'}$ has the arity $D' \cup EXV \rightarrow$ **Bool**, we need to introduce variables ranging over values of a type and exceptions to characterize $N?_{D'}$. We have two options: (i) Introduce two kinds of variables - variables of a single type D_i and variables of a union type $D_i \cup EXV$, or (ii) introduce only variables of a union type. If we adopt the second alternative, the formulas expressing the normal behavior of the operations get long because we make the conditional use of the variables. Since we would mostly be using formulas expressing normal behavior, we have adopted the first alternative. Often, we do not need to have a formula in which both kinds of variables are mixed. Except in the axioms for $N?_{D'}$ and the axioms characterizing the general properties of the exceptional behavior of the data type, we would rarely use variables of a union type. Terms as well as exception terms are constructed using only variables ranging over a single type (except in the next section). Henceforth, we use *xe*, $xe_1, \ldots, xe_n, \ldots$, *ye*, $ye_1, \ldots, ye_n, \ldots$, *ze*, $ze_1, \ldots, ze_n$, etc., as variables of a union type, and *exv*, $exv_1, \ldots, exv_n, \ldots$ as variables of type **EXV**.

We now discuss the axioms defining $N?_{D'}$. First of all, for a variable *x* of type D′, we have the axiom

$$N?_{D'}(x) \equiv T.$$

For an operation σ, let $P_\sigma(X)$ be its precondition. Let us assume that the restrictions component specifies for σ, *l* required exceptions and m optional exceptions. For each $1 \leq i \leq l$, let $R_i(X)$ be the condition on input *X* when σ is required to signal an exception;

similarly, for each $1 \leq j \leq m$, let $O_j(X)$ be the condition when σ has an option to signal.

For every constructor σ of D, we have an axiom defining N? corresponding to D,

$$N?(XE) \Rightarrow ((P_\sigma(XE) \wedge (\sim R_1(XE) \wedge \ldots \wedge \sim R_f(XE)) \wedge (\sim O_1(XE) \wedge \ldots \wedge \sim O_m(XE))) \Rightarrow N?(\sigma(XE)))$$

where XE stands for the variables $xe_1, \ldots, xe_n$; xe_i is a variable of union type $D_i \cup EXV$, and $N?(XE)$ is an abbreviation for $N?_{D_1}(xe_1) \wedge \ldots \wedge N?_{D_n}(xe_n)$.

The above axiom captures the assumption in a specification that if (i) an input to a constructor σ is normal, (ii) the input satisfies the precondition associated with σ, (iii) none of the conditions associated with a required exception for σ holds for the input, and (iv) the condition an input must satisfy in case σ signals an exception specified to be optional, also does not hold for the input, then σ returns a normal value. In other words, this assumption states that the exceptional behavior of the operations on their intended inputs must be completely specified by the Restrictions component.

The extension of the definition of $N?_{D'}$ for every $D' \in \Delta$ is also captured by a similar set of axioms corresponding to every observer $\sigma \in \Omega$ of result type D'. There is an axiom having the above structure corresponding to every observer σ in Ω.

In addition to the above axioms, we have a rule for every operation and auxiliary function expressing that if any argument to a function is not normal, then the result of the function invocation is also not normal.

$$(N?_{D_1}(xe_1) \equiv F \vee \ldots \vee N?_{D_n}(xe_n) \equiv F) \vdash N?_{D'}(\sigma(xe_1, \ldots, xe_n)) \equiv F.$$

Note that there is no axiom so far which states the condition when $N?_{D'}$ is F. In the next subsection on equational subtheory, we introduce a rule characterizing such behavior of $N?_{D'}$.

We use the nonlogical axioms derived from the restrictions and axioms components of S, and the axioms defining $N?_{D'}$ along with the additional axioms and rules characterizing the general properties about the exceptional behavior to build various subsets of Th(S) and finally Th(S) itself.

4.3.2 Equational Subtheory

As in case of specifications without nondeterminism and without exceptional behavior, we define the equational subtheory EQ(S) as a set of atomic formulas. Besides equations of the kind discussed in Subsection 4.2.2, we also have the following atomic formulas:

(a) e **signals** ext, and

(b) $ext_1 \equiv ext_2$.

In addition to the rules characterizing $\equiv$ discussed in Subsection 4.2.2, we use the substitution rule for $\forall$, and the rules characterizing 'signals' and capturing the observable equivalence relation on exception values. The substitution rule for $\forall$,

$$(\forall\, x)\, \Phi(x) \Rightarrow \Phi[x/e],$$

where x is a variable of type D', and e is a term of type D' and is substitutible for x in Φ [16], is modified to

$$(\forall\, x)\, \Phi(x) \Rightarrow (\, N?_{D'}(e) \equiv T \Rightarrow \Phi[x/e]\,),$$

since x is a variable ranging over normal values and e can signal an exception.

Rule (i) below says when $N?_{D'}$ is false, which is if a term of type D' signals an exception, then $N?_{D'}$ on that term is false. Rule (ii) states that if two terms are observably equivalent and one signals an exception, then the other also signals the same exception. Rule (iii) states that if a term signals two exceptions, then the exceptions are observably equivalent. Rule (iv) states how the observable equivalence relation on exception values is related to the observable equivalence relations on normal values.

(i) xe **signals** $exv \vdash N?_{D'}(xe) \equiv F$,

(ii) $xe_1 \equiv xe_2$, xe_1 **signals** $exv \vdash xe_2$ **signals** exv

(iii) xe **signals** exv_1, xe **signals** $exv_2 \vdash exv_1 \equiv exv_2$, and

for every exception name ex of arity $D_1 \times \ldots \times D_n$,

(iv) $x_{11} \equiv x_{21}, \ldots, x_{1n} \equiv x_{2n} \vdash ex(x_{11}, \ldots, x_{1n}) \equiv ex(x_{21}, \ldots, x_{2n})$.

It should be obvious that the above rules are sound under the following interpretation: In a type algebra **A**, for a ground term e and a ground exception term ext, the formula 'e **signals** ext' is interpreted as: The interpretation of e in **A** is the exception value that is the interpretation of ext in **A**. For two ground exception terms ext_1 and ext_2, the formula

'$ext_1 \equiv ext_2$' is interpreted as: The interpretation of ext_1 is observably equivalent to the interpretation of ext_2 in EXV of **A**.

We now show how to use the above rules along with the nonlogical axioms and the axioms and rules defining $N?_{D'}$, to prove some properties of data types. Since many nonlogical axioms and formulas are conditional having the form

(7) $b \Rightarrow e$ **signals** *ext*,

where b is a boolean term, we use a trick similar to the one used in Subsection 4.2.2 to deal with such formulas so that they can be proved in EQ(S). We introduce an auxiliary function **if-then** : **Bool** x **EXV** x D' → $D' \cup$ **EXV** having the behavior defined by the following axioms:

if-then(**T**, *ext*, *e*) **signals** *ext*

if-then(**F**, *ext*, *e*) ≡ *e*.

Using the auxiliary function **if-then**, the formula (7) is equivalent to

e ≡ **if-then**(*b*, *ext*, *e*),

as for an instantiation of the variables in (7), if b interprets to **T**, then (7) is equivalent to '*e* **signals** *ext*.' The boolean term b must not signal.

As an illustration, we prove from the nonlogical axioms of **Stk-Int** that '**Top(Null) signals no-top()**' ∈ EQ(**Stk-Int**) in Figure 4.10. Similarly, we can prove

Top(Pop(Push(Null, i))) signals no-top().

Replace(Push(Push(Null, i1), i2), i3)) ≡ Push(Push(Null, i1), i3).

However,

Replace(Push^{101}((Null, 1), . . . , 101), 0) ≡ Push^{101}(((Null, 1), . . . , 100), 0)

is not derivable because we cannot derive '$N?_{\text{Stk-Int}}$(l.h.s.) ≡ T' due to the optional exception specified for **Push** when its stack argument is of size ≥ 100. But we can prove the

---

Figure 4.10. Proof of 'Top(Null) signals no-top()'

1. Top(s) ≡ if-then(Empty(s), no-top(), Top(s))	Restriction on Top
2. Empty(Null) ≡ T	Axiom 4
3. if-then(Empty(Null), no-top(), Top(Null)) signals no-top()	Axiom of if-then
4. Top(Null) signals no-top()	Substitution in 1, and rule (ii) above

following formula:

$$N?_{\text{Stk-Int}}(\text{Push}^{101}((\text{Null}, 1), \ldots, 101)) \Rightarrow$$
$$\text{Replace}(\text{Push}^{101}((\text{Null}, 1), \ldots, 101), 0) \equiv \text{Push}^{101}(((\text{Null}, 1), \ldots, 100), 0).$$

The formula

$$\text{Pop}(\text{Null}) \equiv \text{Null}$$

is not derivable because of the precondition on **Pop**.

It would be interesting to investigate the conditions under which

(i) an axiom of the form '*e* **signals** *ext*' can be treated as a rewrite rule '$e \rightarrow ext$' and the Knuth-Bendix algorithm be applicable to such axioms, and

(ii) a conditional formula involving **signals** can be rewritten as an equation using the **if-then** and **if-then-else** operators so that the Knuth-Bendix algorithm is applicable to conditional formulas also.

4.3.3 Distinguishability Subtheory

As in case of specifications without nondeterminism and without exceptional behavior, DS(S) is defined to be a set consisting of atomic formulas and the negations of atomic formulas. DS(S) includes EQ(S) as well as formulas having the following structure:

(a) $e_1 \not\equiv e_2$,

(b) $ext_1 \not\equiv ext_2$, and

(c) *e* **signals** *ext*,

where '*e* **signals** *ext*' is an abbreviation for '$\sim (\forall\ x_1, \ldots, x_n)$ [*e* **signals** *ext*]' such that $x_1, \ldots, x_n$ are all the variables in the formula '*e* **signals** *ext*.' Besides the axioms and rules of inference of DS(S) discussed in Subsection 4.2.3, we have the following additional axioms and rules expressing properties about the exceptional behavior of data types which enable us to prove formula having the above structure.

(v) for every exception name $ex : D_1 \times \ldots \times D_n$,

$$(\sim x_{11} \equiv x_{21} \vee \ldots \vee \sim x_{1n} \equiv x_{2n}) \vdash \sim ex(x_{11}, \ldots, x_{1n}) \equiv ex(x_{21}, \ldots, x_{2n}).$$

(vi) for different exception names $ex_1 : D_1 \times \ldots \times D_n$ and $ex_2 : D'_1 \times \ldots \times D'_m$ in L(S),

$$\sim ex_1(x_{11}, \ldots, x_{1n}) \equiv ex_2(x_{21}, \ldots, x_{2m}),$$

(vii) for a union type $D' \cup \mathbf{EXV}$,

$$N?_D(xe_1) \equiv T, N?_D(xe_2) \equiv F \vdash \sim xe_1 \equiv xe_2,$$

where xe_1 and xe_2 are of type D' ∪ EXV, and

(viii) $N?(xe) \equiv T \vdash \sim (\forall\, exv)$ [*xe* **signals** *exv*]

Rule (v) and axiom (vi) capture the distinguishability relation on exception values. Rule (v) is the opposite of rule (iv) given in the previous subsection; it states that two exception values having the same name are distinguishable if any of the arguments in one value is distinguishable from the corresponding argument in the other value. Axiom (vi) states that two exception values are distinguishable if their exception names are different. Rule (vii) states that two values are distinguishable if $N?_D$ holds for one and does not hold for the other. Rule (viii) says that if $N?_D$ holds for a term, then it cannot signal an exception. The above axiom and rules are clearly sound. Note that these rules can be used to derive formulas having the structure '$\sim xe_1 \equiv xe_2$,' which implies that '$xe_1 \not\equiv xe_2$.'

We can derive from the nonlogical axioms of Stk-Int using rule (vii) that

(8) **Top(Null)** $\not\equiv i$,

because '**Top(Null) signals no-top()**,' $N?_{Int}(i) \equiv T$,' and '$N?_{Int}$(**Top(Null)**) ≡ **F**' ∈ DS(Stk-Int). The formula

overflow(s, i) $\not\equiv$ **no-top()**

is immediate from the axiom (vi) above. Using the theorem (8) in DS(Stk-Int), we can prove by contradiction that

Null $\not\equiv$ **Push(s, i)**.

4.3.4 Inductive Subtheory

The inductive subtheory IND(S) can be constructed as in Subsection 4.2.4; we can also use the above axioms and rules characterizing the exceptional behavior. The induction rule (†) in Subsection 4.2.4 has to be modified; instead of requiring that for every constructor ground term *e* of type D, $\Phi[x/e]$ be derivable in the premise, we only need to consider constructor ground terms for which '$N?_D(e) \equiv T$' is derivable. So, we have:

Modified Induction Rule

Given a formula Φ(x) with a free variable x of type D.

For every constructor ground term *e* of type D, $N?_D(e) \equiv T \Rightarrow \Phi[x/e] \vdash (\forall\, x)\, \Phi(x)$.

We can use the methods discussed in Subsubsection 4.2.4.3 to establish the infinitely many premises.

As in Subsubsection 4.2.4.4, if a specification S specifies nontrivial preconditions on constructors, then the above formula can be simplified to

for every legal constructor ground term e of type D, $N?_D(e) \equiv T \Rightarrow \Phi[x/e]$

$\vdash (\forall x)\, \Phi(x)$,

because of the assumption about the semantics of a constructor on inputs not satisfying the associated precondition, discussed in Chapter 3.

For example, for **Stk-Int**, the induction rule is:

For every legal constructor ground term e of type **Stk-Int**,

$N?(e) \equiv T \Rightarrow \Phi[s/e] \vdash (\forall s)\, \Phi(s)$.

The above rule can be simplified using the following theorem in a way similar to **Set-Int'** in the previous section:

Thm. 4.9 Every legal constructor ground term e of type **Stk-Int** such that '$N?_{\text{Stk-Int}}(e) \equiv T$' $\in$ **EQ(Stk-Int)**, is equivalent by equational reasoning to another legal constructor ground term e' having only **Null** and **Push**, i.e., if '$N?_{\text{Stk-Int}}(e) \equiv T$' $\in$ **EQ(Stk-Int)**, then '$e \equiv e'$' $\in$ **EQ(Stk-Int)**.

Proof By induction on the number of **Pop** and **Replace** in a constructor ground term e using axioms 1 and 3 in Figure 4.7. See the details in Appendix III. ∎

The simplified induction rule is:

(9) For every legal constructor ground term e of type **Stk-Int** having occurrences of **Null** and **Push** only, $N?_{\text{Stk-Int}}(e) \equiv T \Rightarrow \Phi[s/e] \vdash (\forall s)\, \Phi(s)$.

4.3.5 The Full Theory

The full theory **Th(S)** is also constructed in a similar way as for data types without exceptional behavior. For example, we can prove the normal form theorem using the simplified induction rule (9):

$s \equiv \text{Null}() \vee (\exists s', i')\, [\, s \equiv \text{Push}(s', i')\,]$.

The diagram summarizing the relationships among different subtheories for specifications not specifying exceptional behavior on p. 135 also holds in this case.

For the extended deductive system, the following extension of Theorem 4.2 holds:

Thm. 4.10 (i) For any two ground terms e_1 and e_2 of the same type, if '$e_1 \equiv e_2$' $\in$ **Th(S)**, then e_1 and e_2 are observably equivalent by S and if '$e_1 \not\equiv e_2$' $\in$ **Th(S)**, then e_1 and e_2 are distinguishable by S,

(ii) for a ground term e and a ground exception term ext, if 'e **signals** ext' $\in$ **Th(S)**, then the interpretation of e in every model **A** in $\mathcal{F}$(S) is the interpretation of ext in **A**,

(iii) for two ground exception terms ext_1 and ext_2, if '$ext_1 \equiv ext_2$' $\in$ **Th(S)**, then ext_1 and ext_2 are observably equivalent by S, and if '$ext_1 \not\equiv ext_2$' $\in$ **Th(S)**, then ext_1 and ext_2 are distinguishable by S, and

(iv) for any ground term e, if '**N?**(e) $\equiv$ **T**' $\in$ **Th(S)**, then the interpretation of e in every model **A** in $\mathcal{F}$(S) is a normal value, and if '**N?**(e) $\equiv$ **F**' $\in$ **Th(S)**, then the interpretation of e in **A** is either an exception value or undefined.

Proof The theorem follows from the facts that

(a) the nonlogical axioms of Th(S) hold in every model in $\mathcal{F}$(S),

(b) the observable equivalence relation used as the interpretation of $\equiv$ is a congruence,

(c) the exceptional behavior of an operation is completely specified by the restrictions component of S on inputs satisfying its preconditions, and

(d) the axioms and rules defining N? and characterizing the exceptional behavior holds in every type algebra. ∎

We demonstrate how the full theory constructed from a specification S can be used to prove properties of programs using the data types specified by S. Figure 4.11 is another implementation of **union** procedure using **Choose** in a CLU-like language. In this implementation, an element of the first set argument to **union** is successively selected using the operation **Choose**, removed from the copy of the first argument, and inserted into the copy of the second argument until the operation **Choose** signals **no-element**, indicating that the set is empty. The handler for **no-element** associated with the loop is then invoked. In

Figure 4.11. Procedure Union - II

```
union = proc(s1, s2 : Set-Int") returns (Set-Int")
        i : Int
        r1 : Set-Int" := s1
        r2 : Set-Int" := s2
          { r1 ≡ s1 ∧ r2 ≡ s2 }
        while true do
            { (Size(r1) = 0 ≡ F ∧ IN(Remove(r1, Choose(r1)), Insert(r2, Choose(r1)), s1, s2))
              ∨ (Size(r1) = 0 ≡ T ∧ R_r2^union(s1, s2)) }
          i := Set-Int"$Choose(r1)
            { IN(Remove(r1, i), Insert(r2, i), s1, s2) }
          r1 := Set-Int"$Remove(r1, i)
          r2 := Set-Int"$Insert(r2, i)
            { IN(r1, r2, s1, s2) }
        end except when no-element :
                    end
        { R_r2^union(s1, s2) }
        return (r2)
        { R }
        end union
```

$$IN(r1, r2, s1, s2) = \neg(\forall j)[(Has(s1, j) \vee Has(s2, j)) \Leftrightarrow (Has(r1, j) \vee Has(r2, j)) \equiv T] \wedge (Size(r1) + Size(r2)) \leq (Size(s1) + Size(s2)) \equiv T \wedge Size(r2) > 0 \equiv T)$$

I/O Specification for **union**

$T \Rightarrow R$, where $R = R1 \wedge R2$, and

$R1 = (\forall i)[(Has(s1, i) \vee Has(s2, i)) \Leftrightarrow Has(union(s1, s2), i) \equiv T]$

$R2 = Size(union(s1, s2)) \leq Size(s1) + Size(s2) \equiv T$

---

the code, we have included formulas within '{ }' that express relations among different variables at that point in the code. The Floyd-Hoare inductive assertion method for proving properties of programs [17, 36, 55] can be extended to incorporate the exceptional behavior of programs. A statement in this case can terminate in more than one way - either normally or by signalling an exception. Corresponding to every possible way of termination of a statement, we associate an input formula for an output formula.

Figure 4.11 includes the input-output specification of **union**. We use the following notation for specifying a procedure $F(X)$: Corresponding to every possible

outcome of F on an input X, there is a formula relating the input to the outcome. Since F can terminate normally or by signalling an exception, we specify the weakest input condition for normal termination, as well as for every exception signalled by F.

$$TC_1(X) \Rightarrow F(X) \textit{ signals } ext_1$$
$$\vdots$$
$$TC_m(X) \Rightarrow F(X) \textit{ signals } ext_m$$
$$TC_{m+1}(X) \Rightarrow R(X, r),$$

where $TC_1(X), \ldots, TC_{m+1}(X)$, and R are first order formulas, and r stands for a possible result returned by F on the input X. '$TC_i(X) \Rightarrow F(X)$ *signals* ext_i' is interpreted as: The weakest input condition for F to terminate by signalling ext_i is $TC_i(X)$. '$TC_{m+1}(X) \Rightarrow R(X, r)$' is similarly interpreted as: The weakest input condition for F to terminate normally returning a value r such that $R(X, r)$ holds is $TC_{m+1}(X)$. If F is deterministic, then such an r is unique for every X; otherwise, there can be many r's such that $R(X, r)$ holds. Instead of using r as denoting a result returned by F on X, we can also use $F(X)$.

The formula 'IN(r1, r2, s1, s2)' is used as an invariant of the loop in the program in Figure 4.11. Using the backward substitution semantics of the control structures, we can generate the verification conditions and show the required formulas to be in Th(Set-Int'). The partial correctness proof of **union** is complete if we can show that

$$\mathbf{IN(r1, r2, s1, s2) \Rightarrow}$$
$$\mathbf{((Size(r1) = 0 \equiv F \wedge IN(Remove(r1, Choose(r1)), Insert(r2, Choose(r1)), s1, s2))}$$
$$\mathbf{\vee (Size(r1) = 0 \equiv F \wedge R_{r2}^{union(s1, s2)}))}$$

To prove the above formula, we need the theorem

$$\mathbf{Size(r1) > 0 \equiv T \Rightarrow Size(Remove(r1, Choose(r1))) + 1 \equiv Size(r1).}$$

The while loop terminates because each time in the loop, **Size(r1)** is reduced, and **Choose(r1)** signals **no-element** when **Size(r1) = 0 ≡ T.**

An alternate approach to the Floyd-Hoare method of reasoning about programs is to use the first order semantics of control structures as suggested by Cartwright and

McCarthy [8]. They have shown how reasoning about recursive programs can be completely carried in first order logic. The definition of a recursive program can be considered as an axiom defining the function computed by the program with an appropriate condition on variables.[6] The termination of such a program can also be proved by adding a minimization scheme corresponding to its function. For example, the above iterative **union** program can be transformed to an equivalent recursive program, and the axiom characterizing the function computed by the program is derived from the recursive program. **Th(Set-Int")** is enriched by adding this axiom about **union** and a minimization scheme corresponding to **union**. The input output specification of **union** can then be proved as a theorem in the enriched theory. We use a similar approach in the next chapter in showing the correctness of an implementation.

4.3.6 Properties of a Specification

It should be clear from the discussion in the previous subsections that the following extension of Theorem 4.3 holds:

Thm. 4.11 For a consistent **S**,

(i) for any ground terms e_1 and e_2 of the same type, both '$e_1 \equiv e_2$' and '$e_1 \not\equiv e_2$' cannot be in **Th(S)**, and

(ii) for any two ground exception terms ext_1 and ext_2, both '$ext_1 \equiv ext_2$' and '$ext_1 \not\equiv ext_2$' cannot be in **Th(S)**, and

(iii) for any ground term e, both '**N?**(e) $\equiv$ **T**' and '**N?**(e) $\equiv$ **F**' cannot be in **Th(S)**. ∎

We extend the definitions of sufficient completeness, completeness, and well definedness properties discussed in Subsection 4.2.6 to the specifications specifying exceptional behavior. The results about these properties in Subsection 4.2.6 directly extend when the modified definitions are used.

---

6. The condition is that a variable is instantiated to a value of its type other than $\perp$, which is used to denote non-termination.

4.3.6.1 Sufficient Completeness

Recall that the sufficient completeness property as defined in Subsection 4.2.6 requires that the behavior of the observers on any intended input should be deducible by equational reasoning. When a specification specifies data types having operations which signal exceptions, then the observable behavior of the operations also includes their exceptional behavior. Two values of a data type can also be distinguished in this case if a sequence of operations signals one exception on one value and does not signal on the other, or if the sequence of operations signals different exceptions on different values. In the extended definition of sufficient completeness, we want to capture the intuition that in addition to the normal behavior of the observers, a sufficient complete specification must also completely specify the exceptional behavior of the operations when their input satisfy the associated preconditions.

If a specification has only required exception conditions for the operations, then the above amounts to requiring that

(i) for any legal ground term e, either '$N?(e) \equiv T$' $\in$ EQ(S) or '$N?(e) \equiv F$' $\in$ EQ(S), and

(ii) (a) if '$N?(e) \equiv T$' $\in$ EQ(S) and e is of type $D' \in \Delta$, then the condition stated in Def. 4.6 must be satisfied (i.e., there is a ground term e' not having any operation symbol of D or auxiliary functions used in S such that '$e \equiv e'$' $\in$ EQ(S)), and

(b) if '$N?(e) \equiv F$' $\in$ EQ(S) and for every subterm e_1 of e, '$N?_D(e_1) \equiv T$' $\in$ EQ(S), then the formula 'e **signals** *ext*' $\in$ EQ(S) for some ground exception term *ext*.

If S specifies optional exceptions also, then there are legal ground terms for which neither '$N?(e) \equiv T$' nor '$N?(e) \equiv F$' is provable. For example, we can neither prove

$$N?_{Int}(Top(Push^{101}((Null, 1), \ldots, 101))) \equiv T$$

nor

$$N?_{Int}(Top((Push^{101}(((Null, 1), \ldots, 101))) \equiv F$$

from the specification of **Stk-Int**. For such a specification, the definition of sufficient completeness must include the condition that for such a ground term, if we assume '$N?_D(e) \equiv T$,' then '$e \equiv e'$' is derivable using equational reasoning. This condition is based on an aspect of the semantics of a specification, namely that if an operation does not signal on an input for which it had the option to signal, then the formulas in the axioms

component for the operation behavior must hold.

Def. 4.9 A specification S is *sufficiently complete* if and only if

(i) for every e of type $D' \in \Delta$, if '$N?(e) \equiv T$' $\in$ EQ(S), then there is a theorem '$e \equiv e'$' $\in$ EQ(S) for some e', a ground term of type D' not having any operation symbol of D and auxiliary function in S,

(ii) for every e ($= \sigma(e_1, \ldots, e_n)$) of type $D' \in \Delta \cup \{ D \}$, if '$N?(e) \equiv F$' $\in$ EQ(S), and '$(N?(e_1) \wedge \ldots \wedge N?(e_n)) \equiv T$' $\in$ EQ(S), then there is a theorem 'e signals ext' $\in$ EQ(S) for some ground exception term ext, and

(iii) for every legal ground term e of type $D' \in \Delta \cup \{ D \}$, if neither '$N?(e) \equiv T$' $\in$ EQ(S) nor '$N?(e) \equiv F$' $\in$ EQ(S), then there exists a subterm e_1 of e such that $e_1 = \sigma(e_{11}, \ldots, e_{1n})$ and '$O[x_1/e_{11}, \ldots, x_n/e_{1n}] \equiv T$' $\in$ EQ(S), where σ is specified to optionally signal if its input satisfies $O(x_1, \ldots, x_n)$, and assuming '$N?(e) \equiv T$,' there is a theorem '$e \equiv e'$' $\in$ EQ(S $\cup$ { $N?(e) \equiv T$ }), where e' is a ground term of type D' having no operation symbol of D and auxiliary function used in S. ∎

S $\cup$ { f } stands for the nonlogical axioms derived from S plus the formula f, and EQ(S $\cup$ { f }) stands for the equational subtheory derived using S $\cup$ { f } as the nonlogical axioms. The condition (iii) above amounts to proving the theorem assuming '$N?(e) \equiv T$.'

For example, **Stk-Int** is sufficiently complete. '**Top(Null) signals no-top()**' $\in$ EQ(S). Assuming '$N?_{Int}$(**Top(Push^{101}((Null, 1), . . . , 101))** $\equiv$ **T**,' we can derive '**Top(Push^{101}((Null, 1), . . . , 101))** $\equiv$ **101**' in EQ(S).

The specification of **Set-Int**'' is not sufficiently complete, because, for instance, though '$N?_{Int}$(**Choose(Insert(Insert(Null, 0), 1))** $\equiv$ **T**' $\in$ EQ(S), there does not exist any ground term e' of type **Int** not having any operation symbol of **Set-Int**'' such that '**Choose(Insert(Insert(Null, 0), 1))** $\equiv e'$' $\in$ EQ(S).

The results discussed about specifications not specifying exceptional behavior in Subsection 4.2.6 directly extend to specifications specifying exceptional behavior when appropriately modified. We have

Thm. 4.12 If S is sufficiently complete, then S is behaviorally complete.

Proof See Appendix III. ∎

The obvious analog to Theorem 4.5 also holds; its converse is a conjecture analogous to Conjecture 4.1. We also have

Thm. 4.13 For a consistent and sufficiently complete S, if any two legal ground terms e_1 and e_2 of type D are distinguishable by S, then '$e_1 \not\equiv e_2$' $\in$ DS(S).

Proof See Appendix III. ∎

4.3.6.2 Completeness and Well Definedness

The completeness property of a specification can be defined in this case in the same way as in Subsection 4.2.6. Def. 4.7 in Subsection 4.2.6 works for this case also. Theorem 4.7 for this case can be proved in the same way as for specifications without exceptional behavior. It can be shown that the specification of **Stk-Int** is complete, whereas the specification of **Set-Int"** is not complete.

The well definedness property is also defined in the same way as in case of specifications without exceptional behavior. Def. 4.8 in Subsection 4.2.6 is valid. It can be shown that the specifications of **Set-Int"** and **Stk-Int** are well defined.

4.4 Theory of Nondeterminism

In this section, we discuss specifications specifying nondeterministic operations. Again, we first discuss specifications without exceptional behavior; later, we incorporate the exceptional behavior also. For the first part, we modify the specification of **Set-Int'** given in Figure 4.1 so that the operation **Choose** is specified to be nondeterministic. Let **Set-Int'''** stand for the modified specification. In the second part, we use the specification of **Set-Int** given in Figure 3.1.

We find it convenient to express properties of a data type with nondeterministic operations as formulas using nondeterministic operation symbols (which is also the reason to allow a specification to have such formulas in the axioms component), but such a formula must be interpreted properly. A nondeterministic function symbol does not have the substitution property with respect to $\equiv$ unless interpreted properly. We discussed this in the previous chapter; we will repeat the discussion here. For example, the formula '**Choose**(s) $\in$ s $\equiv$ T' in the specification is to be interpreted as any integer returned by **Choose** on the argument s is in the set s. The formula

$$s1 \equiv s2 \Rightarrow \textbf{Choose}(s1) \equiv \textbf{Choose}(s2)$$

need not hold if '**Choose**(s1) $\equiv$ **Choose**(s2)' is interpreted as an integer returned by **Choose** on s1 is the same as an integer returned by **Choose** on s2, because different invocations of **Choose** on the same argument may return different integers. However, if we interpret '**Choose**(s1) $\equiv$ **Choose**(s2)' as for every possible integer returned by **Choose** on s1, **Choose** on s2 can return the same integer, and vice versa, then the formula

$$s1 \equiv s2 \Rightarrow \textbf{Choose}(s1) \equiv \textbf{Choose}(s2)$$

holds. We adopt the latter interpretation, so that the substitution property continues to hold.[7] The adopted interpretation is consistent with the definition of observable equivalence on ground terms involving nondeterministic operations induced by S, given in Sections 2.2 and 2.3.

---

7. As is discussed in the previous chapter, the reason for rejecting the former interpretation is that the formula '$\sigma(x_1, \ldots, x_n) \equiv \sigma(x_1, \ldots, x_n)$' for a nondeterministic symbol σ is almost always false under it.

We cannot however express many interesting properties about a data type because in a formula involving a nondeterministic operation symbol σ, different occurrences of a term $\sigma(e_1, \ldots, e_n)$ may result in different values. We often need to express properties in which different occurrences of the term $\sigma(e_1, \ldots, e_n)$ stand for the same value. For example, consider another version of the **union** procedure given in Figure 4.12, which is a slight modification of the version given in Figure 4.11. In this case, the while loop has the condition '$\sim$ (#(s) = 0),' instead of 'true' in Figure 4.11. In verifying this version of **union**, we must use the properties of i, a result returned by **Choose**. In such a case, we introduce an auxiliary function $\sigma\_p : D_1 \times \ldots \times D_n \times D' \rightarrow$ **Bool** corresponding to the nondeterministic operation σ, which is the relation describing the behavior of σ.

$$(*)\quad \sigma\_p(x_1, \ldots, x_n, y) \triangleq \begin{cases} T & \text{if } \sigma \text{ can return } y \text{ as a possible result on } x_1, \ldots, x_n, \\ F & \text{otherwise} \end{cases}$$

For example, we introduce **Choose_p** for **Choose** and use **Choose_p** to express a property of i, a result returned by **Choose**.

Since formulas in the axioms component of S are expressed using nondeterministic operation symbols, we transform them to equivalent formulas having only deterministic symbols using the auxiliary functions corresponding to the nondeterministic symbols. We discuss the transformation procedure **TR** below. L(S) now also includes the auxiliary function σ_p corresponding to every nondeterministic operation symbol σ. The transformed formulas have a restricted interpretation just as the original formulas in the axioms component, so we derive unrestricted formulas from the transformed formulas using the method discussed in Section 4.2 for specifications with deterministic operations. The precondition specified by a nondeterministic operation σ is taken as the precondition for the corresponding auxiliary function σ_p. So in the specification of **Set-Int'''**, '$\sim$ #(s) = 0' is the precondition for **Choose_p**. The unrestricted formulas serve as the nonlogical axioms of S. To prove a formula f involving nondeterministic operation symbols, we first transform f using **TR**, and then prove **TR**(f) from the nonlogical axioms of S.

The transformation procedure **TR** must embed the semantics of S assumed in Chapter 3. Recall that the semantics of S only requires that for every data type in $D(S)$, the

semantics of S, an operation specified to be nondeterministic must return an appropriate value on every input; the operation in every data type in $\mathcal{D}(S)$ need not have the maximum amount of nondeterminism specified by S.

4.4.1 Transformation Procedure TR

We first describe the procedure **TR** and later verify that **TR**(f) is semantically equivalent to f. Before describing the transformation procedure, we illustrate it using examples. Consider the following formula in the axioms component of **Set-Int'''**:

Choose(s) $\in$ **s** $\equiv$ **T**

Figure 4.12. Procedure Union - III

```
union = proc(s1, s2 : Set-Int''') returns (Set-Int''')
        i : Int
        r1 : Set-Int''' := s1
        r2 : Set-Int''' := s2
          { r1 ≡ s1 ∧ r2 ≡ s2 }
        while ~ Set-Int'''$Size(r1) = 0 do
            { Choose_p(r1, i) ≡ T ∧ IN(Remove(r1, i), Insert(r2, i), s1, s2) }
          i := Set-Int'''$Choose(r1)
            { IN(Remove(r1, i), Insert(r2, i), s1, s2) }
          r1 := Set-Int'''$Remove(r1, i)
          r2 := Set-Int'''$Insert(r2, i)
            { IN(r1, r2, s1, s2) }
        end
        { R_r2^union(s1, s2) }
        return (r2)
        { R }
        end union
```

$$IN(r1, r2, s1, s2) = (\forall j)\,[\,(Has(s1, j) \vee Has(s2, j)) \Leftrightarrow (Has(r1, j) \vee Has(r2, j)) \equiv T\,] \wedge (Size(r1) + Size(r2)) \leq (Size(s1) + Size(s2)) \equiv T \wedge Size(r2) > 0 \equiv T$$

I/O Specification for **union**

$T \Rightarrow R$, where $R = R1 \wedge R2$, and

$R1 = (\forall i)\,[\,(Has(s1, i) \vee Has(s2, i)) \Leftrightarrow Has(union(s1, s2), i) \equiv T\,]$

$R2 = Size(union(s1, s2)) \leq Size(s1) + Size(s2) \equiv T$

The above formula states that every value returned by **Choose** is in the set s. The transformed formula obtained after applying the procedure would be

((∀ i) [Choose_p(s, i) ≡ T ⇒ i ∈ s ≡ T] ∧ (∃ i) Choose_p(s, i) ≡ T)

The second conjunct states that **Choose** returns at least one value on every input. The unrestricted formula, which serves a nonlogical axiom of **Set-Int'''**, is obtained using the precondition for **Choose**; it is given below:

((∀ i) [~ #(s) = 0 ≡ T ⇒ (Choose_p(s, i) ≡ T ⇒ i ∈ s ≡ T)] ∧

(∃ i) [~ #(s) = 0 ≡ T ⇒ Choose_p(s, i) ≡ T)

Let us consider another formula ' **Choose**(s1) ≡ **Choose**(s2).' This states that for every value returned by **Choose** on s1, there is an observably equivalent value returned by **Choose** on s2, and vice versa. TR transforms this formula to

~ ((∀ i1) [Choose_p(s1, i1) ≡ T ⇒ (∃ i2) [Choose_p(s2, i2) ∧ i1 ≡ i2]] ∧

(∀ i2) [Choose_p(s2, i2) ≡ T ⇒ (∃ i1) [Choose_p(s1, i1) ∧ i1 ≡ i2]])

We now present the transformation procedure **TR**, which is defined inductively making use of the structure of a formula.

Basis *f* is an atomic formula '$e_1 \equiv e_2$.'

(a) *f* does not have any occurrence of a nondeterministic operation symbol:

$\mathbf{TR}(f) \triangleq f$

(b) both e_1 and e_2 have occurrences of nondeterministic operation symbols:

We wish **TR**(*f*) to roughly express that for every instance of the free variables in *f*, for every possible choice made about the invocations of the nondeterministic operation symbols in e_1, there are choices for the invocations of the nondeterministic operation symbols in e_2 such that the instantiations of e_1 and e_2 return equivalent results, and vice versa.

TR('$e_1 \equiv e_2$') has the following structure:

$(\forall z_1, \ldots, z_m)\,[\,c_1 \Rightarrow (\exists y_1, \ldots, y_p)\,[\,c_2 \wedge e_1' \equiv e_2'\,]\,] \wedge$

$(\forall y_1, \ldots, y_p)\,[\,c_2 \Rightarrow (\exists z_1, \ldots, z_m)\,[\,c_1 \wedge e_1' \equiv e_2'\,]\,],$

where $z_1, \ldots, z_m$ are new variables such that corresponding to each occurrence of a nondeterministic operation symbol σ in e_1, say the occurrence $\sigma(e_{i1}, \ldots, e_{im})$, there is a variable z_i to stand for the possible result returned by σ on its input. The formula c_1 is a

conjunction of the equations of the form '$\sigma\_p(e_{i1}, \ldots, e_{in}, z_i) \equiv T$', stating conditions on z_i. Similarly for e_2, new variables $y_1, \ldots, y_p$ are introduced, and c_2 is obtained from e_2. e_1' and e_2' are obtained from e_1 and e_2 respectively, by substituting $z_1, \ldots, z_m$ and $y_1, \ldots, y_p$ for subterms having nondeterministic operations as the outermost operation in e_1 and e_2 respectively. We discuss later how c_1 and e_1' are constructed from e_1, and c_2 and e_2' are obtained from e_2.

(c) only one side of the equation '$e_1 \equiv e_2$' has occurrences of nondeterministic operation symbols. Without any loss of generality, we assume that only the l.h.s. has occurrences of nondeterministic symbols.

Construct c_1 and e_1' from e_1 as discussed above. Then,

$$\mathbf{TR}(`e_1 \equiv e_2') = (\forall\, z_1, \ldots, z_m)\,[\, c_1 \Rightarrow e_1' \equiv e_2\,] \wedge (\exists\, z_1, \ldots, z_m)\, c_1$$

This completes the basis step of the definition of **TR**. The second conjunct is to ensure that there is at least one value returned by e_1.

Inductive Step

Since all other logical symbols can be expressed in terms of $\wedge$, $\vee$ and $\forall$, we define how **TR** works on formulas having these symbols.

(a) if f is $\sim f_1$, then $\mathbf{TR}(f) = \sim \mathbf{TR}(f_1)$

(b) if f is $f_1 \wedge f_2$, then $\mathbf{TR}(f) = \mathbf{TR}(f_1) \wedge \mathbf{TR}(f_2)$

(c) if f is $(\forall\, x)\, f_1$, then $\mathbf{TR}(f) = (\forall\, x)\, \mathbf{TR}(f_1)$.

This completes the definition of **TR**.

For instance, a conditional equation '$b \Rightarrow e_1 \equiv e_2$,' where b is a boolean term, is transformed to

$$b \Rightarrow \mathbf{TR}(`e_1 \equiv e_2'),$$

if b does not have any nondeterministic operation symbols. If b has nondeterministic symbols, then the conditional equation is transformed to

$$\mathbf{TR}(`b \equiv T') \Rightarrow \mathbf{TR}(`e_1 \equiv e_2')$$

$$= ((\forall\, z_1', \ldots, z_k')\,[\, c \Rightarrow b' \equiv T\,] \wedge (\exists\, z_1', \ldots, z_k')\, b') \Rightarrow \mathbf{TR}(`e_1 \equiv e_2').$$

Since such a b is assumed to behave deterministically (See Section 3.1), i.e., for an instantiation of the free variables X in the conditional equation, b interprets either to **T** or to **F**, the above formula agrees with the interpretation of a conditional equation assumed in

Section 3.2 on the semantics of a specification.

We now describe how to construct c and e' from a term e by induction on the number of occurrences of nondeterministic operation symbols in e. Let k stand for the number of occurrences of nondeterministic operation symbols in e.

Basis $k = 1$

Let $e^1 = \sigma(e_1^1, \ldots, e_n^1)$ be the subterm of e having the nondeterministic operation σ as its outermost operation symbol. Then c is '$\sigma\_p(e_1^1, \ldots, e_n^1, z_1) \equiv T$' and e' is obtained by replacing e^1 in e by z_1. The type of z_1 is the range type of σ.

Inductive Step Assume c and e' can be constructed if e has $k' < k$ occurrences of nondeterministic symbols. Show for k.

(i) If e has the subterm having k occurrences of nondeterministic operation symbols, let the subterm be $e^1 = \sigma(e_1, \ldots, e_n)$, where σ is a nondeterministic operation symbol. Each e_i has less than k occurrences of nondeterministic operation symbols. By the inductive step, let $c_1, \ldots, c_n$ be the formulas obtained by applying this procedure on $e_1, \ldots, e_n$ respectively, and let $e_1', \ldots, e_n'$ be the terms obtained by replacing subterms having nondeterministic operation symbols by new variables in $e_1, \ldots, e_n$ respectively. Then

$$c = \sigma\_p(e_1', \ldots, e_n', z_k) \equiv T \wedge c_1 \wedge \ldots \wedge c_n,$$

and e' is obtained by replacing e^1 in e by z_k.

(ii) There is no such subterm of e_1. Consider all outermost subterms of e_1 having a nondeterministic operation symbol as their outermost operation; let them be $e_1, \ldots, e_n$. Each of these subterms has less than k number of occurrences of nondeterministic operation symbols. By inductive step, let $c_1, \ldots, c_n$ be the formulas obtained by transforming $e_1, \ldots, e_n$ respectively, and let $e_1', \ldots, e_n'$ be the terms obtained by replacing subterms having nondeterministic operation symbols by new variables in $e_1, \ldots, e_n$, respectively. Then

$$c = c_1 \wedge \ldots \wedge c_n,$$

and e' is obtained by replacing $e_1, \ldots, e_n$ by $e_1', \ldots, e_n'$ respectively.

This completes the discussion about how c and e' are obtained from e.

Thm. 4.14 *f* and **TR**(*f*) are semantically equivalent.

Proof See Appendix III. ∎

4.4.2 Th(S)

The nonlogical axioms obtained as discussed above are used to prove properties about the data type. A nonlogical axiom involves existential quantifiers in contrast to a nonlogical axiom of a specification specifying only deterministic operations. So, the whole machinery of first order predicate calculus is needed to prove an arbitrary equation or an inequality involving nondeterministic symbols. So it is not meaningful to discuss the subtheories EQ(S), DS(S), and IND(S); we instead discuss the full theory **Th**(S). The formulas are proved in the same way as in case of specifications specifying deterministic operations only.

As an illustration of the use of **Th**(S), we verify the version of the procedure **union** given in Figure 4.12. Note that the backward substitution semantics of the assignment statement

i := Set-Int$Choose(r1)

is given as

$\{ \textbf{Choose\_p(r1, i')} \equiv T \wedge P^{i}_{i'} \}$ **i := Set-Int$Choose(r1)** $\{ P \}$,

instead of

$\{ P^{i}_{\textbf{Choose(r1)}} \}$ **i := Set-Int$Choose(r1)** $\{ P \}$,

because different occurrences of the expression **Choose**(r1) could possibly return different results. For example, the verification condition

{ IN(Remove(r1, Choose(r1)), Insert(r2, Choose(r1)), s1, s2) }

i := Set-Int$Choose(r1) { IN(Remove(r1, i), Insert(r2, i), s1, s2) }

is not true, where as

{ Choose_p(r1, i') ≡ T ∧ IN(Remove(r1, i'), Insert(r2, i'), s1, s2) }

i := Set-Int$Choose(r1) { IN(Remove(r1, i), Insert(r2, i), s1, s2) }

is true. In this case also, 'IN(r1, r2, s1, s2)' serves as an invariant of the loop. Using the backward substitution semantics of the control structures, we can generate the verification

conditions and show the required formulas to be in Th(Set-Int'''). The partial correctness proof of **union** is complete if we can show that

$(\sim Size(r1) = 0 \equiv T \wedge IN(r1, r2, s1, s2)) \Rightarrow$

$(Choose\_p(r1, i) \equiv T \wedge IN(Remove(r1, i), Insert(r2, i), s1, s2))$

To prove the above formula, we need the theorem

$Size(r1) > 0 \equiv T \Rightarrow Size(Remove(r1, Choose(r1))) + 1 \equiv Size(r1)$.

The termination is also ensured because each time in the loop, Size(r1) is reduced, so the loop condition will eventually become false.

We think that many properties of nondeterministic operations expressed as equations and inequalities can be derived from the untransformed nonlogical axioms (the nonlogical axioms obtained from the formulas in the Axioms component of the specification before applying TR) using techniques employed for deterministic operations, for instance, viewing equations as rewrite rules and using Knuth-Bendix algorithm for deriving properties. We have not investigated the extent to which this can be done. This hypothesis is another reason for preferring to write specifications directly using nondeterministic operation symbols as compared to writing them indirectly using the relations corresponding to nondeterministic operations.

4.4.3 Data Types with Exceptional Behavior

We discuss the modifications required to incorporate the exceptional behavior specified by the specifications with nondeterministic operations. We describe how to derive the nonlogical axioms from a specification. We use the original specification of Set-Int given in Figure 3.1 for illustration; the specification is repeated in Figure 4.13.

As before, an auxiliary function σ_p is associated with every nondeterministic operation symbol σ. σ_p is not strict with respect to its last argument.

Figure 4.13. Specification of Set-Int

Operations

Null	**: → Set-Int**	*as* **Ø**
Insert	**: Set-Int X Int → Set-Int**	
Remove	**: Set-Int X Int → Set-Int**	
Has	**: Set-Int X Int → Bool**	*as* $x_2 \in x_1$
Size	**: Set-Int → Int**	*as* $\#(x_1)$
Choose	**: Set-Int → Int**	*nondeterministic*
	→ no-element()	

Restrictions

#(s) = 0 ⇒ Choose(s) *signals* **no-element**

Axioms

Remove(Ø, i) ≡ Ø
Remove(Insert(s, i1), i2) ≡ if i1 = i2 then Remove(s, i1) else Insert(Remove(s, i2), i1)
i ∈ Ø ≡ F
i1 ∈ Insert(s, i2) ≡ if i1 = i2 then T else i1 ∈ s
#(Ø) ≡ 0
#(Insert(s, i)) ≡ if i ∈ s then #(s) else #(s) + 1
Choose(s) ∈ s ≡ T

$$\sigma : D_1 \times \ldots \times D_n \rightarrow D' \cup EXV$$

$$\sigma\_p : D_1 \times \ldots \times D_n \times (D' \cup EXV) \rightarrow \textbf{Bool},$$

$$\sigma\_p(x_1, \ldots, x_n, ze) \triangleq \begin{cases} T & \text{if } N?(ze) \equiv T \text{ and } \sigma \text{ can return } ze \\ & \text{as a possible result on } x_1, \ldots, x_n, \\ T & \text{if } N?(ze) \equiv F \text{ and } \sigma \text{ signals } ze \text{ on } x_1, \ldots, x_n, \\ F & \text{otherwise} \end{cases}$$

Recall that *ze* is of union type.

We extend the transformation procedure TR discussed in the previous subsection. Besides equations, we have two additional kinds of atomic formulas: '*e* **signals** *ext*' and '$ext_1 \equiv ext_2$'. TR for equations is same as in the previous subsection except that the new variables introduced in the transformation are of union type.

An exception name is treated like a deterministic operation symbol, so '$ext_1 \equiv ext_2$' is treated like an equation '$e_1 \equiv e_2$'. **TR** is extended to treat 'e **signals** ext' as '$e \equiv ext$.' **TR** is applied on '$e \equiv ext$.' In the transformed formula, a subformula of the form '$e' \equiv ext$' ' wherever ext' is an exception term and e' is a non-variable term, is replaced by the subformula 'e' **signals** ext'.' Note that a transformed formula may involve terms constructed using variables ranging over union types.

The restrictions on a nondeterministic operation σ are transformed to get the nonlogical axioms as follows: A restriction specifying a required exception for σ,

$\mathbf{R}_i(X) \Rightarrow \sigma(X)$ **signals** ext,

is transformed to

$\mathbf{P}_\sigma(X) \Rightarrow (\mathbf{R}_i(X) \Rightarrow \sigma\_\mathbf{p}(X, ext) \equiv \mathbf{T})$.

For example, from the restriction on **Choose**,

#(s) = 0 ⇒ Choose(s) signals no-element(),

we get

#(s) = 0 ⇒ Choose_p(s, no-element()) ≡ T.

A restriction specifying an optional exception for σ,

$\sigma(X)$ **signals** $ext \Rightarrow \mathbf{O}_j(X)$,

is transformed to

$\mathbf{P}_\sigma(X) \Rightarrow (\sigma\_\mathbf{p}(X, ext) \equiv \mathbf{T} \Rightarrow \mathbf{O}_j(X) \equiv \mathbf{T})$.

Axioms defining $\mathbf{N?}_{D'}$ are constructed the same way as for the specification with deterministic operations except that there is no axiom due to a nondeterministic operation σ because the range of the corresponding auxiliary function $\sigma\_\mathbf{p}$ is **Bool** and not **Bool** ∪ **EXV**. In addition to the axioms and rules expressing general properties of the exceptional behavior of the operations discussed in the previous sections, we have another rule. Recall that a nondeterministic operation can either signal an exception or has the choice to return one of many possible normal values. An operation does not have the choice between returning a normal value and signalling an exception on the same input. This property is captured by the following axiom for every nondeterministic operation σ:

$\sim((\exists\, ze)[\sigma\_\mathbf{p}(X, ze) \equiv \mathbf{T} \wedge \mathbf{N?}(ze) \equiv \mathbf{T}] \wedge (\exists\, ze)[\sigma\_\mathbf{p}(X, ze) \equiv \mathbf{T} \wedge \mathbf{N?}(ze) \equiv \mathbf{F}])$.

From the formulas in the axioms component of S, the nonlogical axioms are

derived as follows: We apply **TR** on a restricted formula to replace nondeterministic operation symbol by the corresponding auxiliary functions. Since the restricted formula expresses the normal behavior of the operations, the new variables introduced in the transformation range only on normal values. So, we use variables of a single type instead of the union type. For instance, for an equation '$e_1 \equiv e_2$' having nondeterministic operations on both side, we get

$(\forall z_1,..., z_m)\,[\, c_1 \Rightarrow (\exists y_1, \ldots, y_p)\,[\, c_2 \wedge e_1' \equiv e_2' \,]\,] \wedge$
$(\forall y_1, \ldots, y_p)\,[\, c_2 \Rightarrow (\exists z_1, \ldots, z_m)\,[\, c_1 \wedge e_1' \equiv e_2' \,]\,].$

To get the corresponding unrestricted formula incorporating the exceptional behavior of the operations and the preconditions, we must require that

(i) '$N?_D(e_1') \equiv T$' and '$N?_D(e_2') \equiv T$' hold, and

(ii) every operation invocation in the formula must satisfy the associated precondition.

The unrestricted formula for the above restricted formula is

$(\forall z_1,..., z_m)\,[\, N?_D(e_1') \Rightarrow (PC_{c_1} \Rightarrow (c_1 \Rightarrow$
$(\exists y_1, \ldots, y_p)\,[\, N?_D(e_2') \Rightarrow ((PC_{c_2} \wedge PC_{e_1'} \wedge PC_{e_2'}) \Rightarrow (\, c_2 \wedge e_1' \equiv e_2' \,))\,])\,]\,] \wedge$
$(\forall y_1,..., y_p)\,[\, N?_D(e_2') \Rightarrow (PC_{c_2} \Rightarrow (c_2 \Rightarrow$
$(\exists z_1, \ldots, z_m)\,[\, N?_D(e_1') \Rightarrow ((PC_{c_1} \wedge PC_{e_2'} \wedge PC_{e_1'}) \Rightarrow (\, c_1 \wedge e_1' \equiv e_2' \,))\,]\,))\,].$

A similar transformation can be obtained for a restricted formula of the form '$e_1 \equiv$ **if** b **then** e_2.'

For example, the formula

Choose(s) $\in$ **s** $\equiv$ **T**

in the specification of **Set-Int** is transformed first to the restricted formula using **TR**,

((∀ i) [Choose_p(s, i) ≡ T ⇒ i ∈ s ≡ T] ∧ (∃ i) [Choose_p(s, i) ≡ T]),

and later to

((∀ i) [N?$_{Bool}$(i ∈ s) ≡ T ⇒ (Choose_p(s, i) ≡ T ⇒ (N?$_{Bool}$(T) ≡ T ⇒ i ∈ s ≡ T))]
∧ (∃ i) [Choose_p(s, i) ≡ T]),

which gets simplified to

((∀ i) [Choose_p(s, i) ≡ T ⇒ i ∈ s ≡ T] ∧ (∃ i) [Choose_p(s, i) ≡ T]),

because '**N?$_{Bool}$**(i ∈ s) ≡ T' and '**N?$_{Bool}$**(T) ≡ T' are derivable.

Figure 4.14 is yet another implementation of **union** using the nondeterministic

operation **Choose** which signals on the empty set. This version is similar to the version given in Figure 4.11 except that **Choose** is nondeterministic. It can also be verified using the properties in **Th(Set-Int)**.

Figure 4.14. Procedure Union - IV

```
union = proc(s1, s2 : Set-Int) returns (Set-Int)
        i : Int
        r1 : Set-Int := s1
        r2 : Set-Int := s2
          { r1 ≡ s1 ∧ r2 ≡ s2 }
        while true do
            { (Size(r1) = 0 ≡ F ∧ Choose_p(r1, i) ≡ T ∧ IN(Remove(r1, i), Insert(r2, i), s1, s2))
              ∨ (Size(r1) = 0 ≡ T ∧ R_r2^union(s1, s2)) }
          i := Set-Int$Choose(r1)
            { IN(Remove(r1, i), Insert(r2, i), s1, s2) }
          r1 := Set-Int$Remove(r1, i)
          r2 := Set-Int$Insert(r2, i)
            { IN(r1, r2, s1, s2) }
        end except when no-element :
                    end
        { R_r2^union(s1, s2) }
        return (r2)
        { R }
        end union
```

$$IN(r1, r2, s1, s2) = ((\forall j)[(Has(s1, j) \vee Has(s2, j)) \Leftrightarrow (Has(r1, j) \vee Has(r2, j)) \equiv T] \wedge (Size(r1) + Size(r2)) \leq (Size(s1) + Size(s2)) \equiv T \wedge Size(r2) > 0 \equiv T)$$

I/O Specification for **union**

$T \Rightarrow R$, where $R = R1 \wedge R2$, and

$R1 = (\forall i)[(Has(s1, i) \vee Has(s2, i)) \Leftrightarrow Has(union(s1, s2), i) \equiv T]$

$R2 = Size(union(s1, s2)) \leq Size(s1) + Size(s2) \equiv T$

4.4.4 Properties of a Specification

We can prove theorems analogous to Theorems 4.10 and 4.11 for specifications specifying nondeterministic operations and exceptional behavior, demonstrating the soundness of the axioms capturing general properties of data types.

The definition of sufficient completeness property has to be modified significantly, because there is no meaningful definition of the equational subtheory for such specifications. Because of the semantics of S as defined in Section 3.2, it does not help to consider only the formulas involving deterministic operations and the auxiliary functions corresponding to nondeterministic operation symbols. Recall that for a behaviorally complete specification, for every input X to a nondeterministic operation, the corresponding auxiliary function is required to hold for at least one (X, ze), where ze is a possible result returned by σ on X, and the axioms do not precisely specify the values on which the auxiliary function holds. This incompleteness is because the semantics of S does not constrain an operation specified to be nondeterministic to have any fixed amount of nondeterminism (see Section 3.2).

A plausible modification to the definition of sufficient completeness is to require it to use the whole machinery of first order predicate calculus for deduction. Instead of requiring a theorem to be in EQ(S), we require it to be in Th(S). In addition, the definition of sufficient completeness given in Subsection 4.3.6 must also be modified to deal with the case when a legal ground term e involves nondeterministic operation symbols. For e of type $D' \in \Delta$, if '$N?_{D'}(e) \equiv T'$ $\in$ Th(S), it cannot usually be proved equivalent to a ground term of type D' having no operation symbol of D, as in case of **Choose(Insert(Insert(Null, 1), 2))** for example. Instead we must prove that there exists a set of ground terms $\{ e_1, \ldots, e_k \}$ of type D' not having any operation symbol of D such that

$$(\exists z_1, \ldots, z_m)[\, c \wedge (e' \equiv e_1 \vee e' \equiv e_2 \vee \ldots \vee e' \equiv e_k)\,],$$

where c is the condition on $z_1, \ldots, z_m$ generated due to e when we apply the procedure **TR**, and e' is the term obtained from e by substituting $z_1, \ldots, z_m$ for the subterms having nondeterministic operation symbols as their outermost operation. $\{ e_1, \ldots, e_k \}$ consists of all possible outcomes of e. (Since it is assumed that '$N?_{D'}(e) \equiv T'$ $\in$ Th(S), $z_1, \ldots, z_m$ are of a single type instead of a union type.) For example, in case of

Choose(Insert(Insert(Null, 1), 2)), we can show that

$(\exists$ i) [Choose_p(Insert(Insert(Null, 1), 2), i) $\equiv$ T $\wedge$ (i $\equiv$ 1 $\vee$ i $\equiv$ 2)]

We have not investigated the relationship between the above definition of sufficient completeness and the behavioral completeness property for such specifications. We conjecture that most of the results (Theorems 4.12, and 4.13 in particular) of Subsection 4.3.6, when appropriately modified, would hold for such specifications also.

The definition of well definedness given in Subsection 4.2.6 directly extends to this case also. The definition of completeness, like the definition of sufficient completeness, must require in this case that for any two legal ground terms e_1 and e_2 of the same type, '$e_1 \equiv e_2$' $\in$ Th(S) if and only if e_1 and e_2 are observably equivalent. The definition 4.8 of well definedness given in Subsection 4.2.6 is valid in this case also.

4.5 Strong Equivalence of Specifications

In Subsection 3.2.6, we defined the equivalence on specifications; the definition required two equivalent specifications to have the same semantics. As discussed in Subsection 4.2.6, two equivalent specifications can be different in what properties of a data type (a set of data types) can be deduced from them. Below, we define a stronger equivalence relation on specifications, which not only requires that the two specifications have the same semantics, but also that the same properties can be deduced from the specifications.

Def. 4.10 Two specifications S_1 and S_2 are *strongly equivalent* if and only if assuming that for every type used in S_1 and S_2, we use the same theory,

(i) S_1 and S_2 are equivalent, i.e., $\mathcal{D}(S_1) = \mathcal{D}(S_2)$, and

(ii) $Th(S_1)|_{L(D)} = Th(S_2)|_{L(D)}$. ∎

If S_1 (or S_2) specifies a nondeterministic operation σ, we assume that $L(D)$ includes the corresponding auxiliary function σ_p in place of σ.

5. Correctness of Implementation

One of the main purposes of designing a specification of a data type is to have a standard that can be used to verify whether an alleged implementation of the data type is correct. In this chapter, we propose a correctness criterion for an implementation of a data type with respect to its specification, and discuss a method embodying the proposed correctness criterion. In this process, we also exhibit how the theory of a data type discussed in the previous chapter is used.

An implementation of a data type **D** is concerned with how to realize the behavior of **D**, in contrast to its specification where the main concern is to precisely state its behavior. Intuitively speaking, our correctness criterion is that a correct implementation with respect to a specification must have the same observable behavior as prescribed by the specification.

Our approach for proving correctness of an implementation is similar to that of Hoare [37], Zilles [76] and Guttag et al. [29], and is radically different from the ADJ group's approach [23]. We separate the correctness method from the semantics of the host programming language in which an implementation is coded. We do not wish to concern ourselves with the issue of semantics of the control structures in the programming language, so we assume that the semantics of the procedures implementing the operations of **D** is already derived from their code. In contrast, the ADJ group does not seem to separate the correctness method from the semantics of the host programming language. It seems to be incorporating the semantics of the control structures used in implementing the operations into the correctness method, for instance, see their definition of *deriver*, which is a morphism from the specification algebra to the implementation algebra [23]. This makes its approach complex and restrictive.

An implementation uses data types abstractly; it does not refer to any particular implementation of a data type used in it. A recursive implementation of a data type **D** is an exception because a reference to **D** in the recursive implementation is interpreted as the reference to the implementation itself. We discuss recursive implementations later in the chapter; until then, we assume that an implementation of a data type does not use the data

type itself. For the time being, we also rule out mutually recursive implementations of a collection of (recursive or non-recursive) data types in which an implementation I of a data type D uses a data type D′ and an implementation I′ of D′ uses D. We discuss mutually recursive implementations later with recursive implementations.

While deriving the semantics of the procedures implementing the operations of D in an implementation I, we do not use the semantics of any particular implementation of a data type D′ used in I. We instead use the theory constructed from the specification S′ of D′, abstracting from all correct implementations of D′ with respect to S′. The proof of correctness of an implementation of D thus does not depend on any property specific of a particular implementation of D′. It remains valid even when an implementation of D′ is modified or replaced, as long as the new implementation of D′ is correct with respect to the specification of D′. This separation of the proof of use from the proof of implementation hierarchically structures the correctness proof, reducing the complexity of the verification process [37].

In the first section, we discuss the correctness criterion and present an overview of different steps in the correctness method. In the second section, we discuss the implementation structure and the semantics of an implementation. In the third section, we describe in detail the method for proving correctness of an implementation with respect to a specification. In the fourth section, we discuss extensions to the proposed method for proving correctness of recursive and mutually recursive implementations.

5.1 Correctness Criterion and Overview of Correctness Method

As discussed in Chapter 3, a specification S in general specifies a set $D(S)$ of related data types, because the behavior of some of the operations is intentionally left unspecified on certain inputs. In an implementation, the behavior of the procedures implementing these operations must be defined on all inputs in their domains, because an implementation in most programming languages realizes a single data type.[1] The designer of an implementation must pick one data type from the set $D(S)$ of data types.

If a specification specifies preconditions for the operations, the designer of an implementation has the freedom to decide what the procedure implementing such an operation should do on an input not satisfying its precondition. This is because in defining the semantics of a specification, it is assumed to be the user's responsibility to ensure that the input to the procedure satisfies the specified precondition. If a precondition is specified for constructor, the procedure implementing the constructor could either signal or return a value of the defined type. However, the value returned must be constructible by a procedure implementing a constructor using inputs satisfying its precondition (see discussion on p. 89. for the elaboration of this assumption). If a precondition is specified for an observer, the procedure implementing the observer could return a value of its range type, or signal. For example, the operations **Pop** and **Replace** of **Stk-Int** are specified to have '~ (Empty(s) = **0**)' as the precondition. An implementation of **Stk-Int** could have, for example, the procedure implementing the constructor **Pop** either signal on an empty stack or return an arbitrary **stack**.

For an operation specified to optionally signal exceptions, if the input to the procedure implementing the operation satisfies the associated condition, the designer has a choice between signalling the specified exception and returning a normal result that satisfies the axioms. For example, if optional exceptions are used to specify the size requirement on the values of a data type, as in case of **Stk-Int**, an implementation must decide the maximum size of the values. The procedure implementing the constructor **Push**

---

1. We are not considering parameterized implementations.

in an implementation of **Stk-Int** could either signal **overflow** or return a stack constructed by pushing the integer argument on the stack argument.

If a specification specifies nondeterministic operations, the requirement that an implementation of a nondeterministic operation must have maximum amount of nondeterminism specified by the specification is too strong. (In case of the specification of **Set-Int** given in Figure 3.1, such a requirement would mean that the procedure implementing the operation **Choose** must be able to nondeterministically pick any element of the set.) It is more appropriate to leave it to the designer of an implementation to decide how much nondeterminism a procedure implementing a nondeterministic operation should have: The procedure when viewed on 'abstract' values of the data type could be either deterministic, returning a fixed result out of the many possible choices specified by the specification for an input, or it could exhibit limited nondeterminism or maximum amount of nondeterminism specified by S, returning a subset of the set of possible results specified. For example, a correct implementation with respect to the specification of **Set-Int** can have the procedure implementing the operation **Choose** return the maximum integer in the set, say, or it could have the procedure nondeterministically pick between the minimum and maximum integers in the set, etc. As is discussed later, a deterministic procedure can also simulate nondeterministic behavior on 'abstract' values by returning different values on different values of the rep representing the same 'abstract' value of D. We call such a procedure *pseudo-nondeterministic.*

5.1.1 Semantics of an Implementation

By a procedure, henceforth, we mean a procedure in an implementation I of D implementing an operation of D, unless stated otherwise; by a constructor procedure and an observer procedure, we mean a procedure implementing a constructor and a procedure implementing an observer, respectively. We use the name of an operation of D in S written in capital letters, as the name of the procedure implementing the operation in I. Outside I, we use an operation name instead of the name of the procedure implementing the operation to signify that the data type is being used abstractly.

As data types are used abstractly in an implementation, the semantics of an

implementation I is a set of *implementation* algebras. These algebras can be constructed hierarchically as in Chapter 2; we use in the construction, the implementation algebras serving as the semantics of the implementations of the data types being used in I. Like a type algebra, an implementation algebra has a domain corresponding to every defining type $D' \in \Delta$, which is defined by an implementation algebra of an implementation I' of D'.

The domain corresponding to D is in general a subset of a domain corresponding to the rep defined by an implementation algebra of an implementation I_{rep} of the rep. It consists of the values of the rep used to represent the values of D. The subset is characterized by a formula **Inv**(*r*) with exactly one free variable *r* of the rep type. The formula **Inv**(*r*) represents the strongest unary relation on the values of the rep preserved by the constructor procedures in I. It captures the minimality property of the implementation, namely that a value of the rep that represents a value of D can be constructed by finitely many applications of the constructor procedures and that these values constitute the smallest subset closed under the constructor procedures.

Let $F^i(I)$ stand for the semantics of I. This set can be defined inductively. We assume that a set of primitive data types supported by the host programming language are implemented correctly with respect to their specifications by its compiler. The semantics of the specifications of such primitive types serves as the basis step of the inductive definition. If one wishes to prove the correctness of the implementation of a primitive type, the primitive type of the language in which the compiler is coded would then serve as the basis.

In the inductive step, an implementation algebra **A** in $F^i(I)$ has the following structure:

$$\mathbf{A} = [\{ V_D^i \} \cup \{ V_{D'} \mid D' \in \Delta \}, EXV; \{ i_\sigma \mid \sigma \in \Omega \}].$$

$V_D^i = \{ v \mid v \in V_{rep}^i \wedge \mathbf{Inv}(v) \}$, where V_{rep}^i is defined by an implementation algebra in $F^i(I_{rep})$ for an implementation I_{rep} of the rep. For each $D' \in \Delta$, $V_{D'}$ is defined by an algebra in $F^i(I_{D'})$ for an implementation $I_{D'}$ of D'. The specification of the procedure implementing σ is an abstract specification of i_σ.

In the next section, we discuss how to construct $F^i(I)$ after the discussion about the implementation structure and about **Inv**(*r*).

5.1.2 Correctness Method

If we consider specifications not specifying any nondeterministic operations, then the correctness criterion is simple: $F^i(I) \subseteq F(S)$. So, to prove the correctness of an implementation I, we need to show that every implementation algebra in $F^i(I)$ is also in $F(S)$, which can be done using the method discussed in Section 3.2 to show whether a type algebra is in $F(S)$. Two main steps of this method are:

(i) Construct the observable equivalence relation on V_D^i, as discussed in Sections 2.2 and 2.3, using the observable equivalence relation on $V_{D'}$ corresponding to each defining type $D' \in \Delta$ and the observable equivalence relation on V_{rep}, and

(ii) interpret the axioms and restrictions in the algebra, and show that they are satisfied.

Since the set of observable equivalence relations is a congruence, the observable equivalence relations must be preserved by the procedures. The observable equivalence relation is the largest such congruence on the algebra.

The above discussion is the formal basis of the correctness method proposed by Guttag et al. [29] and Kapur [40]. The observable equivalence relation on the domain corresponding to D is Guttag et al.'s equality interpretation. The above method in fact extends the methods in [29] and [40] because it can handle procedures signalling exceptions as well as nondeterministic procedures implementing deterministic operations. [2]

Note that if there exists a correct implementation I of S, then S is consistent, because then $F(S)$ is not empty. This is the basis of Guttag and Horning's statement [28] that one way of showing consistency of S is to design a correct implementation I of S.

---

2. A nondeterministic procedure can implement a deterministic operation if all possible results of the procedure on every input are observably equivalent.

5.1.2.1 Nondeterminism

For a specification S specifying nondeterministic operations, the criterion that $F^i(I) \subseteq F(S)$ is too strong as it rules out implementations with pseudo-nondeterministic procedures which ought to be correct. In such an implementation, a nondeterministic operation is implemented either as a deterministic procedure or as a nondeterministic procedure that does not preserve what should be the observable equivalence relation on the values of the rep. It returns different values when applied on different rep values representing the same 'abstract' value of D, but every value returned is a possible result specified by S on the input; nondeterministic behavior of an operation is realized in this way. If we take the largest equivalence relation on the rep values that is preserved by the procedures as the interpretation of $\equiv$ in the implementation (which is so in case of specifications not specifying nondeterministic operations), the axioms and restrictions in S may not hold for such an implementation. However if an equivalence relation preserved only by the procedures implementing deterministic operations is taken as the observable equivalence relation, then the axioms and restrictions hold in S.

Consider for example, the implementation of **Set-Int** in a CLU-like language given in Figure 5.1. The procedure **CHOOSE** is deterministic and returns the first element of the sequence value used to represent the set argument. The largest equivalence relation on the sequences preserved by all the procedures is the identity relation, and it can be shown that the axioms of the specification of **Set-Int** do not hold if the identity relation is taken as the observable equivalence relation. However if we take the relation

$\mathbf{Eqv(s1, s2) = (SI\$Size(s1) \equiv SI\$Size(s2)) \wedge (\forall i)[IN(s1, i) \Leftrightarrow IN(s2, i)]}$, where

$\mathbf{IN(s, i) = (\exists j)[1 \leq j \leq SI\$Size(s) \wedge SI\$Fetch(s, j) \equiv i]}$,

and SI stands for the data type *Sequence of Integers*, as the observable equivalence relation, then the axioms hold. The procedure **CHOOSE** returns 1, for example, on the sequence **Addh(Addh(New, 1), 2)** and 2 on **Addh(Addh(New, 2), 1)**, so **CHOOSE** behaves differently on members of the same equivalence class of sequences representing the same set value. **CHOOSE** is an example of a pseudo-nondeterministic procedure.

To fully illustrate the correctness method, we discuss two variations of the implementation in Figure 5.1 differing in the implementations of **Choose**. In the first,

Figure 5.1. An Implementation of Set-Int

```
SET-INT = cluster is NULL, INSERT, REMOVE, HAS, SIZE, CHOOSE

        rep = SEQUENCE-INT

NULL = proc() returns (cvt)
        return (rep$New())
        end NULL

INSERT = proc(s: cvt, i: Int) returns (cvt)
        if INDEX(s, i) ≤ rep$Size(s) then return (s) end
        return (rep$Addh(s, i))
        end INSERT

REMOVE = proc(s: cvt, i: Int) returns (cvt)
        j: Int := INDEX(s, i)
        if j ≤ rep$Size(s) then return (rep$Remh(rep$Replace(s, j, rep$Top(s))) ) end
        return (s)
        end REMOVE

HAS = proc(s: cvt, i: Int) returns (Bool)
        return (INDEX(s, i) ≤ rep$Size(s))
        end HAS

SIZE = proc(s: cvt) returns (Int)
        return (rep$Size(s))
        end SIZE

CHOOSE = proc(s: cvt) returns (Int) signals (no-element)
        if rep$Size(s) = 0 then signal no-element end
        return (rep$Bottom(s))
        end CHOOSE

INDEX = proc(s: cvt, i: Int) returns (Int)
        c: Int := 1
        while c ≤ rep$Size(s) do
          if rep$Fetch(s, c) = i then return (c) end
          c := c+1
        end
        return (c)
        end INDEX
```

Choose is implemented as a deterministic procedure **CHOOSE'** which returns the maximum integer in the nonempty sequence; the procedure **CHOOSE'** is given in

Figure 5.2. In the second, **Choose** is implemented as a nondeterministic procedure CHOOSE" which returns the maximum or minimum integer in the nonempty sequence. CHOOSE" is given in Figure 5.3. The construct *Select* in the code of CHOOSE" behaves nondeterministically: **Select(S1, S2, ..., Sn)**, where Si is a statement, arbitrarily picks one of the statements given as its arguments for execution. Note that neither of CHOOSE' and CHOOSE" is pseudo-nondeterministic.

Figure 5.2. CHOOSE'

```
CHOOSE' = proc(s: cvt) returns (Int) signals (no-element)
          if rep$Size(s) = 0 then signal no-element end
          return (MAX(s))
          end CHOOSE'

MAX = proc(s: rep) returns (Int)
          m: = rep$Bottom(s)
          for i: = 2 to rep$Size(s) do
           if m < rep$Fetch(s, i) then m : = rep$Fetch(s, i) end
          end
          return (m)
          end MAX
```

Figure 5.3. CHOOSE"

```
CHOOSE" = proc(s: cvt) returns (Int) signals (no-element)
          if rep$Size(s) = 0 then signal no-element end
          Select(return (MAX(s)), return (MIN(s)))
          end CHOOSE"

MIN = proc(s: rep) returns (Int)
          m: = rep$Bottom(s)
          for i: = 2 to rep$Size(s) do
           if m > rep$Fetch(s, i) then m : = rep$Fetch(s, i) end
          end
          return (m)
          end MIN
```

5.1.2.2 Definition of Correctness

We can now state the correctness criterion. It has two parts. The first part deals with implementations not having pseudo-nondeterministic procedures, and the second part takes care of pseudo-nondeterministic procedures. In the second part, the equivalence relation used on the rep is not required to be preserved by the procedures implementing nondeterministic operations thus allowing them to be pseudo-nondeterministic; the equivalence relation is only required to be preserved by the procedures implementing deterministic operations.

Def. 5.1 An implementation I is *correct with respect to* a specification S if and only if assuming that every data type D′ used in I has a correct implementation I′ with respect to its specification S′,

(i) $F^i(I) \subseteq F(S)$, or

(ii) for every algebra $A \in F^i(I)$, there is a set of equivalence relations, $E = \{ E_{D'} \mid D' \in \Delta \cup \{D\} \} \cup E_{EXV}$, such that

(a) for every defining type $D' \in \Delta$, $E_{D'}$ is the equivalence relation on $V_{D'}$ used to prove correctness of the implementation $I_{D'}$ of D′, and similarly, E_{rep} is the equivalence relation on V_{rep} used to prove correctness of an implementation I_{rep} of the rep,

(b) E_{EXV} is the equivalence relation defined as follows: For an exception name *ex* of arity $D_1 \times ... \times D_n$, if $\langle v_1, v_1' \rangle \in E_{D_1}, ..., \langle v_n, v_n' \rangle \in E_{D_n}$, then $\langle ex(v_1,...,v_n), ex(v_1',...,v_n') \rangle \in E_{EXV}$,

(c) $E_{rep} \subseteq E_D$,

(d) E is preserved by the functions corresponding to deterministic operations in A, and

(e) $A/E \in F(S)$. ∎

A/E is the quotient algebra of A induced by E except that E need not be a congruence; the function f_σ' in A/E corresponding to f_σ in A that does not preserve E behaves nondeterministically. The formal characterization above is complex because an implementation of a defining type or the rep could also have pseudo-nondeterministic procedures.

In the correctness method, we do not explicitly construct the set $F^i(I)$ of implementation algebras defined by I. We reason about the set as a whole by not using any

property specific to any particular implementation of $\mathbf{D}' \in \Delta$ or of the rep, and by instead using the procedure specifications and the theories of the defining types and the rep. We show that the axioms and restrictions of S hold when interpreted in I by deriving them from the procedure specifications.

Roughly speaking, the following steps need to be carried out to show correctness of an implementation:

(i) Derive the specification of every procedure in the implementation as a function on rep values from its code.

(ii) Design a formula **Inv**(r) characterizing the subset of the rep values needed to represent the values of D. It must express the strongest unary relation preserved by the constructor procedures.

(iii) Design the equivalence relation on the values of the rep satisfying **Inv**. The equivalence relation must be preserved by the procedures implementing the deterministic operations.

(iv) Interpret the restrictions and axioms using the procedures in place of the operations. Replace for a variable of type D, a variable of the rep type satisfying **Inv**. Interpret $\equiv$ corresponding to D as the equivalence relation of step (iii).

We discuss each of these steps in detail in the next two sections: The second section discusses the first two steps; the remaining steps and the correctness method are illustrated in the third section. We argue that a formula weaker than **Inv** often suffices; furthermore, the equivalence relation needed in step (iv) is also often weaker than the strongest equivalence relation preserved by the procedures implementing the deterministic procedures. We also discuss what extra steps need to be performed if auxiliary functions are used in a specification.

For recursive and mutually recursive implementations, there is an additional step in the correctness proof. We need to show that the rep (reps in case of mutually recursive implementations) defined by a recursive domain equation(s) is nonempty. The rest of the proof is the same as in case of nonrecursive implementations.

5.2 Implementation Structure and Semantics

Besides the procedures implementing the operations of **D**, an implementation **I** of **D** may include helping procedures needed in writing the procedures implementing the operations. For example, **INDEX** is a helping procedure in the implementation of **Set-Int** given in Figure 5.1. A helping procedure is not available outside the implementation, so we call it an *internal* procedure of **I**. Let I_p stand for the set of all internal procedures used in **I**. The procedures in **I** may also use types other than the rep and the defining types of **D**, if need be; we call such types *internal* types of **I** and denote the set of internal types in **I** as I_t. Note that the internal procedures and internal types of an implementation **I** are different from the auxiliary functions and auxiliary types used in its specification **S**.

In this thesis, we do not wish to be concerned about the semantics of the control structures used in coding the procedures. There are at least two approaches to avoid considering the control structures, which are discussed below. However, we illustrate the correctness method using only the translational approach. We have worked the correctness proofs using the other approach; the proofs in that case are similar in flavor to the proofs using the translational approach. These proofs are not presented in the thesis. We believe that the correctness method would work using any approach for specifying the procedures.

Most programming languages supporting user defined data types provide a mechanism that encapsulates a collection of procedures implementing the operations of a data type and provides an abstract view of data outside the mechanism, for example, cluster in CLU, form in ALPHARD, etc. The encapsulation mechanism constrains the use of the procedures. We discuss below the properties desired of an encapsulation mechanism that facilitate the correctness proof of an implementation. Finally, we discuss how we get the semantics of an implementation **I** as a set $F^i(I)$ of implementation algebras to complete the formal aspects of the correctness method.

5.2.1 Procedures - Approach I

In Chapter 4, we discussed a method based on Floyd-Hoare approach for specifying a procedure. In this method, a procedure is specified as a set of formulas relating its input to the result(s) returned by it. The procedures implementing the operations in an implementation I can be specified in this way; the specifications of internal procedures are not included if they are not referred in the specifications of the procedures implementing the operations. A procedure is specified as a transformation on the values of the rep. To verify the correctness of a procedure with respect to its specification, the theories of the defining types, the rep, and the internal types are used.

Figure 5.4 is the specification of the procedures in the implementation of **Set-Int** given in Figure 5.1 using this method. It also has specifications of **CHOOSE'** and **CHOOSE"**. Instead of using the procedure invocation itself to stand for the result (or a possible result in case of a nondeterministic procedure), we have introduced, for convenience, a name for the result. For example, the specification of the procedure **REMOVE** uses r to stand for the result of **REMOVE** on inputs s and i. The specification captures that

(i) if the integer argument i is in the sequence argument s, then r is the sequence obtained by first replacing the first occurrence of i in s by the topmost element in the sequence and then getting rid of the topmost element; otherwise,

(ii) r is s itself. In deriving these specifications, we have used the specification of the data type **Sequence-Int** given in Appendix IV.

5.2.2 Procedures - Approach II

We translate a procedure implemented in a rich imperative programming language to a simple applicative language similar to the specification language proposed in Chapter 3 using the method suggested by McCarthy [56] (see [54] where the method is well explained). Use the translated procedures to prove the correctness of the implementation I. Guttag et al. [29] and Kapur [40] take this approach; they use a language supporting conditional expressions, composition, recursion, and the use of auxiliary functions.

Figure 5.4. Specification of the Procedures in the Implementation of Set-Int Using Approach I

```
NULL() : ( = r)
  r ≡ rep$New()

INSERT(s, i) : ( = r)
 ( In'(s, i) ⇒ r ≡ s ) ∧ ( ~ In'(s, i) ⇒ r ≡ rep$Addh(s, i) )

REMOVE(s, i) : ( = r)
 (∃ j) [ i ≡ s[j] ∧ (∀ j') [ j' < j ⇒ ~ i ≡ s[j'] ] ∧
          r ≡ rep$Remh(rep$Replace(s, j, rep$Top(s))) ] ∨ (~ In'(s, i) ⇒ r ≡ s)

HAS(s, i) : ( = b)
 (b≡ T) ⇔ In'(s, i)

SIZE(s) : ( = i)
 i ≡ rep$Size(s)

CHOOSE(s) : ( = i)
  rep$Size(s) = 0 ⇒ CHOOSE(s) signals no-element()
  rep$Size(s) > 0 ⇒ i ≡ s[1]


CHOOSE'(s) : ( = i)
  rep$Size(s) = 0 ⇒ CHOOSE'(s) signals no-element()
  rep$Size(s) > 0 ⇒ ( In(s, i) ∧ (∀ j) [ 1 ≤ j ≤ rep$Size(s) ⇒ s[j] ≤ i ] )

CHOOSE''(s) : ( = i)
  rep$Size(s) = 0 ⇒ CHOOSE(s) signals no-element()
  rep$Size(s) > 0 ⇒ ( In(s, i) ∧ ((∀ j) [ 1 ≤ j ≤ rep$Size(s) ⇒ s[j] ≤ i ]
          ∨ (∀ j) [ 1 ≤ j ≤ rep$Size(s) ⇒ i ≤ s[j] ]),

where  In(s, i) = (∃ j) [ 1 ≤ j ≤ rep$Size(s) ∧ s [j] ≡ i]
       In'(s, i) = (∃ j) [ 1 ≤ j ≤ rep$Size(s) ∧ s [j] ≡ i ∧ (∀ j') [ j' < j ⇒ ~ i ≡ s[j'] ] ]
```

We use an extended applicative language that has a *signal* primitive and *guarded expressions* in addition to composition and recursion mechanisms, and the use of auxiliary functions, so that the procedures signalling exceptions and exhibiting nondeterministic behavior can be specified. Conditional expressions can be simulated using guarded expressions. The translation method proposed by McCarthy can be extended to deal with the exception handling mechanism and the nondeterministic construct in a programming language.

An expression is similar to a term; it uses procedure names implementing the operations, internal procedure names, the auxiliary procedure names introduced during the

translation, and terms.

The *signal* primitive takes arbitrarily many (nonzero) arguments; its first argument is an exception name, and other arguments are expressions of various types. Its syntax is **signal**(ex, e_1, ..., e_n), where ex is an exception name with arity $D_1 \times \ldots \times D_n$ and each e_i is an expression of type D_i.

A *guarded expression* is similar to Dijkstra's guarded commands; its syntax is

⟨guarded expression⟩ ::= ⟨expression⟩ | ⟨alternative⟩ [|| ⟨alternative⟩]*

⟨alternative⟩ ::= ⟨condition⟩ ⇒ ⟨guarded expression⟩

⟨condition⟩ ::= ⟨boolean expression⟩,

where [X]* stands for zero or finitely many repetitions, and the symbol '||' stands for nondeterministic choice among various alternatives. If a guarded expression is simply an expression, then its semantics is that of an expression. Otherwise, if a guarded expression is a collection of alternatives, then for an instance of its variables, its semantics is the semantics of the guarded expression of an arbitrarily chosen alternative whose boolean condition is T. If every alternative has its condition as F, then the semantics of the guarded expression is undefined. A guarded expression exhibits nondeterministic behavior because for an instance of the variables, there are in general many alternatives whose condition is T, and one such alternative is arbitrarily chosen. [3]

We translate the procedures in the implementation of **Set·Int** in Figure 5.1 to the above applicative language. Figure 5.5 is their translation; we have also included the translation of the procedures **CHOOSE'** and **CHOOSE"** as well as of the internal procedures **MAX** and **MIN**. In translating the internal procedure **INDEX**, the auxiliary function f is introduced to simulate the effect of the *while* loop used in **INDEX**. Similarly,

---

3. An alternate approach to introducing guarded expressions for specifying the nondeterministic behavior of a procedure **OP** is to specify its non-exceptional behavior using a deterministic boolean auxiliary function **OP_P**, similar to the function σ_p corresponding to a nondeterministic operation σ as discussed in the previous chapter. For an input on which the nondeterministic procedure returns a normal value, the corresponding auxiliary function holds for all possible values returned by the procedure on that input and does not hold for other values. Then the procedures can be specified using conditional expressions and recursion. We have adopted the above approach for specifying the procedures, because it is direct and simple.

the auxiliary procedures f' and f" are introduced to simulate the *for* loop in MAX and MIN respectively.

Cartwright and McCarthy's first order semantics of recursive programs [8] can be used to prove properties about the procedures written in the above applicative language. The recursive definition of a procedure is considered as an axiom defining the function computed by the procedure. Because of the nondeterministic behavior of a guarded expression, we have to be careful in using such an axiom, or we will run into inconsistencies. For a particular instantiation of variables in the axiom, we use every possible alternative whose condition is T, and we do not relate any two alternatives whose conditions are T. For example, for CHOOSE", there are two alternatives, **MAX(s)** and **MIN(s)**, for the case (~ **rep$Size(s) = 0**). We do not equate **MAX(s)** to **MIN(s)**, as relating them can cause inconsistency. The termination of a procedures is proved separately either using the method suggested by Cartwright and McCarthy, or the method based on well founded ordering [14].

The translational approach is purely based on the semantics of the control structures of the host programming language in terms of the primitives of the applicative language incorporated into the translation method. The properties of the types involved in the implementation can be used in simplifying the resulting translations.

5.2.3 Properties of the Encapsulation Mechanism

As was stated earlier, in most of the programming languages supporting user defined data types, an implementation of a data type is an encapsulation of the procedures implementing the operations that disciplines their use. Such an implementation is protected: A procedure implementing an operation of **D** cannot be passed any arbitrary value of the rep as a representation of a value of **D**; rather only a value of the rep constructed earlier as a representation for a value of **D** by the constructor procedures of **D** can be passed. Every value of the rep need not in general be used to represent a value of **D**. The procedures are invoked only on those values of the rep which can be constructed by finitely many applications of the constructor procedures of **D**. (For example, the procedure **REMOVE** in the implementation of **Set-Int** in Figure 5.1 is never passed a sequence having

Figure 5.5. Translation of the Procedures in the Implementation of Set-Int

```
NULL () ≜ rep$New()

INSERT(s, i) ≜ INDEX(s, i) ≤ rep$Size(s) ⇒ s ‖
               (~ INDEX(s, i) ≤ rep$Size(s)) ⇒ rep$Addh(s, i)

REMOVE(s, i) ≜ INDEX(s, i) ≤ rep$Size(s) ⇒
                 rep$Remh(rep$Replace(s, INDEX(s, i), rep$Top(s))) ‖
               (~ INDEX(s, i) ≤ rep$Size(s)) ⇒ s

HAS(s, i) ≜ INDEX(s, i) ≤ rep$Size(s)

SIZE(s, i) ≜ rep$Size(s)

CHOOSE(s) ≜ rep$Size(s) = 0 ⇒ signal(no-element) ‖
            (~ rep$Size(s) = 0) ⇒ rep$Bottom(s)

INDEX(s, i) ≜ f(s, i, 1)

CHOOSE'(s) ≜ rep$Size(s) = 0 ⇒ signal(no-element) ‖
             (~ rep$Size(s) = 0) ⇒ MAX(s)

MAX(s) ≜ f'(s, rep$Bottom(s), 2)

CHOOSE''(s) ≜ rep$Size(s) = 0 ⇒ signal(no-element) ‖
              (~ rep$Size(s) = 0) ⇒ MAX(s) ‖
              (~ rep$Size(s) = 0) ⇒ MIN(s)

MIN(s) ≜ f''(s, rep$Bottom(s), 2)
```

Auxiliary Functions

```
f  : rep X Int X Int → Int
f' : rep X Int X Int → Int
f'' : rep X Int X Int → Int

f(s, i, c) ≜ (~ c ≤ rep$Size(s)) ⇒ c ‖
             (c ≤ rep$Size(s)) ∧ (rep$Fetch(s, c) = i) ⇒ c ‖
             (c ≤ rep$Size(s) ∧ ~ (rep$Fetch(s, c) = i) ⇒ f(s, i, c + 1)

f'(s, m, c) ≜ (~ c ≤ rep$Size(s)) ⇒ m ‖
              ((c ≤ rep$Size(s)) ∧ (m < rep$Fetch(s, c))) ⇒ f'(s, rep$Fetch(s, c), c + 1) ‖
              ((c ≤ rep$Size(s)) ∧ (~ m < rep$Fetch(s, c))) ⇒ f'(s, m, c + 1)

f''(s, m, c) ≜ (~ c ≤ rep$Size(s)) ⇒ m ‖
               ((c ≤ rep$Size(s)) ∧ (m > rep$Fetch(s, c))) ⇒ f''(s, rep$Fetch(s, c), c + 1) ‖
               ((c ≤ rep$Size(s)) ∧ (~ m > rep$Fetch(s, c))) ⇒ f''(s, m, c + 1)
```

multiple occurrences of an integer, as such a sequence cannot be constructed using **NULL**, **INSERT** and **REMOVE**.) We are interested in the behavior of the procedures only on this subset of the values of the rep. The subset is characterized by the formula **Inv**(r) discussed in the previous section, which expresses the strongest unary relation on the values of the rep *preserved* by the constructor procedures of **D**. **Inv**(r) is expressed without alluding to any particular implementation of the rep type.

Def. 5.2 A procedure **OP** implementing a constructor $\sigma : D_1 \times \ldots \times D_n \rightarrow D$ *preserves* **Inv** if and only if

whenever $((\forall\ 1 \leq i \leq n)\ [D_i = D \Rightarrow Inv[x_i]\ ])$, then

(i) if $OP(x_1, \ldots, x_n)$ returns a normal value, $Inv[OP(x_1, \ldots, x_n)]$; otherwise,

(ii) if $OP(x_1, \ldots, x_n)$ **signals** $ex(e_1, \ldots, e_k)$, then for each e_i of type **D**, $Inv[e_i]$.

If **OP** is nondeterministic, all possible results returned by **OP** must satisfy **Inv**. ∎

For the implementation of **Set-Int** given in Figure 5.1, **Inv**(s) is

$$(\forall\ i, j)\ [\ (1 \leq i, j \leq \text{rep\$Size(s)} \wedge i \neq j) \Rightarrow s[i] \neq s[j]\ ],$$

where s[i] is an abbreviation for **rep$Fetch(s, i)**. It can verified that **Inv**(s) is preserved by the constructor procedures of **Set-Int**. Figure 5.6 is a proof that **REMOVE** preserves **Inv**(s), the most difficult among the three cases. Any predicate stronger than the one above is not preserved by the constructor procedures.

Inv may be difficult to deduce from a complex implementation, but the designer of an implementation usually has a good idea about what **Inv** is. In the correctness proof, **Inv** is usually not necessary; a weaker property may suffice. In case **Inv** is available, a property of the representing values needed in the correctness proof can be deduced directly from **Inv**. Otherwise, if **Inv** is not available, then the property can be deduced by checking whether the property is preserved by the constructor procedures; since **Inv** is the strongest unary relation preserved by the constructor procedures, any unary relation preserved by the constructor procedures is implied by **Inv**.

If a module implementing an abstract data type in a programming language is not protected, as would be the case if abstract data types are simulated in PASCAL or PL/I, say, then

Figure 5.6. Proof of REMOVE Preserving Inv

Assume Inv(s) holds. To show that Inv(REMOVE(s, i)) holds.
If type name is not included in the operation names below, we assume that the operation are of type rep.
There are two cases.

Case 1: INDEX(s, i) $\leq$ Size(s)
Size(s) > 0 <= => T, from the specification of INDEX
Inv(REMOVE(s, i)) $\Leftrightarrow$ Inv(Remh(Replace(s, INDEX(s, i), Top(s)))), from the specification of REMOVE
It can be shown using the specification of INDEX and the theory of *Sequence-Int* that

(i) (Inv(s) $\wedge$ 0 < k $\leq$ Size(s) $\wedge$ s' $\equiv$ Replace(s, k, j)) $\Rightarrow$
((($\forall$ k1) [1 $\leq$ k1 $\leq$ Size(s) $\wedge$ $\sim$ k = k1] $\Rightarrow$ s'[k1] $\equiv$ s[k1]) $\wedge$ s[k] $\equiv$ j)

(ii) (Size(s) > 0 $\wedge$ s' $\equiv$ Remh(s)) $\Rightarrow$ ($\forall$ k) [(1 $\leq$ k $\leq$ Size(s')) $\Rightarrow$ (s'[k] $\equiv$ s[k] $\wedge$ Size(s') $\equiv$ Size(s) -1)]

Using (i) and (ii), we have Inv(REMOVE(s, i)) $\Leftrightarrow$ T

Case 2: $\sim$ INDEX(s, i) $\leq$ Size(s)
Inv(REMOVE(s, i)) $\Leftrightarrow$ Inv(s), from the specification of REMOVE
$\Leftrightarrow$ T

(i) restrictions must be imposed on the global variables, if any, as well as on the use of the procedures implementing the operations to ensure the minimality property of the implementation, and

(ii) Inv must be preserved wherever a procedure implementing an operation is invoked.

Such a proof is likely to be global and complex. (Guttag [31] discusses restrictions on the Euclid implementation module to ensure that the module satisfy the minimality property.) In the following discussion, we assume that the semantics of a mechanism encapsulating the procedures implementing the operations of a data type ensures the minimality property.

It is not necessary for the procedures to terminate over their entire input domain if Inv(r) is other than T. To prove total correctness of an implementation, it is sufficient that a procedure implementing an operation σ that has its i-th argument x_i to be of type D terminates whenever Inv[x_i] holds.

5.2.4 Semantics of an Implementation

Now that we have the procedure specifications, we can construct the implementation algebras of **I** using them. Since procedures specifications may use internal types and internal and auxiliary procedures, we first construct the extended implementation algebras and then derive the implementation algebras from them. For every possible implementation **I'** of a type **D'** used in the implementation **I**, we have the set of its implementation algebras. In an implementation algebra of **I**, the domain corresponding to **D'** is the domain defined by an implementation algebra of **I'**. An extended implementation algebra $\mathbf{A}^1$ of **I** has the following structure:

$\mathbf{A}^1 = [\{ V_D^i \} \cup \{ V_{D'} \mid D' \in \Delta \cup I_t \}, \mathbf{EXV}; \{ i_\sigma \mid \sigma \in \Omega \cup I_p \}]$.[4]

$V_D^i = \{ v \mid v \in V_{rep} \wedge \mathbf{Inv}(v) \}$. The function i_σ is the interpretation of the specification of the procedure corresponding to σ in $\mathbf{A}^1$. From $\mathbf{A}^1$, we get an implementation algebra **A**

$\mathbf{A} = [\{ V_D^i \} \cup \{ V_{D'} \mid D' \in \Delta \}, \mathbf{EXV}; \{ i_\sigma \mid \sigma \in \Omega \}]$.

---

4. In addition to the internal procedures, I_p is assumed to include the auxiliary procedures needed in the translation of the procedures into the applicative language discussed above.

5.3 Correctness Method

We describe the remaining steps of the correctness method outlined in Subsection 5.1.2. For completeness, we repeat the steps discussed in the previous section about the termination of the procedures and the preservation of the formula **Inv**. For a specification specifying nondeterministic operations, we discuss the method for three cases: An implementation of a nondeterministic operation is (i) a deterministic procedure, (ii) a nondeterministic procedure, and (iii) a pseudo-nondeterministic procedure. We first use the implementation of **Set-Int** given in Figure 5.1 with **CHOOSE** replaced by **CHOOSE'** for illustrating the method for the deterministic case. Later, we use **CHOOSE"** as the implementation of **Choose** to illustrate the method for the nondeterministic case, and finally, we use **CHOOSE** to illustrate the method for the pseudo-nondeterministic case.

5.3.1 Auxiliary Functions in a Specification

If a specification S uses auxiliary functions and auxiliary types, we extend an implementation I to include the implementations of the auxiliary functions in the correctness proof. We include in the specifications of the procedure of I, the specifications of the implementations of the auxiliary functions. For showing the correctness of I, we use the extended implementation, instead of I in the following steps; an auxiliary functions is treated like an operation. In the following discussion, whenever we say I, we mean the extended implementation if S uses auxiliary functions.

5.3.2 Preservation of Inv

If the formula **Inv**(r), which characterizes the subset of values of the rep used to represent the values of D, is available, verify that **Inv**(r) is preserved by every constructor procedure. We showed in the previous section that for the implementation of **Set-Int** in Figure 5.1, its **Inv** is preserved by every constructor procedure.

If **Inv**(r) is not available and cannot be guessed easily, we temporarily assume that every value of the rep is being used to represent the values of D. In the derivation of the axioms and restriction of S from the procedure specifications, in case we need any property

P(r) of the rep values, we deduce P(r) by showing that P(r) is preserved by the constructor procedures of D, as in that case **Inv**(r) would imply P(r).

In the derivation of an axiom or a restriction in S from the procedure specifications, a variable of type D is instantiated to a value of the rep satisfying **Inv**(r) (or P(r) if **Inv**(r) is not available).

5.3.3 Termination of Procedures

Prove that every procedure in I is total on the arguments it can expect, i.e., if an argument to a procedure is of type D, prove that the procedure terminates if these arguments are values of the rep satisfying **Inv**(r).

5.3.4 Proving Restrictions and Axioms

Show that every restriction in S specifying the exceptional behavior and every axiom in S specifying the normal behavior can be derived from the specifications of procedures in I. The operation symbols and the auxiliary function symbols in the axioms and restrictions are replaced by the names of procedures implementing them. The theories derived from the specifications of the defining types, the rep, and internal types can be used in the derivations.

The symbol $\equiv$ in S is interpreted as the observable equivalence relation. $\equiv_D$ is usually interpreted as the largest equivalence relation on the values of the rep satisfying **Inv**, *preserved* by the procedures. The exception is the case when a nondeterministic operation is implemented as a pseudo-nondeterministic procedure. Then, the observable equivalence relation serving as the interpretation of $\equiv_D$ is required to be preserved only by the procedures implementing deterministic operations, and it need not be the largest such equivalence relation.

5.3.4.1 Preservation of Equivalence Relation

A deterministic procedure **OP** implementing an operation σ : D_1 x...x $D_n \rightarrow D'$ preserves an equivalence relation on the rep values, expressed as a first order formula **Eqv**(s_1, s_2), where s_1 and s_2 are of rep type, and are the only free variables in the formula, if and only if for each $1 \leq i \leq n$, ([$D_i = D \Rightarrow$ **Eqv**(x_i, y_i)] $\wedge$ [$D_i \neq D \Rightarrow x_i \equiv y_i$]), either

(i) '**OP**(x_1,..., x_n) **signals** ext_1' holds and '**OP**(y_1,..., y_n) **signals** ext_2' holds such that '$ext_1 \equiv ext_2$' is provable. In addition to the rules discussed in the previous chapter, we have: For an exception name ex of arity D_1' x...x D_m', if for every $D_i' = D$, **Eqv**(x_i', y_i'), and for every $D_i' \neq D$, $x_i' \equiv y_i'$, then $ex(x_1',..., x_m') \equiv ex(y_1',..., y_m')$ is provable. Or,

(ii) If $D' = D$, then '**Eqv**(**OP**(x_1,..., x_n), **OP**(y_1,..., y_n))' is provable, and if $D' \neq D$ then '**OP**(x_1,..., x_n) $\equiv_{D'}$ **OP**(y_1,..., y_n)' is provable.

If **OP** is nondeterministic then (ii) above is modified to be: If $D' = D$, then for every possible result r_1 returned by **OP**(x_1,..., x_n), **OP**(y_1,..., y_n) can return r_2 such that **Eqv**(r_1, r_2) is provable, and vice versa, and if $D' \neq D$, for every r_1 returned by **OP**(x_1,..., x_n), **OP**(y_1,..., y_n) can return r_2 such that '$r_1 \equiv_{D'} r_2$' is provable and vice versa.

For example, **Eqv**(s1, s2) for the implementation of **Set-Int** in Figure 5.1 with **CHOOSE'** replacing **CHOOSE** is

(SI$Size(s1) ≡ SI$Size(s2)) ∧ (∀ i) [IN(s1, i) ≡ IN(s2, i)], where

IN(s, i) = (∃ j) [1 ≤ j ≤ SI$Size(s) ∧ s[j] ≡ i].

It relates sequences that are permutations of each other. **Eqv** is preserved by every procedure implementing an operation of **Set-Int**. Figure 5.7 has the proofs for the procedures **INSERT** and **HAS**. **Eqv**(s_1, s_2) is the largest equivalence relation preserved by the procedures. Any equivalence relation stronger than **Eqv** would have to relate sequences that are not permutations, and is thus not preserved by **HAS**.

Figure 5.7. Proofs that INSERT and HAS Preserve Eqv

For **INSERT**

assume Eqv(s1, s2), to show that (∀ i) Eqv(INSERT(s1, i), INSERT(s2, i))

Case 1: INDEX(s1, i) ≤ SI$Size(s1) ≡ T
Using Eqv(s1, s2), we have INDEX(s2, i) ≤ SI$Size(s2) ≡ T, so
INSERT(s1, i) ≡ s1, INSERT(s2, i) ≡ s2, so Eqv(INSERT(s1, i), INSERT(s2, i)) ⇔ T

Case 2: INDEX(s1, i) ≤ SI$Size(s2) ≡ F
Using Eqv(s1, s2), we have INDEX(s2, i) ≤ SI$Size(s2) ≡ F, so
INSERT(s1, i) ≡ Addh(s1, i), INSERT(s2, i) ≡ Addh(s2, i), so
Eqv(INSERT(s1, i), INSERT(s2, i)) ⇔ Eqv(Addh(s1, i), Addh(s2, i)) ⇔ T

For **HAS**

From the semantics of INDEX, we have
(i) INDEX(s, i) > 0 ≡ T,
(ii) INDEX(s, i) ≤ SI$Size(s) ⇒ s [INDEX(s, i)] ≡ i,
(iii) INDEX(s, i) > SI$Size(s) ⇒ [(∀ j) (1 ≤ j ≤ SI$Size(s)) ⇒ ~ s [j] ≡ i]

assume Eqv(s1, s2), to show (∀ i) HAS(s1, i) ≡ HAS(s2, i)
HAS(s1, i) ≡ INDEX(s1, i) ≤ SI$Size(s1)

Case 1: INDEX(s1, i) ≤ SI$Size(s1) ≡ T
s1 [INDEX(s1, i)] ≡ i
Using Eqv(s1, s2), we get (∃ j) [(1 ≤ j ≤ SI$Size(s2)) ∧ s2 [j] = i], so
INDEX(s2, i) ≤ SI$Size(s2) ≡ T
HAS(s1, i) ≡ HAS(s2, i) ≡ T

Case 2 INDEX(s1, i) ≤ SI$Size(s1) ≡ F
Using Eqv(s1, s2) and the above facts about INDEX, we get
INDEX(s2, i) ≤ SI$Size(s2) ≡ F, so
HAS(s1, i) ≡ HAS(s2, i) ≡ F

5.3.4.2 Restrictions

For a restriction specifying a required exception condition of σ,

$\mathbf{R_i}(X) \Rightarrow \sigma(X)$ **signals** *ext*

show that whenever $\mathbf{P_\sigma}(X)$ and $\mathbf{R_i}(X)$ interpreted in **I** hold, the procedure **OP** implementing σ must signal *ext*. For example, the specification of **Set-Int** specifies the following required exception condition for **Choose** in its restrictions component:

#(s) = 0 ⇒ Choose(x) signals no-element().

So the procedure **CHOOSE'** must signal **no-element()** when **SIZE(s) = 0**

($\Leftrightarrow$ **SI$SIZE(s) = 0**) holds, which is indeed so (the precondition specified for **Choose** is **T**).

For a restriction associating an optional exception condition with σ,

$\sigma(X)$ **signals** *ext* $\Rightarrow$ $\mathbf{O_j}(X)$,

show that whenever the procedure **OP** implementing σ signals *ext*, $\mathbf{P_\sigma}(X)$ and $\mathbf{O_j}(X)$ interpreted in **I** hold. For example, the specification of **Stk-Int** given in Figure 3.2 specifies the following optional exception condition for the operation **Push**:

Push(s, i) signals overflow(s, i) $\Rightarrow$ **#(s)** $\geq$ **100.**

In an implementation of **Stk-Int**, if the procedure implementing **Push** signals **overflow**, then the size of the input stack must be $\geq$ 100.

We must also show that (i) if an input to a procedure **OP** implementing an operation σ satisfies its precondition, does not satisfy the condition for any of its required exceptions or optional exceptions, then the procedure terminates normally. Let

$$C(X) = (\mathbf{P_\sigma}(X) \wedge (\sim \mathbf{R_1}(X) \wedge \ldots \wedge \sim \mathbf{R_l}(X)) \wedge (\sim \mathbf{O_1}(X) \wedge \ldots \wedge \sim \mathbf{O_m}(X))),$$

where for $1 \leq i \leq l$, $\mathbf{R_i}$ is the condition when σ is required to signal ext_i, and for $1 \leq j \leq m$, $\mathbf{O_j}$ is the condition when σ has the option to signal an exception ext_j. We show that $C(X)$ implies $\mathbf{TC_{normal}}(X)$, where $\mathbf{TC_{normal}}(X)$ is the weakest input condition for **OP** to terminate normally. For example, for every procedure in the implementations of **Set-Int**, the above condition is satisfied.

If a nontrivial precondition $\mathbf{P_\sigma}$ is specified for a constructor σ, then the procedure **OP** implementing σ either signals on input X not satisfying $\mathbf{P_\sigma}$, or returns a rep value which can be constructed by a constructor procedure using an input satisfying its precondition. For example, a correct implementation of **Stk-Int** can have the procedure implementing **Pop** return a stack when applied on an empty stack. If the procedure implementing **Push** signals **overflow** on a stack of size 128, say, then the procedure implementing **Pop** can only return any stack of size $\leq$ 128. It cannot return a stack of size 1000, say; allowing it to do so would be meaningless.

5.3.4.3 Axioms

In the derivation of an axiom, we ensure that (i) for every occurrence of a procedure name **OP** implementing the operation σ, the input to **OP** must satisfy the precondition P_σ associated with σ, and (ii) no subexpression signals any exception.

If an axiom is an equation of the form '$e_1 \equiv e_2$,' we prove that its interpretation in **I** is derivable. If e_1 and e_2 are of type **D**, $\equiv$ is interpreted as **Eqv**; otherwise, the interpretation of e1 $\equiv$ e2 in **I** can be derived using the theories constructed from the specifications of the rep, the defining types, and internal types.

If an axiom is of the form '$e_1 \equiv$ **if** b **then** e_2,' we have to prove that '$b \Rightarrow e_1 \equiv e_2$' when interpreted in **I** is derivable. Similarly, for an axiom '$e_1 \equiv$ **if** b **then** e_2 **else** e_3,' we must prove that '$b \Rightarrow e_1 \equiv e_2$' and '$\sim b \Rightarrow e_1 \equiv e_3$' are derivable in **I**. Recall that the condition b is assumed to behave deterministically even when it involves nondeterministic operation symbols. Figure 5.8 is a proof that the then part of the axiom,

Remove(Insert(s, i1), i2) ≡ if i1 = i2 Then Remove(s, i2) else Insert(Remove(s, i2), i1),

is derivable. The derivation of the else clause,

(~ i1 = i2) ⇒ Remove(Insert(s, i1), i2) ≡ Insert(Remove(s, i2), i1),

uses a property of the representing values that

(∀ i) [(rep$Size(s) > 0 ∧ In(s, i)) ⇒ (∃ ! j) [1 ≤ j ≤ rep$Size(s) ∧ s[j] ≡ i]],

---

Figure 5.8. Proof that an Axiom of Set-Int is Derivable

i1 = i2 ⇒ Remove(Insert(s, i1), i2) ≡ Remove(s, i2)

Assume i1 = i2, to show Eqv(REMOVE(INSERT(s, i1), i2), REMOVE(s, i2))

Case 1: INDEX(s, i1) ≤ rep$Size(s) ≡ T
  INSERT(s, i1) ≡ s, so the above holds.

Case 2: INDEX(s, i1) ≤ rep$Size(s) ≡ F
  Let r ≡ INSERT(s, i1) ≡ Addh(s, i1)
  Using i1 = i2, INDEX(r, i2) ≡ rep$Size(Addh(s, i1)), so
  REMOVE(r, i2) ≡ s, and
  REMOVE(s, i2) ≡ s, so the above holds.

which is preserved by the constructor procedures.[5]

The axiom 'Choose(s) ∈ s ≡ T' under the condition '~ Size(s) = 0,' when interpreted in I is 'HAS(CHOOSE'(s), s) ≡ T.' This is derivable, because 'INDEX(MAX(s), s) ≤ rep$Size(s) ≡ T' is derivable. The remaining axioms in the specification of Set-Int can also be shown to be derivable.

The above five steps constitute the correctness method. If an implementation I can go through the above steps, it is correct with respect to S. For example, the implementation of Set-Int given in Figure 5.1 with CHOOSE replaced by CHOOSE' goes through the above steps, and is thus correct.

5.3.5 Nondeterministic Procedures

We now consider the case when an implementation has a nondeterministic procedure implementing an operation specified to be nondeterministic by S. We have already discussed the conditions for a nondeterministic procedure to preserve Inv and the equivalence relation Eqv. Various steps in the correctness proof discussed above remain the same except that if an axiom involves the nondeterministic procedure, we must use the interpretation of formulas involving nondeterministic function symbols discussed in Chapter 4. In addition, it must be ensured that for any input, the nondeterministic procedure does not have a choice of signalling as well as terminating normally.

For example, if we consider the implementation of Set-Int in Figure 5.1 with CHOOSE replaced by CHOOSE", most of the above proof remains valid. We have to show that the axiom 'Choose(s) ∈ s ≡ T' is derivable under the condition '~ Size(s) = 0.' That is, if 'rep$Size(s) > 0' holds, then

HAS(s, CHOOSE"(s)) ≡ T (*)

is derivable. CHOOSE"(s) can either return MAX(s) or MIN(s). For both possibilities, (*) is derivable, as

INDEX(MAX(s), s) ≤ rep$Size(s) ≡ T

---

5. (∃ ! j) stands for 'there exists a unique j such that.'

is derivable from the specifications of **MAX** and **INDEX**, and

INDEX(MIN(s), s) ≤ rep$Size(s) ≡ T

is derivable from the specifications of **MIN** and **INDEX**. Note that **CHOOSE"** preserves the equivalence relation **Eqv**.

The implementation of **Set-Int** in Figure 5.1 with **CHOOSE** replaced by **CHOOSE"** is also correct.

5.3.6 Pseudo-Nondeterministic Procedures

A pseudo-nondeterministic procedure (which could be either deterministic or nondeterministic) is not required to preserve the equivalence relation **Eqv**.[6] The correctness proof in this case also is carried as above depending on whether the procedure is deterministic or nondeterministic. However, we must ensure that if the procedure terminates normally for any input X, then it must do so for all input equivalent to X, and if it signals on an input X, then it must signal equivalent exceptions for all input equivalent to X. This ensures that a pseudo-nondeterministic procedure does not have a choice of signalling as well as terminating normally on equivalent rep values.

We now take the implementation of **Set-Int** in Figure 5.1. **CHOOSE** is deterministic; it returns the bottom element of the nonempty sequence. **Eqv** is not preserved by **CHOOSE**. If the axiom '**Choose(s)** ∈ s ≡ T' is derivable under the condition that '**Size(s)** ≠ **0**,' then this implementation is also correct. The proof of the axiom is straightforward: If '**rep$Size(s)** > **0**' holds, then

HAS(s, CHOOSE(s)) ≡ T ⇔ HAS(s, Bottom(s)) ⇔ T

When an implementation does not have any pseudo-nondeterministic procedures, then the interpretation of ≡ in **I** is the largest equivalence relation preserved by the procedures. However, a weaker equivalence relation preserved by the procedures may suffice to show that the restrictions and axioms of **S** hold in **I**.

---

6. For example, a procedure **CHOOSE"'** which nondeterministically picks between the top (last) and the bottom (first) element of the sequence is nondeterministic and does not preserve the equivalence relation **Eqv**. So, **CHOOSE"'** is also pseudo-nondeterministic.

Though the designer of an implementation usually has an idea of what the observable equivalence relation is, sometimes it may not be known. In that case, we will not know what procedures are pseudo-nondeterministic. Then, we choose an equivalence relation preserved by the procedures implementing the deterministic operations, and using it as the interpretation of $\equiv$, we attempt to show that every axiom as interpreted in I is derivable. If successful, the implementation I is correct; otherwise, a stronger equivalence relation is chosen and the above process is repeated. If the correctness of I cannot be established even when the strongest equivalence relation preserved by the procedures implementing the deterministic operations is chosen, then I is incorrect.

Another way to view the above correctness method is to consider the specification of the procedures in an implementation I as axioms of the theory of I, defining the functions computed by the procedures, and show that every nonlogical axiom of Th(S) is in the theory of I. The theory of I also includes the theories of the types used in I. Nakajima et al [62] take a similar view.

5.4 Recursive and Mutually Recursive Implementations

Def. 5.3 An implementation I of D *depends* on a data type D' iff only if
(i) D' is used in I , or
(ii) a data type D'' used in I depends on D'. ∎

In Def. 5.3 above, it is assumed that data types other than D are abstractly used in an implementation I of D. In the correctness method discussed in the previous two sections, we have assumed that

(i) an implementation I of D does not depend on D, and
(ii) an implementation of a data type D' used in I does not depend on D.

We relax these constraints. We call an implementation I of D *recursive* if and only if the rep used in I depends on D. We call an implementation I of D and another implementation I' of D' *mutually recursive* if and only if I depends on D' and I' depends on D. We assume that recursion is not due to internal types used in I. It should be noted that if the implementations of a set of data types are mutually recursive, that does not mean that data types are also mutually recursive (mutually recursive data types are discussed in Section 2.4). We first discuss how the method proposed in Section 5.3 be modified to deal with recursive implementation, later we consider mutually recursive implementation.

5.4.1 Recursive Implementations

In proving correctness of a recursive implementation, we consider a reference to

---

Figure 5.9. An Uninteresting Recursive Implementation of D

```
D = cluster is OP1, OP2, ...
    rep = D
OP1 = proc(...) returns....
        return (D$OP1 (...))
        end OP1
        .
        .
        .
```

D in I as a reference to its rep and an invocation of an operation σ of D as a call to the procedure OP implementing σ. The equate defining the rep inside I is considered as a recursive domain equation, as the construction of the rep depends on D itself. For

---

Figure 5.10. Implementation of List-Int

```
LIST-INT = cluster is NIL, CONS, CAR, CDR, IS_IN, IS_EMPTY

  rep = oneof [ empty: Null, pair: Pair]
  Pair = struct [ car: Int, cdr: List-Int]

NIL = proc() returns (cvt)
      return (rep$make_empty(nil))
      end NIL

CONS = proc(i: Int, l: List-Int) returns (cvt)
       return (rep$make_pair(Pair${car:i, cdr:l}))
       end CONS

CAR = proc(l: cvt) returns (Int) signals (empty)
      tagcase l
       tag pair (p: Pair): return (p.car)
       tag empty: signal empty()
       end
      end CAR

CDR = proc(l: cvt) returns (List-Int) signals (empty)
      tagcase l
       tag pair (p: Pair): return (p.cdr)
       tag empty: signal empty()
       end
      end CDR

IS_IN = proc(i: Int, l: cvt) returns (Bool)
       tagcase l
        tag pair (p: pair): if p.car = i then return (true)
                            else return (List-Int$is_in(i, p.cdr)) end
        tag empty: return (false)
        end
       end IS_IN

IS_EMPTY = proc(l: cvt) returns (Bool)
        return (rep$is_empty(l))
       end IS_EMPTY
```

example, consider the implementation of a data type *list of integers*, denoted by **List-Int**, given in Figure 5.10; its rep is a recursive domain equation. A recursive domain equation can be solved by defining an ordering on type algebras and using Kleene's Recursion Theorem. The rep is the least fixed point solution of the equation (see [3] for details about such an ordering).

For a correct implementation **I**, the type algebras of the rep should have a nonempty principal domain. This property is trivially ensured if rep is nonrecursive. For some recursive implementation such as the one given in Figure 5.9, the least fixed point is the empty algebra, an algebra having no domain and no functions. For well founded rep equates such as in case of **List-Int**, the algebras are nonempty. If the rep can be proved to be nonempty, the method proposed in the previous section can be used. The proof that the least fixed point of a domain equation defining the rep is nonempty is the only additional step in proving the correctness of a recursive implementation.

Figure 5.11 has specifications of the procedures in the implementation of **List-Int**. (The specifications of **Null**, **Struct** $[n_1: D_1, \ldots, n_k: D_k]$, and **One-of** $[n_1: D_1, \ldots, n_k: D_k]$ are given in Appendix IV.) Figure 5.12 is a specification of **List-Int**. We give below a sketch of various steps in the correctness proof of the implementation of **List-Int** given in Figure 5.10.

Figure 5.11. Translation of the Procedures of List-Int

```
rep = oneof [ empty: Null, pair: Pair]
Pair = struct [ car: Int, cdr: List-Int]

NIL() ≜ rep$make_empty(nil)

CONS(i, l) ≜ rep$make_pair(Pair${car: i, cdr: l})

CAR(l) ≜ rep$is_pair(l) ⇒ Pair$get_car(rep$value_pair(l)) ▮
         ~ rep$is_pair(l) ⇒ signal(empty)

CDR(l) ≜ rep$is_pair(l) ⇒ Pair$get_cdr(rep$value_pair(l)) ▮
         ~ rep$is_pair(l) ⇒ signal(empty)

IS_IN(i, l) ≜ rep$is_pair(l) ⇒ (i = Pair$get_car(rep$value_pair(l)) ∨
                  IS_IN(i, Pair$get_cdr(rep$value_pair(l)))) ▮
              ~ rep$is_pair(l) ⇒ false

IS_EMPTY(l) ≜ rep$is_empty(l)
```

Figure 5.12. Specification of List-Int

Operations

```
Nil       : → List-Int
Cons      : Int X List-Int → List-Int
Car       : List-Int → Int
                     → empty ( )
Cdr       : List-Int → List-Int
                     → empty ( )
Is_In     : Int X List-Int → Bool
Is-Empty  : List-Int → Bool
```

Restrictions

```
Is-Empty (l) ⇒ Car(l) signals empty ( )
Is-Empty (l) ⇒ Cdr(l) signals empty ( )
```

Axioms

```
Car(Cons(i, l)) ≡ i
Cdr(Cons(i, l)) ≡ l
Is-In (i, Nil) ≡ F
Is-In(i1, Cons(i2, l)) ≡ if i1 = i2 then T else Is-In (i1, l)
Is-Empty(Nil) ≡ T
Is-Empty(Cons(i, l)) ≡ F
```

(i) the least fixed point of the recursive domain equation is nonempty. For any model of Int, the approximations to the rep can be constructed.

(ii) Inv(s) is T.

(iii) The termination of procedures other than IS_IN is obvious, assuming that the tagcase, and the operations of one-of terminate. For IS_IN, we can prove termination using McCarthy and Cartwright's approach, or by using the fact that the rep is well founded with respect to the ordering, $l <$ one-of [pair: [car: i, cdr: l]] for any i and l.

(iv) the equivalence relation on the rep is the identity relation.

(v) The procedures return normally on an input on which the restriction component does not specify the corresponding operation to signal.

(vi) Every restriction is derivable.

(vii) Every axiom is derivable.

5.4.2 Mutually Recursive Implementations

We prove the correctness of mutually recursive implementations in a way similar as in case of a recursive implementation. The correctness of mutually recursive implementations must be proved together. The reps of the two implementations are specified as mutually recursive domain equations; the solution of these equations are the least fixed points, which serve as the rep of D and the rep of D'. For the implementations I and I' to be correct, both reps must be nonempty. The rest of the proof is same as in case of nonrecursive implementations with the exception that the correctness proof for all mutually recursive implementations is done together. The implementations I and I' have to be shown to satisfy the restrictions and axioms in S and S'. The invocation of an operation of D' in I is considered as a call to the procedure in I' implementing the operation, and the invocation of an operation of D in I' is considered as a call to the procedure in I implementing the operation.

The correctness proof cannot be hierarchically structured in case of mutually recursive implementations, because their correctness has to be proved together. For this reason, we do not recommend that hierarchically structured (nonrecursive) data types be implemented mutually recursively. However, for a set of mutually recursive data types, their implementations have to be proved correct together, so these data type can be implemented mutually recursively without adding to the complexity of the correctness proof.

6. Conclusions

We have presented a rigorous framework for abstract data types, and studied four important aspects of abstract data types, namely definition, specification, theory, and implementation correctness, within this framework. An overview of the approach taken in studying these issues is given in Chapter 1. The framework has provided a base from which to to ask many interesting and important questions about data types. Some of these questions have been answered in the thesis, while others suggest directions for further research. Below, we first summarize the contributions of our work and then indicate areas where further work is required.

6.1 Summary of Contributions

We have made a clear distinction between a data type and its specification(s) in our research. The behavioral approach for defining a data type developed in the thesis embodies the view of a data type taken in programming languages. It considers only the input-output behavior of the operations. It abstracts from the representational structure of the values and the operations of a data type as well as from multiple representations of values for a particular representational structure. Our definitional method can handle data types with nondeterministic operations and with operations exhibiting exceptional behavior. It is independent of specification methods for data types. Specification languages other than the one proposed in the thesis can also be developed based on it. It can be used to give the semantics of existing specification languages. In [43], we have studied and compared the expressive power of various specification languages for data types. Using the definitional method, we have been able to characterize computability over the values of a data type, and study the expressive power of the operation set of different designs of a data type [42].

The specification language proposed in the thesis is structured and flexible. The normal behavior and the exceptional behavior of the operations are specified separately. The language provides mechanisms to specify (i) nondeterministic operations, (ii) preconditions for operations stating what portion of the input domain of an operation is

interesting, (iii) exceptions which must be signalled by the operations, and (iv) exceptions which the operations can optionally signal. In designing the specification language, one of the goals has been to facilitate writing specifications as well as proving properties of data types from their specifications without having to express the properties that can be deduced. The semantics of a specification is given as a set of data types. Equivalence among specifications is defined.

We have proposed a deductive system for abstract data types and studied its different components. A first order theory of a data type is defined, which is constructed from its specification using the deductive system. The well definedness, sufficient completeness and completeness properties of a specification are defined based on what can be deduced from it. These properties are related to the model theoretic properties of a specification. A clear distinction is made between the model theoretic and proof theoretic properties of a specification.

We propose a correctness criterion for an implementation of a data type with respect to its specification, independent of implementation correctness methods and specification methods. Many implementation correctness methods can be developed embodying this criterion. We develop a correctness method which is simple and natural for a wide class of specifications.

Throughout this research, we have emphasized modularity and hierarchical structure, be it the definition, specification, deductive system, or implementation of a data type.

The development of the framework has also provided useful insights into data type behavior and the programming language features, such as the advantage of having a protected encapsulation mechanism for implementing a data type, separation of the exception handlers from the type behavior, significance of hierarchical structure and modularity, etc.

6.2 Directions for Further Research

We first discuss topics of further research emerging from the discussion in various chapters. We later discuss other aspects of data type behavior not studied in the thesis, and finally, the topics in which the assumptions made about data type behavior in the thesis are relaxed.

We have not investigated how easily the deductive system proposed in Chapter 4 can be automated or incorporated into an existing automatic data type deduction system such as AFFIRM. We do not anticipate any major problems in incorporating the subsystem for reasoning about the exceptional behavior of a data type, because the axioms describing the exceptional behavior are similar to equations and can be transformed to rewrite rules. However, the subsystem for reasoning about nondeterministic operations involves axioms using existential quantifiers. A verification system based on first order predicate calculus can in principle incorporate this subsystem. We feel that the full power of first order predicate calculus with its complexity is not required. An approach for untransformed axioms (in which properties are expressed using nondeterministic symbols) similar to rewrite rules for equational axioms needs to be investigated.

The implementation correctness method discussed in Chapter 5 uses an equivalence relation on the values of the rep (representing type) and requires that the implementation be extended to include the definitions of auxiliary functions used in a specification, if any. It would be useful to develop a method that can derive this information from the specification and the implementation. We do not anticipate any problems in automating the remaining steps of the method; however, the interface between a verification system embodying proof rules for control structures and a data type deduction system may need to be analyzed. We are investigating another method that does not require the equivalence relation and the definitions of auxiliary functions for an implementation. It is based on the behavioral equivalence relation on models: For every computation having an observer as its outermost operation, if the specification prescribes a result, a value returned by the computation when interpreted in the implementation must be one of the possible results prescribed by the specification.

The proposed implementation correctness method tells whether an

implementation is correct with respect to a specification. It would be interesting to extend it so that the bug(s) in a incorrect implementation can be located; this would help in debugging the implementation.

Another complimentary area for further study is that of systematic testing for enhancing confidence in a piece of software. In addition to using it for testing programs using the data type, a specification of a data type can be used to design a set of test cases for checking the implementations of the data type. Gannon et al. [19] discuss a system in which a specification of a data type as a set of conditional equations is presented along with a set of test cases which can be executed using the implementation to test for the consistency of the implementation with the specification. A methodology for designing an 'adequate' set of test cases from a specification would be very useful for such a system.

Specifications are often hard to write, and especially the writing of an 'algebraic' specification has been found to be hard [41, 3]. We are investigating a method for writing a specification in a systematic manner; using this method, we have been able to write specifications of data types such as traversable stack [41], file [42], etc. A system that embodies such a method and helps a designer in writing a specification would be very useful. It should assist the designer in analyzing a specification so as to enhance his confidence in the specification. It should check for general structural properties of a specification such as well definedness and completeness, which ensure proper relations among different components of the specification. The undecidability of completeness and well definedness can be shown by reducing them to the Post Correspondence problem [58] in Post systems. However, sufficient conditions on axioms and restrictions which guarantee well definedness and completeness of a specification need to be investigated. The results of Guttag and Horning [28] and Polajnar [67] will probably be helpful in arriving at these conditions.

It is equally important to ensure that a specification indeed captures the intent of the designer. This can be checked in several ways, some of which are complimentary: The designer can express additional properties that a data type should satisfy. He then attempts to prove these properties from its specification using the deductive system. Another approach is for the designer to come up with a model of the data type and then check that

the axioms and restrictions hold in that model. Third approach can be similar to program testing; the specification can be validated on a set of test cases.

Guttag and Horning [32] have suggested how formal specifications can be used as a tool for designing software. Our specification language can be used to aid the design of the data component of software. For it to be used for writing specifications of general software, it must be extended to include mechanisms for specifying mutable behavior, procedural abstractions, other control abstractions, etc.

An important aspect of data types not studied in our framework is the relationships among different data types. One important relationship is among the set of data types defined by a type scheme (also called a parameterized type). Data types in the set defined by a type scheme have similar behavior except that the values of these data types may have their constituents belonging to different types, and the values may have different structural constraints, for example, different upper bounds on the size of the values, etc. This variation in the behavior of different types is expressed using two kinds of parameters: *Constant* parameters ranging over the values of a data type, often used to express the structural constraints on the values, such as bounds on the size of the values, and *type* parameters stating the type of the constituents of the values. For example, a type scheme **Stk[***n* **: Int,** *t* **: Types]** defines a set of data types that have the behavior of stacks, and that differ in the type of the elements of stacks and the upper bound on the size of stacks. **Types** stands for the set of all data types, and is itself not a data type. The data type **Stk-Int-100**, for example, is an instance of the above type scheme with $n = 100$, and $t =$ **Int**.

A type scheme is in general a partial function from the cartesian product of the domains of its parameters to the set of all types, **Types**. For a particular set of parameters, this function either returns a data type or is undefined. For example, the type scheme **Stk** is a function from **Int x Types** to **Types**, and is defined for every set of parameters. However, if parameters of a type scheme are required to satisfy certain properties, then the function returns a data type only if the parameters satisfy the desired properties. For example, in case of the type scheme **Set[***t* **: Types]**, its type parameter must have an equal operation with standard properties.

The specification language proposed in Chapter 3 can be easily extended to specify type schemes. A specification should have an additional component, called *Requires*, stating conditions on the parameters ranging over types. The *Requires* component can specify both the operations that the type parameter must have and their properties. The semantics of such a specification can be easily given. How the deductive system proposed in Chapter 4 can be extended to type schema would need further investigation.

Apart from a type scheme, there are other interesting relations among different data types. There are standard mathematical relations, such as the relation between a cartesian product of data types and its components; the relation between discriminated unions and its components; etc. Some of these relations can be expressed as type schema. The notion of a subtype of a type needs investigation. For example, what relations exist between integers, rationals, and algebraic reals? How do sets, multisets, ordered sets, and sequences relate, and how do stacks and traversable stacks relate?

Our framework is limited in three respects. Firstly, the definition of a data type only incorporates the input-output behavior of its operations. It does not consider another aspect of the operations, namely how efficiently these operations can be performed. It is not even clear whether the computational complexity of the operations should be included in a definition of a data type, or whether it is an orthogonal constraint on the implementations that should be included in a specification. We think that the input-output behavior of the operations of a data type should be kept separate from their computational complexity, but a specification should have another component stating the performance requirements on the implementations of the operations.

Secondly, we have assumed a simple model of nondeterminism in analyzing the input-output behavior of the operations. For an input on which a nondeterministic operation can return many possible results, we have not considered how these results are scheduled. It would be interesting to incorporate the scheduling information and extend the definitions of observable behavior and distinguishability of values. It would also be interesting to investigate how our formalism is affected if we relax the assumption that a nondeterministic operation cannot have the choice of signalling as well as terminating

normally on a particular input.

Thirdly, the definitional method handles only immutable data types. As is discussed in Appendix I, for a wide class of mutable data types, the states of their objects can be modeled as the values of an immutable data type. However, the framework needs to be extended to handle arbitrary mutable data types including data types having objects whose state is also mutable, for example, the data type *list* in MACLISP. The specification language and a deductive system based on the extended framework need to be developed. Berzins's work [3] can be useful in studying this extension.

References

1. Preliminary ADA Reference Manual and Rationale. *SIGPLAN Notices* Vol. 14 No. 6, June, 1979.

2. Berzins, V. *Personal Communication.* Lab. for Computer Science, MIT, Dec., 1976.

3. Berzins, V. Abstract Model Specification for Data Abstractions. LCS-TR-221, Lab. for Computer Science, MIT, MA, 1979.

4. Birkhoff, G., Lipson, J.D. Heterogeneous Algebras. *Journal of Combinatorial Theory* Vol. 8, 1970, pp. 115-133.

5. Brand, D., Daringer, J.A., Joyner, W.H. Completeness of Conditional Reductions. IBM Research Report RC7404, Yorktown Heights, New York, Dec., 1978.

6. Burstall, R.M. Proving Properties of Programs by Structural Induction. *Computer Journal* Vol. 12, Feb., 1969, pp. 41-48.

7. Burstall, R.M., Goguen, J.A. Putting Theories Together To Make Specifications. Invited Paper at the *Fifth International Joint Conf. on Artificial Intelligence* Cambridge, MA, Aug., 1977.

8. Cartwright, R, and McCarthy, J. Recursive Programs as Functions in a First Order Theory. Report No. STAN-CS-79-717, Stanford University, March, 1979.

9. Cohen, J. Nondeterministic Algorithms. *Computing Surveys* Vol. 11 No. 2, June, 1979, pp. 79-94.

10. Cohn, P.M. Universal Algebra. Harper and Row, New York, 1965.

11. Dahl, O.-J., Nygaard, K., Myhrhaug, B. The Simula 67 Common Base Language. Norwegian Computing Center, Forskningsvein 1B, Oslo, 1968.

12. Dijkstra, E.W. Notes on Structured Programming. In *Structured Programming* (Dahl, O.-J., Dijkstra, E.W., Hoare, C.A.R.), Academic Press, London and New York, 1972, pp. 1-81.

13. Dijkstra, E.W. A Discipline of Programming. Prentice Hall, Englewood Cliffs, NJ, 1976.

14. Dershowitz, N., and Manna, Z. Proving Termination with Multiset Ordering. *Comm. ACM* Vol. 22 No. 8, Aug., 1979, pp. 465-476.

15. Ehrig, H., Kreowski, H., Padawitz, P. Stepwise Specification and Implementation of Abstract Data Types. Proceedings of the *Fifth International Colloq. on Automata, Language, and Programming*, Udine, as *Lecture Notes in Computer Science* Vol. 62, Springer-Verlag, 1978, pp. 205-226.

16. Enderton, H.B. A Mathematical Introduction to Logic. Academic Press, New York and London, 1972.

17. Floyd, R.W. Assigning Meanings to Programs. Proceeding of a *Symposium in Applied Math.*, Vol. 19 as *Mathematical Aspects of Computer Science* (ed. Schwartz, J.T.), American Mathematical Society, Providence, R.I., 1967, pp. 19-32.

18. Friedman, D.P., Wise, D.S. CONS should not Evaluate its Arguments. Technical Report No. 44, Computer Science Dept., Indiana University, Nov., 1975.

19. Gannon, J., McMullin, P., Hamlet, R., Ardis, M. Testing Traversable Stack. *SIGPLAN Notices* Vol. 15 No. 1, Jan., 1980, pp. 58-65.

20. Goguen, J.A. Abstract Errors for Abstract Data Types. Proceedings of the IFIP Working Conference on *Formal Basis of Programming Concepts* Vol. 2, Aug., 1977, pp. 21.1-21.32.

21. Goguen, J.A., and Tardo, J.J. An Introduction to OBJ : A Language for Writing and Testing Formal Algebraic Program Specifications. Proceedings IEEE Conf. on *Specifications of Reliable Software*, Cambridge, MA, April, 1979, pp. 170-189.

22. Goguen, J.A., Thatcher, J.W., Wagner, E.G., Wright, J.B. Abstract Data Types as Initial Algebras and Correctness of Data Representations. Proceedings, Conference on *Computer Graphics, Pattern Recognition and Data Structure*, May, 1975, pp. 89-93.

23. Goguen, J.A., Thatcher, J.W., Wagner, E.G. Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types. In *Current Trends in Programming Methodology*, Vol. IV, Data Structuring, (ed. Yeh, R.T.), Prentice Hall, Englewood Cliffs, NJ, 1978.

24. Goodenough, J.B. Exception Handling: Issues and A Proposed Notation. *Comm. ACM* Vol. 18 No. 12, Dec., 1975, pp. 683-696.

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

26. Guttag, J.V. Abstract Data Types and the Development of Data Structures. *Comm. ACM* Vol. 20 No. 6, June, 1977, pp. 396-404.

27. Guttag, J.V., Horowitz, E., Musser, D.R. The Design of Data Type Specification. In *Current Trends in Programming Methodology*, Vol. IV, Data Structuring, (ed. Yeh, R.T.), Prentice Hall, Englewood Cliffs, NJ, 1978.

28. Guttag, J.V., Horning, J.J. The Algebraic Specification of Abstract Data Types. *Acta Informatica* Vol. 10 No. 1, 1978, pp. 27-52.

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

30. Guttag, J.V. *Personal Communication*, May, 1979.

31. Guttag, J.V. Notes on Type Abstraction. *IEEE Trans. on Software Engineering* Vol. SE-6 No. 1, Jan., 1980, pp. 13-23.

32. Guttag, J.V., Horning, J.J. Formal Specification as a Design Tool. Proceedings of the *Seventh ACM Symposium on Principles of Programming Languages*, Las Vegas, Nevada, Jan., 1980.

33. Guttag, J.V. *Personal Communication*, Jan., 1980.

34. Harel, D., Pratt, V.R. Comments on Program Verification. In *Research Directions in Software Technology* (ed. Wegner, P.), M.I.T. Press, Cambridge, MA, 1979, pp. 387-391.

35. Hewitt, C. *Personal Communication.* Lab. for Computer Science, MIT, Dec., 1978.

36. Hoare, C.A.R. Procedures and Parameters: An Axiomatic Approach. In *Symposium on Semantics of Algorithmic Languages*, (ed. Engeler, E.) as *Lecture Notes in Mathematics*, No. 188, Springer Verlag, 1971, pp. 102-115.

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

38. Hoare, C.A.R. Notes on Data Structuring. In *Structured Programming*, (Dahl, O.-J., Dijkstra, E.W., Hoare, C.A.R.), Academic Press, London and New York, 1972, pp. 83-174.

39. Hoare, C.A.R. Recursive Data Structures. *Intl. Journal of Computer and Information Sciences* Vol. 4 No. 2, June, 1975, pp. 105-132.

40. Kapur, D. Proving Correctness of Implementation of a Data Abstraction Using the Algebraic Method. Unpublished Handout, M.I.T. Course 6.891 *Specification Techniques*, Nov., 1975.

41. Kapur, D. Specifications of Majster's Traversable Stack and Veloso's Traversable Stack. *SIGPLAN Notices* Vol. 14 No. 5, May, 1979, pp. 46-53.

42. Kapur, D., Srivas, M.K. Expressiveness of the Operation Set of A Data Abstraction. Proceedings of the *Seventh ACM Symposium on Principles of Programming Languages*, Las Vegas, Nevada, Jan., 1980. An expanded version appeared as Computation Structures Group Memo 179-1, Lab. for Computer Science, MIT, Jan., 1980.

43. Kapur, D. The Expressive Power of Algebraic Languages for Specifying Abstract Data Types. Draft Manuscript, Lab. for Computer Science, MIT, June, 1979.

44. Knuth, D.E., Bendix, P.B. Simple Word Problems in Universal Algebra. In *Computational Problems in Abstract Algebra* (ed. Leech, J.), Pergamon Press, 1970, pp. 263-297.

45. Lampson, B.W., Horning, J.J., London, R.L., Mitchell, J.G., Popek, G.L. Report on the Programming Language Euclid. *SIGPLAN Notices* Vol. 12 No. 2, Feb., 1977.

46. Levin, R. Program Structures for Exceptional Condition Handling. Ph.D. Thesis, Dept. of Computer Science, Carnegie-Mellon University, June, 1977.

47. Liskov, B.H., Zilles, S.N. Specification Techniques for Data Abstractions. *IEEE Trans. on Software Engg.* Vol. SE-1 No. 1, 1975, pp. 7-19.

48. Liskov, B.H., Berzins, V. An Appraisal of Program Specifications. Computation Structures Group Memo 141-1, Lab. for Computer Science, MIT, Jan., 1977. Also in *Research Directions in Software Technology* (ed. Wegner, P.), M.I.T. Press, Cambridge, MA, 1979, pp. 276-301.

49. Liskov, B.H., Snyder, A., Atkinson, R., Schaffert, C. Abstraction Mechanisms in CLU. *Comm. ACM* Vol. 20 No. 8, Aug., 1977, pp. 564-576.

50. Liskov, B.H., Snyder, A.S. Exception Handling in CLU. *IEEE Trans. on Software Engg.* Vol. SE-5 No. 6, Nov., 1979, pp. 547-557.

51. Liskov, B.H. Modular Program Construction Using Abstraction. Computation Structures Group Memo 184, Lab. for Computer Science, MIT, Sept., 1979.

52. Liskov, B.H. et al. CLU Reference Manual. MIT-LCS-TR-225, Lab. for Computer Science, MIT, Oct., 1979.

53. Majster, M.E. Limits of the Algebraic Specification of Abstract Data Types. *SIGPLAN Notices* Vol. 12 No. 10, Oct., 1977, pp. 37-42.

54. Manna, Z. Mathematical Theory of Computation. McGraw Hill, Computer Science Series, 1974.

55. Manna, Z. Six Lectures on the Logic of Computer Programming. Stanford A.I. Laboratory AIM-318, Nov., 1978.

56. McCarthy, J. Towards a Mathematical Science of Computation. Proceedings *IFIP Congress*, 1962, pp. 27-28.

57. McCarthy, J. A Basis for a Mathematical Theory of Computation. In *Computer Programming and Formal Systems* (eds. Braffort and Hirschberg), North Holland Publishing Co., Amsterdam-London, 1963, pp. 33-70.

58. Minsky, M. Computation: Finite and Infinite Machines. Prentice Hall, Englewood Cliffs, NJ, 1967.

59. Morris, J.H., Jr. Types Are Not Sets. Proceedings of the *First ACM Symposium on Principles of Programming Languages*, Boston, Oct., 1973, pp. 120-124.

60. Musser, D.R. Abstract Data Types in the AFFIRM System. *IEEE Trans. on Software Engg.* Vol. SE-6 No. 1, Jan., 1980, pp. 24-31.

61. Musser, D.R. Proving Inductive Properties of Abstract Data Types. Proceedings of the *Seventh ACM Symposium on Principles of Programming Languages*, Las Vegas, Nevada, Jan., 1980.

62. Nakajima R., Nakahara, H., Honda, M. Hierarchical Program Specification and Verification - A Many Sorted Logical Approach. Preprint RIMS 265, Nov., 1978.

63. Nourani, F. Constructive Extension and Implementation of Abstract Data Types and Algorithms. Ph.D. Thesis, Dept. of Computer Science, University of California, Los Angeles, June, 1979.

64. Okrent, H.F. Synthesis of Data Structures from Algebraic Descriptions. Ph.D. Thesis, Dept. of E.E. & C.S., MIT, Feb., 1977.

65. Palme, J. Protected Program Modules in Simula 67. National Defense Research Institute, Stockholm, Sweden, July, 1973.

66. Parnas, D.L. Information Distribution Aspects of Design Methodology. *Information Processing 71*, Vol. I, North Holland, Amsterdam, 1972, pp. 339-344.

67. Polajnar, J. An Algebraic View of Protection and Extendibility in Abstract Data Types. Ph.D. Thesis, Dept. of Computer Science, University of Southern California, Sept., 78.

68. Srivas, M.K. Preliminary Investigations of a Thesis Topic on Automatic Synthesis of Abstract Data Types. Unpublished Manuscript, Lab. for Computer Science, MIT, Dec., 1978.

69. Standish, T.A. Data Structures - An Axiomatic Approach. Bolt, Boranek, and Newman, Inc., Technical Report 2639, Aug., 1973.

70. Subrahmanyam, P. On a Finite Axiomatization of the Data Type L. *SIGPLAN Notices* Vol. 13 No. 4, April, 1978, pp. 80-84.

71. Thatcher, J.W., Wagner, E.G., Wright, J.W. Data Type Specification: Parameterization and the Power of Specification Techniques. Proceedings of the *Tenth SIGACT Conference*, May, 1978. Also an IBM Report RC7757, July, 1979.

72. Wegbreit, B., and Spitzen, J.M. Proving Properties of Complex Data Structures. *JACM* Vol. 23 No. 2, April, 1976, pp. 389-396.

73. Wirth, N. Program Development by Stepwise Refinement *Comm. ACM* Vol. 14 No. 4, April, 1971, pp. 221-227.

74. Wulf, W., London, R.L., and Shaw, M. Abstraction and Verification in ALPHARD: Introduction to Language and Methodology. USC Information Sciences Institute Research Report, 1976.

75. Wulf, W., London, R.L., and Shaw, M. An Introduction to the Construction and Verification of Alphard Programs. *IEEE Trans. on Software Engg.* Vol. SE-2 No. 4, Dec., 1976, pp. 253-265.

76. Zilles, S.N. Algebraic Specification of Data Types. Project MAC Progress Report, 1974, pp. 52-58. Also Computation Structure Group Memo 119, Lab. for Computer Science, MIT, 1974.

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

Appendix I - Elaboration of Scope and Assumptions

In this appendix, we elaborate on the scope of the thesis and the assumptions made about abstract data types and their operations.

1. Immutable and Mutable Data Types

We adopt the commonly accepted informal view of a data type as a collection of objects with a finite collection of operations to manipulate these objects. The objects by themselves are not meaningful and the operations are the only way to construct, manipulate and observe the objects as well as to extract information stored in them.

Data types can be classified based on their object behavior. An object of a data type may or may not exhibit time varying behavior. An object exhibiting time varying behavior is called a *mutable* object, whereas an object whose behavior does not change is called an *immutable* object [49]. We also call an immutable object a *value*. A data type having only immutable objects is called an *immutable* data type; otherwise, a data type is called a *mutable* data type. A mutable data type may also have immutable objects, but at least one of its objects must be mutable. A mutable object can be factored into two components: (i) *identity*, and (ii) *state* [47]. A mutable data type has at least one operation constructing new objects. Its operations may change the state of a mutable object without affecting the object identity. At a given point in a computation, there can exist many different mutable objects having the same state. For a wide class of mutable data types, the state component of the mutable objects can be described as an immutable data type.

In this thesis, we have considered only immutable data types with a finite set of computable operations. We have not considered immutable data types with iterators [49] nor data types involving streams and lazy evaluation [18].

2. Exceptional Behavior

During the design and construction of reliable software, there is often a need to have data types with operations exhibiting *exceptional behavior.* (See [24, 46, 52, 50] for a discussion on the need for an *exception handling mechanism* in a programming language.) It is only meaningful to apply such operations on a subset of their domains. If an input falls outside the subset, such operations notify their callers indicating that the input is not 'good,' by *signalling exceptions.* An exception is assumed to have two components, a descriptive name and a possible set of arguments which carry information from the point where the exception is signalled, to its handlers.

We assume that every operation of a data type terminates on every input in its domain: it either terminates normally by returning a value of its range type or terminates by signalling an exception. We think it is not a good practice to design data types having operations that do not terminate on some inputs. If a partial function on the values of a data type needs to be realized, it can be programmed in terms of the operations of the data type in a host programming language supporting the data type mechanism.

The assumption of the operations being total simplifies the formalism developed in the thesis. Our formalism can be extended to partial operations without much difficulty by introducing a special value 'undefined' for every data type such that if a partial operation is not defined on an input, then it returns 'undefined' on that input.

3. Nondeterminism

There are data types some of whose operations exhibit nondeterministic behavior. These operations return one of many possible values for a given input. For example, the **Choose** operation of the data type *finite set of integers*, which returns any element of a given nonempty set, is nondeterministic. Similarly, the **Index** operation of the data type *finite sequence of elements*, which returns a position of a given element in a given sequence, is also nondeterministic because the sequence can have more than one occurrence of the same element. All prior work on data types has assumed the operations to be deterministic. We feel that a formalism for data types must be capable of handling data types with

nondeterministic operations, as nondeterminism is a powerful and elegant abstraction mechanism for designing programs [13, 9]. Furthermore, allowing nondeterministic operations permits the handling of data types with operations implemented in a parallel environment.

We assume that a nondeterministic operation has only finitely many choices on a particular input. We rule out data types having operations with infinitely many choices. Such an operation can be used to write programs having unbounded nondeterminism [13]. There is a controversy about the the realizability of programming constructs having unbounded nondeterminism and about the limitation of the expressive power of a language that rules out programs with unbounded nondeterminism [35]. Using our formalism, it is possible to define a data type whose values are 'infinite' (e.g., 'infinite' sets, 'infinite' sequences, etc.,) insofar as these values can be finitely constructed using the operations; but, nondeterministic operations on these values that have infinitely many choices are ruled out. Our formalism would however extend without much difficulty to the case where the constraint that a nondeterministic operation has only finitely many choices on an input, is dropped.

We also assume that if a nondeterministic operation signals an exception on an input, then the operation behaves deterministically on the input. Thus a nondeterministic operation is not allowed to have a choice between signalling and terminating normally on any particular input. This assumption leads to a simpler and modular characterization of the observable behavior of the data type than would otherwise be possible.

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

In the first section, we extend the definitions of congruence, homomorphism, and isomorphism to extended heterogeneous algebras having nondeterministic functions. In the second section, we present the proof of Theorem 2.2. In the third section, we explain how the Definition 2.12 of behavioral equivalence on type algebras captures the desired property that a computation (i.e., an interpretation of a ground term) results in equivalent values in two behaviorally equivalent type algebras.

1. Congruence, Homomorphism, and Isomorphism

Def. A2.1 A *congruence* R on a conventional heterogeneous algebra

$$\mathbf{A} = [\{ V_{D'} \mid D' \in \Delta' \}; \{ f_\sigma \mid \sigma \in \Omega \}],$$

in which each f_σ is a total deterministic function, is a family of equivalence relations $\{ R_{D'} \mid D' \in \Delta' \}$ such that

for every $\sigma \in \Omega, \sigma : D_1 X \ldots X D_n \rightarrow D'$,

for all $v_1 \in V_{D_1}, \ldots, v_n \in V_{D_n}$

$$v_1 R_{D_1} v_1', \ldots, v_n R_{D_n} v_n' \Rightarrow f_\sigma(v_1, \ldots, v_n) R_{D'} f_\sigma(v_1', \ldots, v_n'). \qquad (*)$$

We also say that R has the *substitution property.*

In an extended heterogeneous algebra having nondeterministic functions, when f_σ is a nondeterministic total function, then (*) is modified to

$$v_1 R_{D_1} v_1', \ldots, v_n R_{D_n} v_n' \Rightarrow (\forall y \in \{ f_\sigma(v_1, \ldots, v_n) \} \exists z \in \{ f_\sigma(v_1', \ldots, v_n') \} [y R_{D'} z]$$
$$\wedge \forall z \in \{ f_\sigma(v_1', \ldots, v_n') \} \exists y \in \{ f_\sigma(v_1, \ldots, v_n) \} [y R_{D'} z]).$$

If $R_{D'}$ is the identity relation (equality), then the above reduces to

$$\{ f_\sigma(v_1, \ldots, v_n) \} = \{ f_\sigma(v_1', \ldots, v_n') \}. \blacksquare$$

Congruences on an extended heterogeneous algebra **A** can also be partially ordered in the same way as in case of a conventional heterogeneous algebra:

Given two congruences E^1 and E^2, E^2 is *larger* than E^1, expressed as $E^1 \leq E^2$, if and only if for each $D' \in \Delta'$, $E_{D'}^1 \subseteq E_{D'}^2$.

Congruences form a lattice with respect to $\leq$, and have the least element (the identity congruence) and the greatest element (the universal congruence).

Def. A2.2 Let $\mathbf{A}_1$ and $\mathbf{A}_2$ be

$$\mathbf{A}_1 = [\{ V^1_{D'} \mid D' \in \Delta' \} ; \{ f^1_\sigma \mid \sigma \in \Omega \}]$$
$$\mathbf{A}_2 = [\{ V^2_{D'} \mid D' \in \Delta' \} ; \{ f^2_\sigma \mid \sigma \in \Omega \}].$$

A family of total (deterministic) functions $\Phi = \{ \Phi_{D'} : V^1_{D'} \to V^2_{D'} \mid D' \in \Delta' \}$ is called a *homomorphism* from $\mathbf{A}_1$ to $\mathbf{A}_2$ if

for each $\sigma : D_1 \times \ldots \times D_n \to D'$,

for each v_1 of type D_1 (i.e., $v_1 \in V^1_{D_1}$), ..., v_n of type D_n,

(i) if f^1_σ is deterministic, then f^2_σ is also deterministic and

$$\Phi_{D'}(f^1_\sigma(v_1, \ldots, v_n)) = f^2_\sigma (\Phi_{D_1} (v_1), \ldots, \Phi_{D_n} (v_n)), \text{ and}$$

(ii) if f^1_σ is nondeterministic, then f^2_σ is either nondeterministic or deterministic, and

$$\Phi_{D'}(\{ f^1_\sigma(v_1, \ldots, v_n) \}) = \{ f^2_\sigma (\Phi_{D_1} (v_1), \ldots, \Phi_{D_n} (v_n)) \}. \blacksquare$$

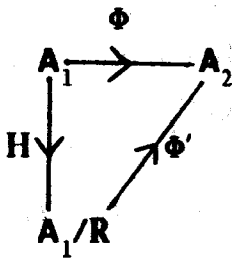
(Case (ii) above covers case (i) also.) We call Φ an onto *homomorphism* from $\mathbf{A}_1$ to $\mathbf{A}_2$ if every function in Φ is onto; in that case, $\mathbf{A}_2$ is called a *homomorphic image* of $\mathbf{A}_1$. If every function in Φ is a bijection, then Φ is an *isomorphism* from $\mathbf{A}_1$ to $\mathbf{A}_2$, and $\mathbf{A}_1$ and $\mathbf{A}_2$ are *isomorphic*. Note that, if $\mathbf{A}_1$ and $\mathbf{A}_2$ are isomorphic nondeterministic algebras, then they have the same amount of nondeterminism, which is not necessarily the case if $\mathbf{A}_2$ is a homomorphic image of $\mathbf{A}_1$.

It can be shown that the results from conventional heterogeneous algebras in [4] extend to the extended heterogeneous algebras. In particular, we can show that

Prop. A2.1 If R is a congruence on an extended heterogeneous algebra $\mathbf{A}$, then there exists an onto homomorphism from $\mathbf{A}$ to $\mathbf{A}/R$. ∎

Prop. A2.2 If Φ is an onto homomorphism from $\mathbf{A}_1$ to $\mathbf{A}_2$ then the kernel R of Φ on $\mathbf{A}_1$, where $R = \{ R_{D'} \mid D' \in \Delta' \}$ and $R_{D'} = \{ \langle v, v' \rangle \mid \Phi_{D'}(v) = \Phi_{D'}(v') \}$, is a congruence on $\mathbf{A}_1$. ∎

The following diagram in which Φ is an onto homomorphism from $\mathbf{A}_1$ to $\mathbf{A}_2$, **R** is the kernel of Φ on $\mathbf{A}_1$, **H** is the homomorphism induced by **R** from $\mathbf{A}_1$ to $\mathbf{A}_1/\mathbf{R}$, and Φ' is an isomorphism from $\mathbf{A}_1/\mathbf{R}$ to $\mathbf{A}_2$, commutes, i.e., $\Phi = \Phi' \cdot \mathbf{H}$.



2. Proof of Theorem 2.2

Thm. 2.2 Assuming that $\mathbf{E}_{\mathbf{BOOL}}$ is the largest congruence on a model of **Bool**, **E** is the largest congruence on **A**.

Proof By induction on type algebras.

Basis: $\Delta = \varnothing$, the null set.

(i) **Bool** - the statement holds because of the assumption.

(ii) D other than **Bool** - since every value in $\mathbf{V}_{\mathbf{D}}$ is observably equivalent to every other value, the statement is true.

Inductive Step: $\Delta \neq \varnothing$,

Assume that the statement holds for each $\mathbf{D}' \in \Delta$.

To prove the statement for **D**, we must show that $\mathbf{E}_{\mathbf{D}}$ is the largest equivalence relation such that **E** is a congruence on **A**. We prove this by contradiction.

Suppose $\mathbf{E}_{\mathbf{D}}$ is not the largest equivalence relation and $\mathbf{E}'_{\mathbf{D}}$ is a larger equivalence relation containing $\mathbf{E}_{\mathbf{D}}$ such that $\mathbf{E}' = \{ \mathbf{E}_{\mathbf{D}'} \mid \mathbf{D}' \in \Delta \} \cup \{ \mathbf{E}'_{\mathbf{D}} \}$ is a congruence on **A**. There exists $\langle v, v' \rangle \in \mathbf{E}'_{\mathbf{D}}$ such that $\langle v, v' \rangle \notin \mathbf{E}_{\mathbf{D}}$. So, there is a $\mathbf{c}(x)$ of type $\mathbf{D}' \in \Delta$ such that there is an interpretation of $\mathbf{c}[x/v]$ in **A** distinguishable from every interpretation of $\mathbf{c}[x/v']$ in **A** or vice versa. But, this is contradictory to **E'** being a congruence which requires that for every interpretation v_1 of $\mathbf{c}[x/v]$ in **A**, there is an interpretation v'_1 of $\mathbf{c}[x/v']$ in **A** such that $\langle v_1, v'_1 \rangle \in \mathbf{E}_{\mathbf{D}'}$, and vice versa. So, $\mathbf{E}_{\mathbf{D}}$ is the largest equivalence relation. ∎

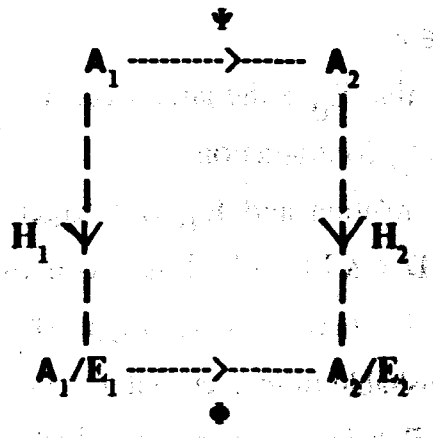
Modification for type algebras having an exception domain

The proof has the same structure as above, except that we also have to consider the case when $\langle v, v' \rangle \notin E_D$ implies that v and v' are distinguishable because a computation $c(x)$ (i) signals on v and returns a normal value on v', or vice versa, or (ii) signals distinguishable exceptional values on v and v'. In the basis step, for the case of D other than **Bool**, E_D need not be the universal relation on V_D.

3. Elaboration of the Definition of Behavioral Equivalence and Proofs of Theorems 2.5 and 2.6

In Section 2.2, we defined two type algebras to be behaviorally equivalent if their reduced algebras are isomorphically equivalent. We further elaborate on this definition. We prove Theorems 2.5 and 2.6 of Section 2.2. The discussion and theorems of this section extend to modified type algebras having the exception domain. The set of mappings from a modified type algebra **A** to another modified type algebra **A'** includes a mapping from the exception domain of **A** to the exception domain of **A'** which gets defined by the mappings on the normal domains.

As is discussed in Subsection 2.2.5, the behavioral equivalence of type algebras A_1 and A_2 can be expressed as



such that the above diagram commutes, i.e.,

$$\Phi \cdot H_1 = H_2 \cdot \Psi, \qquad (\dagger)$$

where A_1/E_1 and A_2/E_2 are the reduced algebras corresponding to A_1 and A_2 respectively

and Φ is the isomorphism defined by the isomorphic equivalence of $\mathbf{A}_1/\mathbf{E}_1$ and $\mathbf{A}_2/\mathbf{E}_2$. The equation (†) above defines the set Ψ of many to many mappings, where $\Psi = \{ \Psi_{D'} : V^1_{D'} \rightarrow V^2_{D'} \mid D' \in \Delta \cup \{ D \} \}$.

We first discuss how for two isomorphically equivalent algebras $\mathbf{A}_1$ and $\mathbf{A}_2$, the bijection Φ_D in an isomorphism Φ can be constructed, and show that the interpretations of a ground term e in $\mathbf{A}_1$ and $\mathbf{A}_2$ are 'equivalent.' Later, we discuss these properties for behaviorally equivalent algebras.

3.1 Isomorphically Equivalent Type Algebras

For the case when the deterministic constructors of a data type D can generate all the values of D, we have

Thm. A2.1 If $\mathbf{A}_1$ and $\mathbf{A}_2$ are isomorphically equivalent, then $\{ \Phi_{D'} \mid D' \in \Delta \}$ uniquely determines the bijection Φ_D.

Proof By definition of isomorphic equivalence, there exists a bijection $\Phi_D : V^1_D \rightarrow V^2_D$ such that $\Phi = \{ \Phi_{D'} \mid D' \in \Delta' \}$ is an isomorphism. We prove the statement by contradiction. Let us assume that Φ_D is not unique; instead, there are two bijections Φ^1_D and Φ^2_D such that $\Phi^1 = \{ \Phi_{D'} \mid D' \in \Delta \} \cup \{ \Phi^1_D \}$ and $\Phi^2 = \{ \Phi_{D'} \mid D' \in \Delta \} \cup \{ \Phi^2_D \}$ are isomorphisms.

Since Φ^1_D and Φ^2_D are different, there exists $v \in V^1_D$, $\Phi^1_D(v) \neq \Phi^2_D(v)$. We pick a v that can be constructed by the minimum number (say k) of applications of the deterministic constructors and on which Φ^1_D and Φ^2_D differ. We have $v = f^1_\sigma(v_1, \ldots, v_n)$ for some σ, and if $D_i = D$, v_i can be constructed by $k' < k$ number of applications of constructors; thus, $\Phi^1_D(v_i) = \Phi^2_D(v_i)$.

By the definition of isomorphic equivalence,

$\Phi^1_D(v) = f^2_\sigma(\Phi_{D_1}(v_1), \ldots, \Phi^1_D(v_i), \ldots, \Phi_{D_n}(v_n))$, and

$\Phi^2_D(v) = f^2_\sigma(\Phi_{D_1}(v_1), \ldots, \Phi^2_D(v_i), \ldots, \Phi_{D_n}(v_n))$,

meaning that $\Phi^1_D(v) = \Phi^2_D(v)$, which is a contradiction.

So, there are not any v such that $\Phi^1_D(v) \neq \Phi^2_D(v)$.

Hence the proof of the theorem. ∎

We can construct the bijection Φ_D as follows:

For every constructor σ: $D_1 \times \ldots \times D_n \rightarrow D$

$$(\Phi_{D_1}(v_1) = v_1' \wedge \ldots \wedge \Phi_{D_n}(v_n) = v_n') \Rightarrow \Phi_D(f_\sigma^1(v_1, \ldots, v_n)) = f_\sigma^2(v_1', \ldots, v_n').$$

The case of σ's not taking any argument of type D serves as the basis step in the construction of Φ_D.

The above theorem holds in case A_1 and A_2 are reduced even if some of the values of D cannot be constructed without using a nondeterministic constructor. However, it does not hold in general; for example, consider a variation of the type algebra A_{si}^1 for Set-Int denoted by A_{si}^5, having everything else the same as in A_{si}^1 except that In^5, the interpretation of the operation **Insert**, is a nondeterministic function, which appends the integer being inserted to the beginning of the sequence representing the given set or at the end of the sequence.

$$In^5(\langle i_1, \ldots, i_m \rangle, i) \triangleq \begin{cases} \langle i_1, \ldots, i_m \rangle & \exists\, 1 \le j \le m,\ i_j = i, \\ \langle i_1, \ldots, i_m, i \rangle \text{ or } \langle i, i_1, \ldots, i_m \rangle & \text{otherwise.} \end{cases}$$

A_{si}^5 is clearly isomorphically equivalent to itself and there is more than one isomorphism from A_{si}^5 to itself.

Thm. A2.2 Given two isomorphically equivalent type algebras A_1 and A_2 defining an isomorphism Φ, a value v of type D in A_1 has the same observable behavior in A_1 as $\Phi_D(v)$ in A_2 in the sense that for every term $c(x)$ of type $D'' \in (D)^*$ with one free variable of type D,

$$\Phi_{D''}(\{\, c[x/v] \mid_{A_1} \}) = \{\, c[x/\Phi_D(v)] \mid_{A_2} \}.$$

Proof By induction on the depth of x in $c(x)$.

$\text{depth}(x) = 0$.

$\text{depth}(\sigma(e_1, \ldots, e_n)) = \max(\text{depth}(e_1), \ldots, \text{depth}(e_n)) + 1$,

where e_i has x as a variable.

Basis $\text{depth}(c(x)) = 0$.

So, $c(x)$ is x, and the statement of the theorem trivially holds.

Inductive Step Assume the statement of the theorem for the case when $\text{depth}(c(x)) < k > 0$, to show for the case when $\text{depth}(c(x)) = k$. Let

$c(x) = \sigma(e_1, ..., e_n)$,

where e_i is of type D_i. We assume that the statement holds for each e_i, so

$\Phi_{D_i}(\{ e_i[x/v]\|_{A_1} \}) = \{ e_i[x/\Phi_D(v)]\|_{A_2} \}$.

$\Phi_{D''}(\{ c[x/v]\|_{A_1} \}) = \Phi_{D''}(\{ f^1_\sigma (\{e_1[x/v]\|_{A_1} \}, ..., \{e_n[x/v]\|_{A_1} \}) \})$

$= \{ f^2_\sigma(\Phi_{D_1}(\{ e_1[x/v]\|_{A_1} \}), ..., \Phi_{D_n}(\{ e_n[x/v]\|_{A_1} \})) \}$ (since Φ is an isomorphism)

$= \{ f^2_\sigma(\{e_1[x/\Phi_D(v)]\|_{A_2} \}, ..., \{ e_n[x/\Phi_D(v)]\|_{A_2} \}) \}$

$= \{ \sigma(e_1, ..., e_n)[x/\Phi_D(v)]\|_{A_2} \} = \{ c[x/\Phi_D(v)]\|_{A_2} \}$. ∎

For the case of modified type algebras, we are interested in terms that such that $c[x/v]\|_{A_1}$ and $c[x/\Phi_D(v)]\|_{A_2}$ are not undefined.

3.2 Behaviorally Equivalent Type Algebras

Thm. A2.3 If A_1 and A_2 are behaviorally equivalent,

then $\langle v, v' \rangle \in \Psi_D \Rightarrow \langle [v], [v'] \rangle \in \Phi_D$.

Proof Obvious from the diagram. Since $\Phi \cdot H_1 = H_2 \cdot \Psi$, from $\langle v, v' \rangle \in \Psi_D$, we get

$\Phi_D([v]) = [v']$. ∎

We now present the proofs of Theorems 2.5 and 2.6 of Subsection 2.2.5.

Thm. 2.5 For behaviorally equivalent A_1 and A_2, for every ground term e of type $D'' \in (D)^*$, for every $v \in \{ e\|_{A_1} \}$, there is a $v' \in \{ e\|_{A_2} \}$ such that $\langle [v], [v'] \rangle \in \Phi_{D''}$, and vice versa.

Proof By induction on the structure of type algebras.

1. Basis $\Delta = \emptyset$

(i) D is **Bool**: Since all behaviorally equivalent algebras are isomorphic and the observable equivalence relation is the identity relation, the above is true.

(ii) D is other than **Bool**: Since the observable equivalence relation is the universal relation, the above is true.

1. Inductive Step $\Delta \neq \emptyset$

Assume that the above statement holds for all ground terms of type $D'' \in (D)^+$ not

having any operation symbol in Ω. (1)

To show for a ground term e by induction on number of operation symbol from Ω in e.

2. The basis step holds because of the assumption.

2. Inductive Step Assume for e having $k' < k$ occurrences of operation symbols from Ω, to show for e having k occurrences. (2)

This is also proved by induction on the depth of the outermost operation symbol from Ω in e.

$\text{depth}(\sigma(e_1, ..., e_n)) = 0$ if $\sigma \in \Omega$, and

$\text{depth}(\sigma(e_1, ..., e_n)) = \min(\text{depth}(e_1), ..., \text{depth}(e_n)) + 1$ if $\sigma \notin \Omega$.

3. Basis $\text{depth}(e) = 0$, i.e., $e = \sigma(e_1, ..., e_n)$, and $\sigma \in \Omega$.

So, an e_i can have at most k-1 occurrences of operations from Ω.

We prove the statement of the theorem in one direction; the proof in the other direction is the same except that v is to be replaced for v'.

If $v \in \{ e|_{A_1} \}$, i.e., if $[v] \in \{ e|_{A_1/E_1} \}$, there is a choice of g^1_σ, the interpretation of σ in A_1/E_1, such that

$[v] = g^1_\sigma([v_1], ..., [v_n])$, where $[v_i] \in \{ e_i|_{A_1/E_1} \}$ for each $1 \leq i \leq n$.

By inductive hypothesis (2), for every $[v_i] \in \{ e_i|_{A_2/E_2} \}$, there is a $[v_i'] \in \{ e_i|_{A_2/E_2} \}$ such that $\Phi_{D_i}([v_i]) = [v_i']$. Because Φ is an isomorphism, there is a choice of g^2_σ such that

$\Phi_{D'}([v]) = [v'] = g^2_\sigma([v_1'], ..., [v_n'])$ meaning that $v' \in \{ e|_{A_2} \}$.

3. Inductive Step Assume for e having $\text{depth}(e) < m > 0$, to show for e having

$\text{depth}(e) = m$. (3)

$e = \sigma(e_1, ..., e_n)$ $\sigma \notin \Omega$.

The proof goes the same way as for the basis step except that we use the models of the data type D' that has the operation σ. ∎

For modified type algebras, we are interested in ground terms whose interpretations are not undefined. It can be shown for behaviorally equivalent type algebras A_1 and A_2 that if for some ground term e, $e|_{A_1}$ is undefined, then $e|_{A_2}$ is also undefined and vice versa. ∎

Thm. 2.6 For behaviorally equivalent A_1 and A_2, for any ground terms e_1 and e_2 of type

$\mathbf{D}''$, $\{ [e_1 | \mathbf{A}_1] \} = \{ [e_2 | \mathbf{A}_1] \} \Leftrightarrow \{ [e_1 | \mathbf{A}_2] \} = \{ [e_2 | \mathbf{A}_2] \}$.

Proof From the above two theorems and the fact that $\mathbf{A}_1/\mathbf{E}_1$ and $\mathbf{A}_2/\mathbf{E}_2$ are isomorphically equivalent, the statement is immediate. ∎

Appendix III - Proofs of Theorems in Chapter 4

This appendix contains proofs of various theorems in Chapter 4.

1. Specifications without Nondeterminism and without Exceptional Behavior

Thm. 4.1 Every constructor ground term e of type **Set-Int'** is equivalent by equational reasoning to a ground term e' not having any occurrence of **Remove**, i.e., the equation '$e \equiv e'$' $\in$ **EQ(Set-Int')**.

Proof For every constructor ground term e of type **Set-Int'**, there is a constructor ground term e' such that

(*) '$e \equiv e'$' $\in$ **EQ(Set-Int')** $\wedge$ **#re**(e') = 0,

where **#re**(e) gives the number of occurrences of the operation symbol **Remove** in e. Similarly, the function **#in** gives the number of occurrence of the operation symbol **Insert** in a term. We show (*) by induction on **#re**(e).

Basis **#re**(e) = 0,

The above statement trivially holds, because e' is same as e.

Inductive Step Assume the statement holds for e such that **#re**(e) < k,

show for **#re**(e) = k.

Consider the outermost subterm e_1 in e such that e_1 = **Remove**(e_{11}, il). Clearly, **#re**(e_{11}) < k, so there is a subterm e'_{11} such that '$e_{11} \equiv e'_{11}$' $\in$ **EQ(Set-Int')** and **#re**(e'_{11}) = 0. Thus we have '$e_1 \equiv$ **Remove**(e'_{11}, il)' $\in$ **EQ(Set-Int')**. We show that (*) holds for **Remove**(e'_{11}, il) by induction on **#in**(e'_{11}).

Basis **#in**(e'_{11}) = 0.

'$e_1 \equiv$ **Remove(Null, il)**

$\equiv$ **Null'** $\in$ **EQ(Set-Int')** using Axiom 1.

e' is obtained by substituting **Null** for e_1 in e.

Inductive Step Assume the above holds for $\#\mathbf{in}(e'_{11}) < m$,

to show for e'_{11} having m **Insert**'s.

e'_{11} = **Insert**(e_{21}, i2), so

'$e_1 \equiv$ **Remove(Insert**(e_{21}, i2), i1)' $\in$ **EQ(Set-Int')**.

There are two cases.

Case 1 i1 = i2

'$e_1 \equiv$ **Remove**(e_{21}, i1)' $\in$ **EQ(Set-Int')**. Axiom 2.

By the inductive step, there is an e'_{21} such that

'**Remove**(e_{21}, i1) $\equiv e'_{21}$ ' $\in$ **EQ(Set-Int')** and $\#\mathbf{re}(e'_{21}) = 0$.

So, '$e_1 \equiv e'_{21}$ ' $\in$ **EQ(Set-Int')**.

We get e' by replacing e_1 by e'_{21}.

Case 2 ~ i1 = i2

'$e_1 \equiv$ **Insert(Remove**(e_{21}, i1), i2)' $\in$ **EQ(Set-Int')**. Axiom 2.

By the inductive step, there is a e'_{21} such that

'**Remove**(e_{21}, i1) $\equiv e'_{21}$ ' $\in$ **EQ(Set-Int')**, and thus '$e_1 \equiv$ **Insert**(e'_{21}, i2)' $\in$ **EQ(Set-Int')**.

We get e' by replacing e_1 by **Insert**(e'_{21}, i2). ∎

Thm. 4.4 If a specification S is sufficiently complete, then S is behaviorally complete.

Proof If S is inconsistent, then since $F(S) = \varnothing$, so S is trivially behaviorally complete.

If S is consistent, we show that a sufficiently complete S is also behaviorally complete by contradiction.

Suppose S is not behaviorally complete, so there exists two reduced algebras $\mathbf{A}_1$ and $\mathbf{A}_2$ in $F(S)$ that are not isomorphically equivalent w.r.t $\{ \mathbf{P}_\sigma \mid \sigma \in \Omega \}$. Without any loss of generality, we can assume that $\mathbf{A}_1$ and $\mathbf{A}_2$ share the same domain corresponding to a defining type, so for each $D' \in \Delta$, $\Phi_{D'}$ is the identity function. Since every constructor is deterministic, there is a unique mapping $\Phi_D : \mathbf{V}^1_D \rightarrow \mathbf{V}^2_D$ which can possibly satisfy the following for every σ in Ω.

for each set of values $v_1, \ldots, v_n$, such that $\mathbf{P}_\sigma[x_1/v_1, \ldots, x_n/v_n]\|_{\mathbf{A}_1} = \mathbf{T}$,

(*) $\quad \Phi_{D'}(\mathbf{f}^1_\sigma(v_1, \ldots, v_n)) = \mathbf{f}^2_\sigma(\Phi_{D_1}(v_1), \ldots, \Phi_{D_n}(v_n))$.

If $\mathbf{A}_1$ and $\mathbf{A}_2$ are not isomorphically equivalent w.r.t. $\{ P_\sigma \mid \sigma \in \Omega \}$, this means that there must exist an observer σ and a set of values $v_1, \ldots, v_n$ such that $P_\sigma[x_1/v_1, \ldots, x_n/v_n]\|_{\mathbf{A}_1}$ holds and (*) is not satisfied.

Using the minimality property, we can construct a legal ground term $\sigma(e_1, \ldots, e_n)$ of type $D' \in \Delta$, where D' is the range of σ, and for each $1 \leq i \leq n$, e_i is the ground term whose interpretation is v_i in $\mathbf{A}_1$. Since S is sufficiently complete, there exists a ground term e' of type D' not having any operation symbol of D and auxiliary function used in S such that '$\sigma(e_1, \ldots, e_n) \equiv e'$' $\in$ EQ(S). This means that

$$f^1_\sigma(e_1, \ldots, e_n)|_{\mathbf{A}_1} = f^2_\sigma(e_1, \ldots, e_n)|_{\mathbf{A}_2} = e'|_{\mathbf{A}_1},$$

because $\mathbf{A}_1$ and $\mathbf{A}_2$ are reduced algebras. This is in contradiction to (*) not being satisfied.

Hence the result. ∎

Thm. 4.6 For a consistent and sufficiently complete S, if any two legal ground terms e_1 and e_2 of type D are distinguishable by S, then '$e_1 \not\equiv e_2$' $\in$ DS(S).

Proof: e_1 and e_2 are distinguishable by S, means that for any $\mathbf{A} \in \mathcal{F}(S)$, $e_1|_{\mathbf{A}}$ and $e_2|_{\mathbf{A}}$ are distinguishable, i.e., there exists a term c(x) of type $D' \in \Delta$ with one free variable x of type D such that $c[x/v_1]|_{\mathbf{A}}$ is distinguishable from $c[x/v_2]|_{\mathbf{A}}$ in $\mathbf{A}$.
Using the above fact, we prove the theorem by induction on specifications.

Basis Specifications with no defining types.

Case 1 **Bool**

'T $\not\equiv$ F' $\in$ DS(**Bool**). Every ground term of type **Bool** is equivalent to either T or F, so the theorem holds.

Case 2 D other than **Bool**

All ground terms are observable equivalent, so the theorem holds.

Inductive Step Assume the above statement for the specification S' of a data type D' used in the specification S of D. To show for S.

We can prove by contradiction that '$e_1 \not\equiv e_2$' $\in$ DS(S) as follows:

Assume $e_1 \equiv e_2$

then $c[x/e_1] \equiv c[x/e_2]$,

since S is sufficiently complete, there exists ground terms e_1' and e_2' of type D' such that e_1', e_2' do not have any occurrence of an operation symbol of D, and '$e_1 \equiv e_1'$' $\in$ EQ(S) and '$e_2 \equiv e_2'$' $\in$ EQ(S), so we have '$e_1' \equiv e_2'$' $\in$ EQ(S). Since e_1', e_2' are distinguishable by S', by inductive hypothesis, '$e_1' \not\equiv e_2'$' $\in$ DS(S'), so '$e_1' \not\equiv e_2'$' is also in DS(S). This is a contradiction, as S is consistent. So, '$e_1 \not\equiv e_2$' $\in$ DS(S). ∎

2. Specifications with Exceptional Behavior and without Nondeterminism

Thm. 4.9 Every legal constructor ground term e of type **Stk-Int** such that '$\mathbf{N?_{Stk\text{-}Int}}(e) \equiv$ T' $\in$ EQ(Stk-Int), is equivalent by equational reasoning to another legal constructor ground term e' having only **Null** and **Push**, i.e., if '$\mathbf{N?_{Stk\text{-}Int}}(e) \equiv$ T' $\in$ EQ(Stk-Int), then '$e \equiv e'$' $\in$ EQ(Stk-Int).

Proof Proof is similar to that of Theorem 4.1 above.

Let **#po** and **#rep** be the functions on terms computing number of occurrences of **Pop** and **Replace** respectively. We show by induction on **#po**(e) + **#rep**(e) that

(*) if '$\mathbf{N?_{Stk\text{-}Int}}(e) \equiv$ T' $\in$ EQ(Stk-Int), then there exists an e' such that '$e \equiv e'$' $\in$ EQ(Stk-Int) and **#po**(e') + **#rep**(e') = 0.

Basis **#po**(e) + **#rep**(e) = 0,

e serves as e'.

Inductive Step Assume (*) above for the case **#po**(e) + **#rep**(e) < k,

to show for **#po**(e) + **#rep**(e) = k.

Consider the outermost subterm e_1 in e having **Pop** or **Replace** as the outermost operation. It is obvious that if '$\mathbf{N?_{Stk\text{-}Int}}(e) \equiv$ T' $\in$ EQ(Stk-Int), then '$\mathbf{N?_{Stk\text{-}Int}}(e_1) \equiv$ T' $\in$ EQ(Stk-Int).

Case 1 $e_1 = \mathbf{Pop}(e_{11})$

Since '$\mathbf{N?_{Stk\text{-}Int}}(e_{11}) \equiv$ T' $\in$ EQ(Stk-Int), by inductive step, there exists an e_{11}' such that '$e_{11} \equiv e_{11}'$' $\in$ EQ(Stk-Int) and **#po**(e_{11}') + **#rep**(e_{11}') = 0.

Since '$\mathbf{N?_{Stk\text{-}Int}}(e_1) \equiv$ T' $\in$ EQ(Stk-Int), e_{11}' is not Null, and so $e_{11}' = \mathbf{Push}(e_{21}, i)$.

Thus '$e_1 \equiv \mathbf{Pop}(\mathbf{Push}(e_{21}, i)) \equiv e_{21}$' $\in$ EQ(Stk-Int) Axiom 1.

By replacing e_1 by e_{21} in e, we get the required e'.

Case 2 $e_1 = \mathbf{Replace}(e_{11}, i1)$

Since '$\mathbf{N?}_{\mathbf{Stk\text{-}Int}}(e_{11}) \equiv \mathbf{T}$' $\in$ EQ(Stk-Int), by inductive step, there exists an e'_{11} such that '$e_{11} \equiv e'_{11}$' $\in$ EQ(Stk-Int) and $\#\mathbf{po}(e'_{11}) + \#\mathbf{rep}(e'_{11}) = 0$.

Since '$\mathbf{N?}_{\mathbf{Stk\text{-}Int}}(e_1) \equiv \mathbf{T}$' $\in$ EQ(Stk-Int), e'_{11} is not **Null**, and so $e'_{11} = \mathbf{Push}(e_{21}, i2)$.

Thus $e_1 \equiv \mathbf{Replace}(\mathbf{Push}(e_{21}, i2), i1)$

$\equiv \mathbf{Push}(\mathbf{Pop}(\mathbf{Push}(e_{21}, i2)), i1)$ Axiom 3

$\equiv \mathbf{Push}(e_{21}, i1)$ Axiom 1

So '$e_1 \equiv \mathbf{Push}(e_{21}, i1)$' $\in$ EQ(Stk-Int).

By replacing e_1 in e by $\mathbf{Push}(e'_{21}, i1)$, we get the required e'. ∎

Thm. 4.12 If a specification S is sufficiently complete, then S is behaviorally complete.

Proof If S is inconsistent, then since $F(S) = \emptyset$, so S is trivially behaviorally complete.

If S is consistent, we show that a sufficiently complete S is behaviorally complete by contradiction.

Suppose S is not behaviorally complete, so there exists two reduced algebras $\mathbf{A}_1$ and $\mathbf{A}_2$ in $F(S)$ such that for every $D' \in \Delta$, the domain corresponding to D' in $\mathbf{A}_1$ and $\mathbf{A}_2$ are defined by isomorphically equivalent algebras in $F(S')$, where S' is a specification of D', and $\mathbf{A}_2$ is not partially isomorphically embeddable w.r.t. S in $\mathbf{A}_1$. Without any loss of generality, we can assume that $\mathbf{A}_1$ and $\mathbf{A}_2$ share the same domain corresponding to a defining type, so for every $D' \in \Delta$, $\Phi_{D'}$ is the identity function. Since every constructor is deterministic, there are unique one to one partial functions $\Phi_D : V_D^1 \rightarrow V_D^2$ and $\Phi_{EXV} : EXV \rightarrow EXV$ which can possibly satisfy the requirements for $\mathbf{A}_2$ to be partially isomorphically embeddable in $\mathbf{A}_1$ (see Def. 3.13 of isomorphic embeddability in Section 3.5). The first two requirements there can be easily satisfied. The third requirement is complex and is restated below:

For every operation $\sigma \in \Omega$, for every set of values $v_1, \ldots, v_n$, such that $\Phi_{D_i}(v_i)$ is defined for each $1 \leq i \leq n$, and $\mathbf{P}_\sigma[x_1/v_1, \ldots, x_n/v_n]\|_{\mathbf{A}_1} = \mathbf{T}$,

(a) if f^1_σ signals an exception value $ex(v'_1, \ldots, v'_m)$ specified to be optional by S on the

input $v_1, \ldots, v_n$, then the associated condition $O(x_1, \ldots, x_n)$ holds for $v_1, \ldots, v_n$, and

$f^2_\sigma(\Phi_{D_1}(v_1), \ldots, \Phi_{D_n}(v_n))$ either signals $ex(\Phi_{D_1'}(v_1'), \ldots, \Phi_{D_m'}(v_m'))$ or returns $\Phi_{D'}(v')$ for some v', or

(b) if $\Phi_{D_1'}(v_1'), \ldots, \Phi_{D_m'}(v_m')$ are defined and f^2_σ signals an exception value $ex(\Phi_{D_1'}(v_1'), \ldots, \Phi_{D_m'}(v_m'))$ specified to be optional by S on the input $\Phi_{D_1}(v_1), \ldots, \Phi_{D_n}(v_n)$, then the associated condition $O(x_1, \ldots, x_n)$ holds for $\Phi_{D_1}(v_1), \ldots, \Phi_{D_n}(v_n)$, and

$f^1_\sigma(v_1, \ldots, v_n)$ either signals $ex(v_1', \ldots, v_m')$ or returns v'; otherwise,

(c) $\Phi_{D'}(f^1_\sigma(v_1, \ldots, v_n)) = f^2_\sigma(\Phi_{D_1}(v_1), \ldots, \Phi_{D_n}(v_n))$ (*).

For A_2 not to be partially isomorphically embeddable in A_1, at least one of the above conditions is not satisfied. Supposingly if the condition (a) is not satisfied, we have

$f^2_\sigma(\Phi_{D_1}(v_1), \ldots, \Phi_{D_n}(v_n)) \neq ex(\Phi_{D_1'}(v_1'), \ldots, \Phi_{D_m'}(v_m'))$,

meaning that A_2 does not satisfy the optional exception condition for σ in S, which is contradictory to the assumption that $A_2 \in F(S)$. So, the condition (a) could not have been violated. Similarly, it can be shown that the condition (b) could not have been violated.

The violation of condition (c) is then the only possibility. In that case, for some $\sigma \in \Omega$,

(i) exactly one of the two sides of the equation (*) signals an exception,

(ii) different sides signal different exceptions, or

(iii) different sides return different values.

Using minimality property, we can construct a legal ground term $e = \sigma(e_1, \ldots, e_n)$ of type D', where for each $1 \leq i \leq n$, e_i is the ground term whose interpretation is v_i in A_1. The possibilities (i) and (ii) above are ruled out because of the following reasons:

For both (i) and (ii), the exception signalled by either side must be different from the optional exception. Since S is sufficiently complete, either '$N?_{D'}(e) \equiv T' \in EQ(S)$, or '$N?_{D'}(e) \equiv F' \in EQ(S)$. If '$N?_{D'}(e) \equiv T' \in EQ(S)$, then none of $e|_{A_1}$ and $e|_{A_2}$ can be an exception value, ruling out (i) and (ii). If '$N?_{D'}(e) \equiv F' \in EQ(S)$, then '*e* signals *ext*' $\in$ EQ(S) for some *ext* meaning that

$$e|_{A_1} = e|_{A_2} = ext|_{A_1}$$

again ruling out (i) and (ii).

The only possibility is (iii). Then *e* must be type $D' \in \Delta$, as if *e* is of type D, then

the definition of Φ_D ensures that the equation (*) is satisfied. We have either '$N?_D(e) \equiv T$' $\in$ EQ(S) or neither '$N?_D(e) \equiv T$' $\in$ EQ(S) nor '$N?_D(e) \equiv F$' $\in$ EQ(S). If '$N?_D(e) \equiv T$' $\in$ EQ(S), then there is a ground term e' without any operation symbol of D and auxiliary functions used in S such that '$e \equiv e'$' $\in$ EQ(S); so $e|_{A_1} = e|_{A_2} = e'|_{A_1}$ ruling out (iii). If neither '$N?_D(e) \equiv T$' $\in$ EQ(S) nor '$N?_D(e) \equiv F$' $\in$ EQ(S), then also there exists a ground term e' without any operation symbol of D and auxiliary functions used in S such that '$e \equiv e'$' $\in$ EQ(S $\cup$ { $N?_D(e) \equiv T$ }), which again rules out (iii) because of the reasons similar to the ones discussed above.

The above thus implies that A_2 is partially isomorphically embeddable in A_1.

Hence the result. ∎

Thm. 4.13 For a consistent and sufficiently complete S, if any two legal ground terms e_1 and e_2 of type D are distinguishable by S, then '$e_1 \not\equiv e_2$' $\in$ DS(S).

Proof: e_1 and e_2 are distinguishable by S, means that for any $A \in \mathcal{F}(S)$, $e_1|_A$ and $e_2|_A$ are distinguishable, i.e.,

(a) $e_1|_A$ is an exception value and $e_2|_A$ is a normal value,

(b) $e_1|_A$ and $e_2|_A$ are distinguishable exception values, or

(c) $e_1|_A$ and $e_2|_A$ are normal values and there exists a term c(x) of type D' $\in \Delta \cup \{ D \}$ with one free variable x of type D such that $c[x/v_1]|_A$ is distinguishable from $c[x/v_2]|_A$ in A.

Since S is sufficiently complete, it can be shown that if

(i) a ground term e interprets to an exception value in every algebra $A \in \mathcal{F}(S)$, then '$N?_D(e) \equiv F$' $\in$ EQ(S), and also

(ii) if e interprets to a normal value in every algebra $A \in \mathcal{F}(S)$, then '$N?_D(e) \equiv T$' $\in$ EQ(S).

Using the above facts, we prove the theorem by induction on specifications.

Basis Specifications with no defining types.

Case 1 **Bool**

'T $\not\equiv$ F' $\in$ DS(**Bool**). Every ground term of type **Bool** is equivalent to either T or F, so the theorem holds.

Case 2 D other than **Bool**

Subcase 1 S does not specify any operation to signal,

All ground terms are observable equivalent, so the theorem holds.

Subcase 2 S specifies operations to signal

Assume e_1 and e_2 are distinguishable by S, so there is one of the above three possibilities. We show in each case how '$e_1 \not\equiv e_2$' can be derived in DS(S).

(a) Since S is sufficiently complete, '$N?_D(c_1) \equiv$ F' $\in$ EQ(S) and '$N?_D(e_2) \equiv$ T' $\in$ EQ(S), and by the axiom (vii) in Subsection 4.3.3, '$e_1 \not\equiv e_2$' $\in$ DS(S).

(b) by sufficient completeness of S, using the axiom (vi) in Subsection 4.3.3 and repeatedly using the argument in case 2, we get '$e_1 \not\equiv e_2$' $\in$ DS(S).

(c) By the substitution property of the operations, and the sufficient completeness of S, we get '$e_1 \not\equiv e_2$' $\in$ DS(S), by the method of proof by contradiction.

Inductive Step Assume the above statement for the specification S' of a data type D' used in the specification S of D. To show for S.

Assume e_1 and e_2 are distinguishable by S. For the possibilities (a) and (b), the argument used in the basis step applies. For the third possibility, in addition to the case considered in the basis step, we have the case when the interpretations of e_1 and e_2 are distinguishable in **A** because of a computation c(x) returning distinguishable results of type D' $\in \Delta$. For this case also, we can prove by contradiction that '$e_1 \not\equiv e_2$' $\in$ DS(S) as follows:

Assume $e_1 \equiv e_2$

then $c[x/e_1] \equiv c[x/e_2]$, (*)

We have three subcases:

Subcase 1 Both sides of (*) interpret to a normal value in **A**.

Since S is sufficiently complete, there exists ground terms e_1' and e_2' of type D' such that e_1', e_2' do not have any occurrence of an operation symbol of D, and '$e_1 \equiv e_1'$',

'$e_2 \equiv e_2'$' $\in$ EQ(S), so we have '$e_1' \equiv e_2''$' $\in$ EQ(S). Since e_1', e_2' are distinguishable by S', by inductive hypothesis, '$e_1' \not\equiv e_2'$' $\in$ DS(S'), so '$e_1' \not\equiv e_2'$' is also in DS(S). This is a contradiction, as S is consistent. So, '$e_1 \not\equiv e_2$' $\in$ DS(S).

Subcase 2 One of the two sides of (*) interprets to a normal value.

Without any loss of generality, assume l.h.s. interprets to a normal value. By sufficient completeness of S, there is a e_1' such that '$e_1 \equiv e_1''$' $\in$ EQ(S), and there is an exception ground term *ext* such that 'e_2 signals *ext*' $\in$ EQ(S), so again, we have using the axioms, '$e_1 \not\equiv e_2$' $\in$ DS(S).

Subcase 3 Both sides of (*) interpret to distinguishable exception values.

Using the sufficient completeness of S, we can show using a similar argument that '$e_1 \not\equiv e_2$' $\in$ DS(S).

Hence the theorem. ∎

3. Specifications with Exceptional Behavior and Nondeterminism

Thm. 4.14 *f* and TR(*f*) are semantically equivalent.

Proof By induction on structure of *f*. We only need to show the basis step; the inductive step is straightforward because the symbols ~, ∨, and ∀ have the same interpretation. So, we have *f* as '$e_1 \equiv e_2$.' Consider an extended type algebra **A** of D in which *f* and TR(*f*) can be interpreted (i.e., **A** has an interpretation for every nondeterministic operation symbol σ and the corresponding auxiliary function symbol σ_p such that the interpretation of the auxiliary function is the relation computed by the interpretation of the nondeterministic operation symbol).

Case (a). *f* does not have any occurrence of a nondeterministic operation symbol.

TR(*f*) = f, so the statement trivially holds.

Case (b). Both e_1 and e_2 have occurrences of nondeterministic symbols:

It is obvious from the description of the procedure TR in Subsection 4.4.1 that the interpretation of '$e_1 \equiv e_2$' is equivalent to the interpretation of TR(*f*).

Case (c) Exactly one of e_1 and e_2 has occurrences of nondeterministic symbols: Again from

the description of **TR** in Subsection 4.4.1, the interpretation of '$e_1 \equiv e_2$' is equivalent to the interpretation of **TR**(f). ∎

Appendix IV - Specifications of Data Types used in Chapter 5

In this appendix, we give specifications of the data types **Null**, **Struct** $[n_1: D_1, \ldots, n_k: D_k]$, **Oneof** $[n_1: D_1, \ldots, n_k: D_k]$, and **Sequence-Int** used in Chapter 5. **Struct**, and **Oneof** are type schema. Below, we specify an instance of these schema assuming fixed but unspecified parameters, i.e., k as well as D_1, ..., D_k are fixed. Since the specification is given for an arbitrary k, we have used the '...' notation. The specification of any particular instance, such as **Oneof [empty: Null, pair: Pair]**, **Struct [car: Int, cdr: List-Int]** used in Chapter 5, can be given without using the '...' notation.

Figure A4.1. Specification of Null

Operations

Nil : → **Null**
Equal : **Null X Null** → **Bool** *as* **x1 = x2**

Axioms

Nil = Nil ≡ T

Figure A4.2. Specification of Struct $[n_1: D_1, \ldots, n_k: D_k]$

Struct $[n_1: D_1, \ldots, n_k: D_k]$ *as* **D**

Operations

Create : $D_1 \times \ldots \times D_k \rightarrow D$
Fetch_n_1 : $D \rightarrow D_1$
.
.
.
Fetch_n_k : $D \rightarrow D_k$
Replace_n_1 : $D \times D_1 \rightarrow D$
.
.
.
Replace_n_k : $D \times D_k \rightarrow D$
Equal : $D \times D \rightarrow$ **Bool** *as* **x1 = x2**

Axioms

Fetch_n_1(Create(x1, ..., xk)) ≡ x1
.
.
.
Fetch_n_k(Create(x1, ..., xk)) ≡ xk
Replace_n_1(Create(x1, ..., xk), y1) ≡ Create(y1, ..., xk)
.
.
.
Replace_n_k(Create(x1, ..., xk), yk) ≡ Create(x1, ..., yk)
Create(x1, ..., xk) = Create(y1, ..., yk) ≡ (x1 = y1) ∧ ... ∧ (xk = yk)

Figure A4.3. Specification of Oneof $[n_1: D_1, \ldots, n_k: D_k]$

Oneof $[n_1: D_1, \ldots, n_k: D_k]$ *as* D

Operations

Make_n_1 : $D_1 \rightarrow D$
.
.
.
Make_n_k : $D_k \rightarrow D$
Value_n_1 : $D \rightarrow D_1$
$\rightarrow$ wrong-tag
.
.
.
Value_n_k : $D \rightarrow D_k$
$\rightarrow$ wrong-tag
Is_n_1 : $D \rightarrow$ Bool
.
.
.
Is_n_k : $D \rightarrow$ Bool
Equal : $D \times D \rightarrow$ Bool *as* x1 = x2

Restrictions

$\sim$ Is_n_1(x) $\Rightarrow$ Value_n_1(x) signals wrong-tag
.
.
.
$\sim$ Is_n_k(x) $\Rightarrow$ Value_n_k(x) signals wrong-tag

Axioms

Value_n_1(Make_n_1(x1)) $\equiv$ x1
.
.
.
Value_n_k(Make_n_k(xk)) $\equiv$ xk
Is_n_1(Make_n_1(x1)) $\equiv$ T
.
.
.
Is_n_1(Make_n_k(xk)) $\equiv$ F
.
.
.
Is_n_k(Make_n_1(x1)) $\equiv$ F

.
.
.

Is_n_k(Make_n_k(xk)) $\equiv$ T

Make_n_1(x1) = Make_n_1(y1) $\equiv$ x1 = y1

.
.
.

Make_n_1(x1) = Make_n_k(yk) $\equiv$ F

.
.
.

Make_n_k(xk) = Make_n_1(y1) $\equiv$ F

.
.
.

Make_n_k(xk) = Make_n_k(yk) $\equiv$ xk = yk

x = y $\equiv$ y = x

Figure A4.4. Specification of Sequence-Int

Sequence-Int *as* **SI**

Operations

New : → **SI**
Addl : **SI X Int** → **SI**
Addh : **SI X Int** → **SI**
Concat : **SI X SI** → **SI** *as* **x1 • x2**
Subseq : **SI X Int X Int** → **SI**
→ **bounds**
→ **negative-size**
Fill : **Int X Int** → **SI**
→ **negative-size**
Fetch : **SI X Int** → **Int** *as* **x[i]**
→ **bounds**
Bottom : **SI** → **Int**
→ **bounds**
Top : **SI** → **Int**
→ **bounds**
Reml : **SI** → **SI**
→ **bounds**
Remh : **SI** → **SI**
→ **bounds**
Size : **SI** → **Int**
Empty : **SI** → **Bool**
Replace : **SI X Int X Int** → **SI**
→ **bounds**
Index : **SI X Int** → **Int**
→ **element-not-in**
Member : **SI X Int** → **Bool**
Equal : **SI X SI** → **Bool** *as* **x1 = x2**

Restrictions

(i1 < 1 ∨ i1 > (Size(s) + 1)) ⇒ Subseq(s, i1, i2) signals bounds
(~ (i1 < 1 ∨ i1 > (Size(s) + 1)) ∧ (i2 < 0)) ⇒ Subseq(s, i1, i2) signals negative-size
i < 0 ⇒ Fill(i, j) signals negative-size
(i < 1 ∨ i > Size(s)) ⇒ Fetch(s, i) signals bounds
Size(s) = 0 ⇒ Bottom(s) signals bounds
Size(s) = 0 ⇒ Top(s) signals bounds
Size(s) = 0 ⇒ Reml(s) signals bounds
Size(s) = 0 ⇒ Remh(s) signals bounds
(i < 1 ∨ i > Size(s)) ⇒ Replace(s, i, j) signals bounds
~ Member(s, j) ⇒ Index(s, j) signals element-not-in

Axioms

Addl(New, j) ≡ Addh(New, j)
Addl(Addh(s, j1), j2) ≡ Addh(Addl(s, j2), j1)

$s \cdot New \equiv s$

$s1 \cdot Addh(s2, j) \equiv Addh(s1 \cdot s2, j)$

$Subseq(s, i1, 0) \equiv New$

$Subseq(Addh(s, j), i1, i2 + 1) \equiv$ if $(i1 + i2) < (Size(s) + 1)$ then $Subseq(s, i1, i2 + 1)$
else if $(i1 + i2) = (Size(s) + 1)$ then $Addh(Subseq(s, i1, i2), j)$
else $Subseq(Addh(s, j), i1, Size(s) - i1 + 2)$

$Fill(0, j) \equiv New$

$Fill(i + 1, j) \equiv Addh(Fill(i, j), j)$

$Fetch(Addh(s, j), i) \equiv$ if $i = Size(s) + 1$ then j else $Fetch(s, i)$

$Bottom(s) \equiv Fetch(s, 1)$

$Top(s) \equiv Fetch(s, Size(s))$

$Reml(s) \equiv Subseq(s, 2, Size(s)-1)$

$Remh(s) \equiv Subseq(s, 1, Size(s)-1)$

$Size(New) \equiv 0$

$Size(Addh(s, j)) \equiv Size(s) + 1$

$Empty(New) \equiv T$

$Empty(Addh(s, j)) \equiv F$

$Member(New, j) \equiv F$

$Member(Addh(s, j1), j2) \equiv$ if $j1 = j2$ then T else $Member(s, j2)$

$Replace(Addh(s, j1), i, j2) \equiv$ if $i = Size(s) + 1$ then $Addh(s, j2)$ else $Addh(Replace(s, i, j2), j1)$

$Fetch(s, Index(s, j)) \equiv j$

$x = x \equiv T$

$x = y \equiv y = x$

$New = Addh(s, j) \equiv F$

$Addh(s1, j1) = Addh(s2, j2) \equiv (j1 = j2) \wedge (s1 = s2)$