
Data Abstraction Transformations

by

Mark A. Ardis

Dissertation submitted to the Faculty of the Graduate School of the University of Maryland in partial fulfillment of the requirements for the degree of Doctor of Philosophy 1980

AIR FORCE OFFICE OF SCIENTIFIC RESEARCH (AFSC)

NOTICE OF THE NOTICE TO DDC

This technical report has been reviewed and is approved for public release and is distribution is unlimited.
A. D. Blose
Technical Information Officer
ABSTRACT

Title of Dissertation: Data Abstraction Transformations

Mark Alan Ardis, Doctor of Philosophy, 1980

Dissertation directed by: Dr. Richard G. Hamlet
Associate Professor
Department of Computer Science

A data abstraction is a collection of sets together with a collection of operations. Methods exist for specifying and for implementing data abstractions. The central question for any particular example is whether the semantics of each of these two methods corresponds with the intended abstraction.

An algebraic comparison of data abstraction specifications and implementations is presented. It is shown that the specified and implemented abstractions always overlap and have a common (lattice) structure that is valuable in understanding the modification of code and specification.
A new specification technique, *table specification*, is proposed that emphasizes the underlying congruence-class structure of data abstractions. Algorithms to transform tables are defined.
ACKNOWLEDGEMENTS

I would like to express my thanks to my advisor, Dr. Richard Hamlet, for his guidance and support during my work on this dissertation. If any of the ideas expressed in this document make sense it is due to his insatiable appetite for knowledge and his patience to digest it properly.

It is a pleasure to acknowledge the encouragement and advice I received from Dr. John Gannon. I am also indebted to Dr. Victor Basili and Dr. Marvin Zelkowitz for their help in my research.

I am especially grateful to my wife, Cheryl Dugas, whose patience and support were always more than I deserved. Her technical and emotional assistance made the completion of this work possible.

This work was supported by a grant from the Air Force Office of Scientific Research (AFOSR). Computer time was provided in part by the Computer Science Center of the University of Maryland.
TABLE OF CONTENTS

1. Introduction 1

2. Data Abstractions 3

3. Word Algebra 5
   3.1 Algebras 6
   3.2 Semantic Interpretations 14
   3.3 Lattices 19

4. Specifications 24
   4.1 Algebraic Specifications 25
   4.2 Correctness 28
   4.3 Inequalities 39

5. Implementations 49
   5.1 Classes 50
   5.2 Correctness 53

6. Specification/Implementation Intersection 63

7. Table Specifications 69
   7.1 T-Grammars 72
   7.2 Constructors 81
   7.3 Tables 84

8. Transformations of Tables 94
   8.1 Benign Transformations 98
   8.2 Rewrite Sets 114
   8.3 Adding Axioms 127

9. Implementations and Tables 141
   9.1 Program Partitions 142
   9.2 Table Partitions 146
   9.3 Relative Correctness 148

10. Summary 151

11. Bibliography 153
LIST OF FIGURES

Figure 3-1. Signature of natural numbers 9
Figure 3-2. Signature of Stack 10
Figure 3-3. Part of $L_w(\{\text{Nat}_2, \text{Zero}, \text{Succ}\})$ 23
Figure 4-1. Specification of Natl2 26
Figure 4-2. Specification of Stack 27
Figure 4-3. Examples of derivations 30
Figure 4-4. The lattice $L_s$ for Natl2 38
Figure 4-5. Specification of Natl2 with inequalities 41
Figure 4-6. Semilattice $S_s$ for Natl2 with inequalities 46
Figure 4-7. Initial vs. Final Algebra 48
Figure 5-1. Implementation Natl8 51
Figure 5-2. Implementation of Stack 52
Figure 5-3. Correctness of implementations 56
Figure 5-4. The lattice $L_c$ for Natl8 61
Figure 6-1. The lattices $L_s$, $L_c$ and SL for Nat 67
Figure 7-1. Examples of form(w) and rform(w) 74
Figure 7-2. The Nat-grammar of Stack 76
Figure 7-3. The Nat-grammar of Natplus 80
Figure 7-4. A type with three nontrivial sets of constructors 82
Figure 7-5. A table specification of Stack 86
Figure 7-6. The table specification of $W_c$ of Stack 92
Figure 8-1. Table specification of Stack with variable columns  

Figure 8-2. Example of function addition (Algorithm F)  

Figure 8-3. The result of deleting Pop from Figure 8-1  

Figure 8-4. Example of row expansion (Algorithm X)  

Figure 8-5. Example of row contraction (Algorithm C)  

Figure 8-6. The standard order on Stack  

Figure 8-7. Example of rewrite set completion (Algorithm R)  

Figure 8-8. Example of non-convergent rewrite set  

Figure 8-9. Example of axiom addition (Algorithm A)
1. Introduction

Software maintenance is a process of changing existing implementations to meet new specifications. It is often easy to change a specification, but difficult to make the corresponding change to its implementation. In this thesis we examine a cause for this difficulty in the domain of data abstractions.

A method for specifying data abstractions, called the algebraic or axiomatic specification technique has been proposed by [Zilles 74], [Guttag 75] and [ADJ 75b]. In this method data abstractions are modeled by heterogeneous algebras. We show that these algebras have a lattice structure that is shared by models of implementations that share the same syntax.

Each algebra has an inner structure, congruence classes, that is useful in studying changes to specifications and implementations. A new method of specification, table specification, is proposed which emphasizes this inner structure.

Chapter 2 introduces the domain of interest, data abstractions. Distinctions and relationships between data abstractions, specifications, and implementations are
introduced. Chapter 3 reviews some algebraic concepts and presents some properties of the structure of word algebras. The word algebra structure is used in Chapters 4 through 6 to examine specifications and implementations. Table specifications and their transformations are covered in Chapters 7 through 9.
2. Data Abstractions

**DEFINITION 2.1** - A data abstraction is a collection of sets with a collection of operations on those sets. Each set in a data abstraction is called a domain or a data type. 

Several programming languages provide facilities for defining data abstractions as program objects [Dahl et al. 70], [Liskov et al. 77], [Wulf, London & Shaw 76], [Gannon & Rosenberg 79]. A class is a program object that may be viewed as a data abstraction. The correspondence between classes and data abstractions is described by an interpretation, a mapping of program objects and operations onto abstract objects and operations.

As we will show in Chapter 5, for each class there is at least one corresponding data abstraction. There may be one abstraction that best captures the intentions of the programmer, the creator of the class. This does not rule out the existence of other abstractions, to which the class corresponds under other interpretations. A class describes a collection of data abstractions in the sense that some, but not all data abstractions may be interpretations of the class.
Another way to describe a collection of data abstractions is by a specification. In particular, we will be concerned with algebraic specifications. An algebraic specification has syntax that must obey certain restrictions in order to be well-formed. Every well-formed algebraic specification corresponds to at least one data abstraction, by means of an interpretation of the symbols in the specification onto the objects and operations of the data abstraction. As with classes, there may be one data abstraction that best captures the intentions of the specifier, the creator of the specification, but there may be other data abstractions to which the specification corresponds.

Because classes and specifications both describe data abstractions, it is possible to compare them to one another. In particular, a given class and a given specification may describe the same collection of data abstractions. The intersection of these collections is a measure of the correspondence between the class and the specification. Similarly, one may make the same type of comparison between two specifications or between two classes.

The algebraic structure of a data abstraction may be used to partition each set in the abstraction into blocks. These blocks will be used to explain similarities between different specifications and classes.
3. Word Algebra

By reserving the term data abstraction for abstract, intuitive objects we avoid confusing it with another idea: the syntax of a class or a specification. In other words, the meaning of a class or a specification is a data abstraction. In any particular case, a data abstraction arises from the syntax of a class or a specification. However, we can also study data abstractions apart from their defining classes or specifications.

Collections of sets and operations on those sets may be described by a mathematical formalism: heterogeneous (many-sorted) algebra. In fact, common abstract algebra suffices to describe most phenomena. The only need for heterogeneous algebra is to extend concepts to objects with more than one set. Because the number of sets is always finite, these extensions are straightforward.
3.1. Algebras

**DEFINITION 3.1** - An algebra is a pair \((D, M)\),

where \(D\) is a collection of domains and \(M\) is a

collection of mappings from cross-products of sets

in \(D\) to sets in \(D\). When the operations are

understood from context, we omit \(M\). ***

For example, the natural numbers may be described as an

algebra with one domain, the set of natural numbers, and two

operations, Zero and Successor. An example with more

than one domain is the ubiquitous stack. Two domains,

natural numbers and stacks, are needed. The usual

operations are Newstack, Push, Pop, Top, and the

natural number operations.

It is clear that names are needed to describe the sets

and operations, but that names are not enough. Two algebras

may have the same names, but the operations may do different

things. Still, the two algebras have something in common.

We capture this syntactic idea by the term signature.
DEFINITION 3.2 - A signature is a triple (S,F,V), where S is a finite, nonempty set of set names, called types; F is a finite, nonempty set of function names and their arities: the names of the types that make up their domains and ranges; and V is a finite set of variables \( V_i \). When \( V \) is empty we omit it from the signature.

For each function \( f: S_1 \times \ldots \times S_n \rightarrow S_i \) the product set \( S_1 \times \ldots \times S_n \), sometimes written \((S_1, \ldots, S_n)\), is its domain arity. For constants \( f: \rightarrow S_i \), the empty tuple ( ) is used to denote the empty domain arity. Each variable \( V_i \) has a type, an element from \( S \). The domain of a variable, written \( \text{dom}(V_i) \), is a particular subset of its type. Every algebra has a signature. Two algebras with the same signature are similar algebras. ***

DEFINITION 3.3 - The order of a signature is the maximum of the number of arguments of each function in the signature. The T-order of a signature is the maximum of the number of arguments of type T of each function in the signature. ***

It is often useful to build up a data abstraction from
its components. Each type in the signature is specified (implemented) separately, in a sequence of specifications (classes). Types that have been specified (implemented) earlier in the sequence may be referenced in the new specification (class).

**DEFINITION 3.4** - a **hierarchical signature** is a signature whose domains and functions are divided into two classes: old and new. Only one new domain is allowed, called the **type-of-interest**, or TOI. ***

**DEFINITION 3.5** - A **hierarchical specification** (respectively, **class**) is a sequence of specifications (classes) with hierarchical signatures, in which each old domain or function in any specification (class) is a new domain or function in some previous specification (class) in the sequence. ***

The hierarchical signatures of the natural numbers and stack are shown in Figures 3-1 and 3-2.
Types: Natural

Functions: Zero: --> Natural
           Succ: Natural --> Natural

Figure 3-1. Signature of natural numbers
Types: Nat (Old)  
Stack

Functions: Zero: $\rightarrow$ Nat (Old)  
Succ: Nat $\rightarrow$ Nat (Old)

Newstack: $\rightarrow$ Stack  
Push: Nat x Stack $\rightarrow$ Stack  
Pop: Stack $\rightarrow$ Stack  
Top: Stack $\rightarrow$ Nat

Figure 3-2. Signature of Stack
Two signatures are the same if and only if the names of the sets and operations are the same and the operations have the same arities. On the other hand, assigning new names (e.g., changing each name "abc" to "abc2") consistently to one of the signatures does not change the fact that the two signatures describe the same structure. We will ignore such distinctions between signatures and treat two signatures as if they were the same if the only difference between them is such a renaming.

Implicit in this discussion of signatures is the notion of possible algebras they describe. In general, a signature may be shared by many different algebras. However, there is a natural unique algebra for each signature. If each operation produced values in its range that were different from all the values produced by all the other operations, and if a new value was produced for each new set of input values, then this algebra would be the most "general" algebra for that signature. That is, it would have as many distinct values as possible. Note that this definition allows the domains to contain values that are not in the ranges of any operations. We dispense with such values by the following:

CONVENTION 3.6 - The elements of domains of an algebra are restricted to those values that are results of operations in the algebra.
This is an intuitive restriction for program objects. The only values that can exist are those that arise from operations. (We treat initialization as a constant operation.) We now have a unique algebra to associate with each signature, the constant word algebra.

**DEFINITION 3.7 -** The word algebra $W_{v}(S,F,V)$ of a signature $(S,F,V)$ is the set of all words formed as follows:

1. For each function $f: S_i \rightarrow S_i$, the symbol "f" is a word of type $S_i$.
2. Each variable symbol $V_i$ with domain $S_i$ is a word of type $S_i$.
3. If $f: S_1 \times \ldots \times S_n \rightarrow S_i$ is a function in $F$ and $w_1, \ldots, w_n$ are words of types $S_1, \ldots, S_n$, then $f(w_1, \ldots, w_n)$ is a word of type $S_i$.

A word containing no variable symbols is called a constant. The constant word algebra $W_{c}(S,F,V)$ is the set of all constants in $W_{v}(S,F,V)$. *****

For example, the constant word algebra of the signature of the natural numbers contains constants:

Zero
Succ(Zero)
Succ(Succ(Zero)), etc.

The constant word algebra of the stack signature contains
such constants of type Stack as:

Newstack

Push(Newstack, Zero)

Pop(Push(Newstack, Zero)), etc.

The same algebra contains constants of type Nat:

Zero

Succ(Zero)

Top(Push(Newstack, Zero)), etc.

**DEFINITION 3.8** - Let \((S,F,V)\) be a signature.

An instance \(w'\) of a word \(w \in W_v(S,F,V)\) is an element of \(W_c(S,F,V)\) obtained by consistently substituting constants for variables in \(w\). An instance \((w_1', \ldots, w_n')\) of an \(n\)-tuple \((w_1, \ldots, w_n)\) is obtained by using the same substitution scheme for variables in each word \(w_1, \ldots, w_n\). ***

As a special case of the definition, for each variable \(V_i\) of type \(T_i\) in a signature \((S,F,V)\), the set of all instances of \(V_i\) is the set of all constants of type \(T_i\) in \(W_c(S,F,V)\).
3.2. Semantic Interpretations

As can be seen in the stack example, the constant word algebra may contain more distinct values than intended. In particular, Newstack and Pop(Push(Newstack, Zero)) are probably intended to be the same value. We can accomplish this by defining equivalence relations on the domains.

**DEFINITION 3.9** - An equivalence relation \( \sim \) over a set \( S \) is a binary relation satisfying the following properties, for all \( x, y \) and \( z \) in \( S \):

1. \( x \sim x \). (Reflexive)
2. If \( x \sim y \) then \( y \sim x \). (Symmetric)
3. If \( x \sim y \) and \( y \sim z \) then \( x \sim z \). (Transitive)

The subset of \( S \) of all elements equivalent to \( x \), called the equivalence class of \( x \), is denoted by \( [x] \). ***

The first two laws of equivalence relations are obviously needed for any relation that is meant to capture equality. Transitivity ensures that if two values are equal, it does not matter how they were produced. For example, if Pop(Push(Newstack, Zero)) is equal to Newstack, and (Pop(Push(Newstack, Succ(Zero)))) is equal to Newstack, then Pop(Push(Newstack, Zero)) is equal to...
Pop(Push(Newstack, Succ(Zero))). The history of production of a value does not matter.

It would not be correct to allow any set of equivalence relations on domains to define equality of values. The relations on each domain must be consistent with one another. Further, if two values are equal, then they should yield equal results when passed as parameters to the same operation. These two properties are captured by the following definition.

**DEFINITION 3.10** - A **congruence** on an algebra (D, M) is a set \( \{ \sim_i \} \) of equivalence relations, one relation defined on each set \( D_i \in D \), with the substitution property:

(4) For all functions \( f: D_1 \times \ldots \times D_n \rightarrow D_m \),

\[
x_1 \sim_i y_1 , \text{ where } i = 1, \ldots, n ,
\]

implies

\[
f(x_1, \ldots, x_n) \sim_m f(y_1, \ldots, y_n) .
\]

\(|x|\) denotes the congruence class of \( x \).

***

For example, in a congruence on stack,

\[
\text{Zero} = \text{Succ}(\text{Zero})
\]

implies

\[
\text{Push}(\text{Newstack}, \text{Zero}) = \text{Push}(\text{Newstack}, \text{Succ}(\text{Zero})) .
\]

Furthermore,
Zero = Succ(Zero)

implies

Succ(Zero) = Succ(Succ(Zero)) = ... 

Unfortunately, defining a congruence on an algebra does not change the number of elements in the domains of the algebra. Pop(Push(Newstack, Zero)) and Newstack may be in the same congruence class, but they are still different words. What we want is another algebra with one object for each class of equal words according to a congruence defined on \( W_C \). Such an algebra is uniquely defined for each congruence.

**DEFINITION 3.11** - A **quotient algebra** \((D/C, M)\) is the algebra formed from \((D, M)\) by substituting a congruence class of \( C_i \) for each set of elements equal under \( C_i \) in each set \( D_i \). For each function \( f: D_1 \rightarrow D_2 \) in the original algebra \((D, M)\), the new function \( f': D_1/C_1 \rightarrow D_2/C_2 \) is defined in the natural way: 

\[
f'(\{x\}) = \{f(x)\}.
\]

Where there is no confusion we reuse the old names for the new functions and drop the class brackets, writing \( f(x) \) for \( f'(\{x\}) \) and \( x \) for \( \{x\} \).

***

Suppose, for example, that we defined a congruence on the natural numbers by: Zero = Succ\(^2\)(Zero), where
\( f^n(x) \) is an abbreviation for \( f(f(\ldots f(x)\ldots)) \).

Note that several other equalities are implied, such as:

\[ \text{Succ}(\text{Zero}) = \text{Succ}^{13}(\text{Zero}). \]

It can be shown that there are only twelve classes of values, where all the values in each class are equal to one another. If we call this congruence \( \text{mod} \text{12} \), then we can define a quotient algebra \( (\mathbb{N}/\text{mod}12, \text{Zero}, \text{Succ}^3) \). The quotient algebra has one domain with twelve different values in it. The value produced by \( \text{Succ}(\text{Zero}) \) is the same value as produced by \( \text{Succ}^{13}(\text{Zero}) \).

Because every quotient algebra arises from some larger algebra by means of a congruence, there is a natural mapping between the two algebras. For every value in the large algebra there is a corresponding value in the quotient algebra that "behaves" in the same way with respect to the operations of the algebras. Every value in the quotient algebra corresponds to some value or set of values in the large algebra. This relationship is described by an **epimorphism** from the large algebra to the quotient algebra.
DEFINITION 3.12 - An algebra homomorphism, or just a homomorphism \( h: (D, M) \rightarrow (D', M') \) is a mapping between similar algebras (i.e., they have the same signature) that preserves the functions:

\[
h(f(w_1, \ldots, w_n)) = f(h(w_1), \ldots, h(w_n))
\]

for all elements \( w_i \) of \( D \) and all functions \( f \) in \( M \). An epimorphism is an onto homomorphism.

We state without proof a theorem of [Birkhoff & Lipson 70]:

THEOREM 3.13 - The set of all epimorphisms of an algebra \( A \) is completely determined by the set of all quotient algebras of \( A \). ***

Given a class or a specification, we can generate the constant word algebra of its signature. In general, \( W_c \) will be too large: the number of values intended will be smaller than the number of values in \( W_c \). \( W_c \) cannot be too small, given Convention 3.6 for algebras. The intended algebra of a class or specification, then, must be some quotient algebra of \( W_c \). We call an intended algebra, a semantic interpretation.

DEFINITION 3.14 - A semantic interpretation of a class or a specification is an epimorphism of the constant word algebra of the signature.
3.3. Lattices

Just as every quotient algebra is related to its original algebra by an epimorphism, some of the quotient algebras of a given algebra are related to one another by epimorphisms. For example, we could define a new congruence \( \text{mod}_4 \) on the natural numbers by the equation:

\[
\text{Zero} = \text{Succ}^4(\text{Zero}).
\]

The new algebra, \((\mathbb{N}/\text{mod}_4, \{\text{Zero}, \text{Succ}^3\})\), is the epimorphic image of the algebra \(\text{mod}_{12}\) under the following mapping:

\[
\begin{align*}
\text{Zero, Succ}^4(\text{Zero}), \text{Succ}^8(\text{Zero}) & \rightarrow \text{Zero} \\
\text{Succ}(\text{Zero}), \text{Succ}^5(\text{Zero}), \text{Succ}^9(\text{Zero}) & \rightarrow \text{Succ}(\text{Zero}) \\
\text{Succ}^2(\text{Zero}), \text{Succ}^6(\text{Zero}), \text{Succ}^{10}(\text{Zero}) & \rightarrow \text{Succ}^2(\text{Zero}) \\
\text{Succ}^3(\text{Zero}), \text{Succ}^7(\text{Zero}), \text{Succ}^{11}(\text{Zero}) & \rightarrow \text{Succ}^3(\text{Zero}).
\end{align*}
\]

On the other hand, the algebra \(\text{mod}_{13}\) defined by:

\[
\text{Zero} = \text{Succ}^{13}(\text{Zero})
\]

is not related to either \(\text{mod}_4\) or \(\text{mod}_{12}\). Such relationships are described by partially ordered sets.
DEFINITION 3.15 - A *partially ordered set*, or *poset*, \((X,\leq)\) is a set \(X\) with a relation \(\leq\) satisfying the following properties, for all \(x, y, z\) in \(X\):

1. \(x \leq x\). (Reflexive)
2. \(x \leq y\) and \(y \leq x\) implies \(x = y\). (Antisymmetric)
3. \(x \leq y\) and \(y \leq z\) implies \(x \leq z\). (Transitive)

***

A relation \(A \leq B\) on quotient algebras is always defined by the existence of an epimorphism from \(B\) to \(A\). The congruence classes of the domain algebra \(B\) are contained (in the set-theoretic sense) in the congruence classes of the range algebra \(A\). For example, the classes of \(\text{mod}12\) are contained in the classes of \(\text{mod}4\).

Some posets are closed. That is, there exists a value in the poset that is smaller than every value, and there exists a value in the poset that is larger than every other value. When this happens, the poset is called a *complete lattice*. 
DEFINITION 3.16 - A lattice is a partially ordered set in which every two elements have a least upper bound, called the join, and a greatest lower bound, called the meet, in the set. A complete lattice is a lattice in which every subset of L has a join and a meet in L. A sublattice is a subset L of a lattice M closed under the join and meet operations of M, operating on subsets of L. A lower semilattice is a partially ordered set in which every two elements have a meet. ***

The quotient algebras of \( W_c \) are closed under the ordering defined above, containment of congruence classes, because \( W_c \) is smaller (under the defined ordering) than every quotient algebra, and the trivial algebra, which has one value in each domain, is larger than every other quotient algebra. In fact, for all heterogeneous algebras we have the following theorem of [Birkhoff & Lipson 70]:

THEOREM 3.17 - The poset of all congruences on an algebra forms a complete lattice. ***

And, in particular, we have the following corollary:
COROLLARY 3.18 - The collection of semantic interpretations of a class or a specification forms a complete lattice, denoted by $L_w$. ***

As an example, part of the lattice of semantic interpretations of the natural numbers signature is given in Figure 3-3. Each box in the Figure represents a quotient algebra defined by the congruence described by the equation in the box. The entire lattice is infinite: there are an infinite number of quotient algebras for that signature. In fact, there are an infinite number of quotient algebras just below the trivial algebra: one for each prime number. Similarly, there are an infinite number of levels in the lattice: there are numbers that have an infinite number of powers of two, say. Nevertheless, there is one unique algebra at the very bottom of the lattice: the natural numbers. Every other algebra equates at least two words.
Figure 3-3. Part of $L_w(\{\text{Nat}^3\}, \{\text{Zero, Succ}^3\})$
4. Specifications

The purpose of this section is to relate algebraic specifications, syntactic objects, to semantic interpretations, semantic objects. The particular type of specifications considered here are essentially those of [Guttag 75], but without conditional axioms.
4.1. Algebraic Specifications

**DEFINITION 4.1** - An algebraic specification consists of:

(1) A signature $(S,F,V)$.

(2) A finite collection of axioms: pairs of words of the same type from $W_v(S,F,V)$, the two members of each pair separated by " = ".

Note that there may be an empty collection of axioms. ***

Figures 4-1 and 4-2 contain specifications of Nat12 and Stack, two data abstractions discussed in the previous chapter.
Types: Nat

Functions: Zero: \( \rightarrow \) Nat
Succ: Nat \( \rightarrow \) Nat

Axioms: \( \text{Succ}^{12}(\text{Zero}) = \text{Zero} \)

Figure 4-1. Specification of Nat12
Types: Nat (old)
Stack

Functions: Zero: \[\rightarrow\] Nat (Old)
Succ: Nat \[\rightarrow\] Nat (Old)

Newstack: \[\rightarrow\] Stack
Push: Nat x Stack \[\rightarrow\] Stack
Pop: Stack \[\rightarrow\] Stack
Top: Stack \[\rightarrow\] Nat

Variables: N: Nat
S: Stack

Axioms: \[\text{Succ}^{12}(\text{Zero}) = \text{Zero (Old)}\]
Pop(Push(N,S)) = S
Top(Push(N,S)) = N
Pop(Newstack) = Newstack
Top(Newstack) = Zero

Figure 4-2. Specification of Stack
4.2. Correctness

It is clear that the lattice of semantic interpretations contains more interpretations than are intended by the specifications. In particular, the semantic interpretation of the signature of Nat12 (Figure 4.1) that corresponds to the congruence \text{mod}13 (see Section 3.3) is in conflict with the axiom:

\[ \text{Zero} = \text{Succ}^{12}(\text{Zero}) \, . \]

Axioms are intended to be true statements about the objects described by the specification. To make this notion more precise, we define the collection of words from \( W_c \) that may be derived equal by a specification.
DEFINITION 4.2 - A derivation from a specification $S$ is a finite sequence of equations which may be formed as follows:

1. $w = w$, where $w$ is any constant of $W$, is an equation.
2. If $w_1 = w_2$ is an equation then $w_2 = w_1$ is an equation.
3. If $w_1 = w_2$ and $w_2 = w_3$ are equations, then $w_1 = w_3$ is an equation.
4. An equation is formed from an axiom of $S$ by an assignment of constants to variables, where each occurrence of a variable $x$ of type $D$ is consistently replaced by a constant $w$ of type $D$.
5. If $w_1 = w_2$ and $f(..., c, ...) = f(..., c, ...)$ are equations, and the constant $c$ is of the same type as $w_1$ and $w_2$, then $f(..., w_1, ...) = f(..., w_2, ...)$ is an equation.

The last equation in a derivation is the equation derived. ***

For example, Figure 4-3 contains some derivations from the specification of Nat12 in Figure 4-1.
Figure 4-3. Examples of derivations

(a) Derivation of $\text{Succ}^{13}(\text{Zero}) = \text{Succ}(\text{Zero})$

<table>
<thead>
<tr>
<th>Equation</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Succ}^{12}(\text{Zero}) = \text{Zero}$</td>
<td>(4)</td>
</tr>
<tr>
<td>$\text{Succ}^{13}(\text{Zero}) = \text{Succ}(\text{Zero})$</td>
<td>(5)</td>
</tr>
</tbody>
</table>

(b) Derivation of $\text{Zero} = \text{Succ}^{12}(\text{Zero})$

<table>
<thead>
<tr>
<th>Equation</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Succ}^{12}(\text{Zero}) = \text{Zero}$</td>
<td>(4)</td>
</tr>
<tr>
<td>$\text{Zero} = \text{Succ}^{12}(\text{Zero})$</td>
<td>(2)</td>
</tr>
</tbody>
</table>

(c) Derivation of $\text{Succ}^{24}(\text{Zero}) = \text{Zero}$

<table>
<thead>
<tr>
<th>Equation</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Succ}^{12}(\text{Zero}) = \text{Zero}$</td>
<td>(4)</td>
</tr>
<tr>
<td>$\text{Succ}^{24}(\text{Zero}) = \text{Succ}^{12}(\text{Zero})$</td>
<td>(5)</td>
</tr>
<tr>
<td>$\text{Succ}^{24}(\text{Zero}) = \text{Zero}$</td>
<td>(3)</td>
</tr>
</tbody>
</table>
The collection of all derivations forms a relation on the constant word algebra.

**DEFINITION 4.3** - Two elements, \( w_1 \) and \( w_2 \), of the constant word algebra \( W_c \) of a specification \( S \) are in the relation \( \text{speceq} \) if and only if the equation \( w_1 = w_2 \) can be derived from \( S \). We say that \( w_1 \) and \( w_2 \) are **equal** in \( \text{speceq} \).

***

Because derivations obey reflexivity, symmetry, transitivity and substitutivity, we have the following result.

**LEMMA 4.4** - \( \text{speceq} \) is a congruence on \( W_c \).

***

**PROOF** - By rule (1) \( \text{speceq} \) is reflexive. By rule (2) it is symmetric. By rule (3) it is transitive. So, \( \text{speceq} \) is an equivalence relation. To show that it is a congruence we must demonstrate substitutivity. Let \( w_1 \) and \( w_2 \) be equal constants of type \( S' \) in \( \text{speceq} \). Then by definition there is a derivation \( D \) of \( w_1 = w_2 \). For each function:

\[
f : S_1 \times \ldots \times S' \times \ldots \times S_n \rightarrow S_1
\]

construct a derivation \( D' \) of:
This is always possible for nonempty domains $S_1, \ldots, S_n, S', S_1$ by rules (1) and (5).

Appending $D'$ to $D$, we can derive:

$$f(c_1, \ldots, c', \ldots, c_n) = f(c_1, \ldots, c', \ldots, c_n).$$

by rule (5). ***

**DEFINITION 4.5** - The quotient algebra $\text{specalg}$ of a specification $(D, M)$ is defined by the congruence $\text{speceq}$ of $S$:

$$\text{specalg} = (W_c(D, M)/\text{speceq}, M).$$

One way of looking at specifications is to view them as describing models. That is, an algebra in which all the axioms are true is a model of that specification.
DEFINITION 4.6 - Let $S$ be a specification with signature $(T,F,V)$. Let $A$ be an algebra with the same signature. Define the extension $A'$ of $A$ as follows:

(1) Add a special type $BOOL$ with constant functions TRUE: $\rightarrow BOOL$ and FALSE: $\rightarrow BOOL$. (This new type is not to be confused with any other type $Bool$ already in $A$. If necessary, the old type $Bool$ is renamed so as not to conflict with the new type $BOOL$.)

(2) For each type $T_i$ in $T$ add a function $T_iEQ: T_i \times T_i \rightarrow BOOL$.

$T_iEQ(w_1,w_2) = TRUE$ if and only if $w_1$ and $w_2$ are the same constant of type $T_i$ in $W_c(T,F,V)$. Otherwise,

$T_iEQ(w_1,w_2) = FALSE$.

An axiom $w_1 = w_2$ in $A$, where $w_1$ and $w_2$ are words of type $T_i$, is true if and only if every instance $(w_1',w_2')$ of $(w_1,w_2)$ yields $T_iEQ(w_1',w_2') = TRUE$ in $A'$. $A$ is a model for $S$ if and only if every axiom in $A$ is true. We say that $A$ is correctly specified by $S$ whenever $A$ is a model for $S$. ***
**LEMMA 4.7** - Given a specification $S$, its quotient algebra $\text{specalg}$ is correctly specified by $S$. ***

**PROOF** - Let $w_1 = w_2$ be any axiom in $S$. By derivation rule (4), any instance $w_1' = w_2'$, where all variables have been consistently replaced by constants, may be derived. Therefore, every axiom is true in $\text{specalg}$. ***

In a sense, $\text{specalg}$ is the "best" model of a specification, because it contains as many different values in each domain as allowed by the axioms. However, it is not, in general, the only quotient algebra of the constant word algebra that is a model of a given specification. For example, $\text{Nat}4 = \langle \text{Nat}3/\text{mod}4, \{\text{Zero}, \text{Succ}3\} \rangle$ is a model of $\text{Nat}12$ (Figure 4-1), because the axiom:

$$\text{Zero} = \text{Succ}^{12}(\text{Zero})$$

is true in $\text{Nat}4$. It is easy to describe the collection of models of a specification.

**DEFINITION 4.8** - A quotient algebra is said to satisfy a specification if and only if it is the epimorphic image of $\text{specalg}$ of that specification. ***
THEOREM 4.9 - A data abstraction $A$ is correctly specified by a specification $S$ if and only if $A$ satisfies $S$. ***

PROOF -

( Satisfy $S$ $\implies$ Correct )

Let $h: \text{specalg} \rightarrow A$ be an epimorphism. Then, for every equation $w_1 = w_2$ true in \text{specalg} we have $h(w_1) = h(w_2)$ true in $A$. In particular, the image of every instance of every axiom in $S$ is true in $A$, because they are true in \text{specalg}. So, $A$ is a model for $S$.

( Correct $\implies$ Satisfy $S$ )

Let $[w_i]$ denote the congruence class of $w_i$ in specalg, $\{w_i\}$ denote the set of all words that are equal to $w_i$ in $A$. We show that $\{w_i\} \sqsupseteq [w_i]$ for all $i$. Let $E$ be an equation in a derivation of $w_1 = w_2$. If $E$ is a consequence of any of rules (1), (2), (3) or (5) it must be true in $A$. If $E$ is a consequence of rule (4), then it follows by an axiom of $S$. But, all axioms in $S$ are true in $A$. So, $E$ is true. Therefore, $w_1 = w_2$ in $A$. ***

The axioms are treated here as minimal, but not maximal.
conditions. That is, a specification does not describe a unique object, but a collection of objects, all of which satisfy the axioms. Fortunately, the collection is closed.

**THEOREM 4.10** - The collection of data abstractions that are correctly specified by a specification form a complete sublattice of $L_w$. We denote the sublattice by $L_s$. ***

**PROOF** - Let $S$ be the set of all correctly specified data abstractions. Every data abstraction in $S$ is an epimorphic image of $specalg$. So, it is an epimorphic image of the constant word algebra, and is in $L_w$. The meet and join operations in $L_w$ are congruence class intersection and congruence-closure union. We must show that $S$ is closed under these operations. Every element of $S$ contains the congruence classes of $specalg$ in its congruence classes. The intersection and congruence-closure union of classes will contain the congruence classes of $specalg$. So, $S$ is closed under meet and join. The trivial algebra is the top element of $L_s$. $Specalg$ is the bottom element. ***
Figure 4-4 contains the lattice of data abstractions, or algebras, that are correctly specified by Natl2. Each box in the Figure represents an algebra. The equation in the box is an axiom that must be true in that algebra. There are only six data abstractions in the lattice.

Comparing this lattice to Figure 3-3, we see that it is a sublattice of $L_w$ of the signature $(\text{Nat}, \text{Zero}, \text{Succ})$. Note that the lines connecting boxes represent epimorphisms implied by transitivity. The data abstraction at the bottom of the lattice is specalg. The data abstraction at the top is the trivial algebra, with one element in Nat.

The data abstraction at the bottom of the lattice $L_s$ for a specification is the initial algebra of that specification, in the terminology of [ADJ 77]. That is, there is a unique homomorphism from specalg to every other algebra that satisfies the axioms. In fact, there is a unique epimorphism defined by the lattice.
Figure 4-4. The lattice $L_8$ for Nat12
4.3. Inequalities

[Giarratana et al. 76] and [Polajnar 78] describe similar lattices, but without allowing as many interpretations. In particular, the trivial algebra is disallowed. This can be done by insisting that some domains have a single, allowed interpretation in abstractions. For example, one could insist that the natural numbers in stack have a fixed (perhaps infinite) number of elements. To accomplish this, we introduce the notion of inequalities in specifications.

DEFINITION 4.11 - An algebraic specification with inequalities consists of:

(1) An algebraic specification

(2) A collection of inequalities: pairs of well-formed terms composed from the elements of the algebraic specification (function names and variables), the two members of each pair separated by " # ".

The signature of an algebraic specification with inequalities is the signature of (1). If all the inequalities are composed of constant terms we call the specification an algebraic specification with constant inequalities. ***

Note that the set of inequalities may be infinite. When the
set is empty we have a specification as defined in Section 4.1.

An example of a specification with inequalities is shown in Figure 4-5. It is clear that an inequality may imply other inequalities. For example:

\[ \text{Zero} \neq \text{Succ}^6(\text{Zero}) \]

implies that

\[ \text{Zero} \neq \text{Succ}^3(\text{Zero}) \] .

However, inequality is not a transitive relation. It is the transitivity of equality combined with the contradiction of inequality that yields the implication. That is,

\[ \text{Zero} = \text{Succ}^3(\text{Zero}) \]

implies, by transitivity,

\[ \text{Zero} = \text{Succ}^6(\text{Zero}), \]

which is contradicted by

\[ \text{Zero} \neq \text{Succ}^6(\text{Zero}) \] .

A logical way to proceed, then, is to treat each inequality as potentially contradicting a collection of equalities. To find the collection, we find the collection of equal word derived from an equality.
Types: Nat

Functions: Zero: \rightarrow Nat
Succ: Nat \rightarrow Nat

Axioms: Succ^1(Zero) = Zero

Inequalities: Succ^6(Zero) \# Zero

Figure 4-5. Specification of Nat12 with inequalities
DEFINITION 4.12 - Two elements, \( w_1 \) and \( w_2 \), of the constant word algebra \( W_c \) of a specification with inequalities \( S(I) \) are in the relation \( \text{specineq}(i) \) if and only if the equation \( w_1 = w_2 \) can be derived from the specification \( S' \); where \( S' \) is formed from the signature of \( S \) and one axiom: the axiom that equates the two terms of the inequality \( i \). The relation \( \text{speceq} \) is defined to be the same as for the specification without inequalities. ***

We can form a quotient algebra from \( \text{specineq}(i) \) just as we did from \( \text{speceq} \).

LEMMA 4.13 - \( \text{specineq}(i) \) is a congruence on \( W_c \), for each inequality \( i \). ***

PROOF - \( \text{specineq}(i) \) is the congruence relation \( \text{speceq} \) for the specification with one axiom, the axiom that equates the two sides of the inequality \( i \). ***
DEFINITION 4.14 - The quotient algebra 
\( \text{specinalg}(i) \) of a specification with 
inequalities \( S(I) \) is defined by the congruence 
\( \text{specineq}(i) \) of \( S \):

\[
\text{specinalg}(i) = W_c / \text{specineq}(i)
\]

for each inequality \( i \in I \). The quotient 
algebra \( \text{specalg} \) is defined to be the same as for 
the specification without inequalities. ***

The quotient algebra \( \text{specinalg}(i) \) is in contradiction with 
the inequality \( i \). That is, it only has one value where 
the inequality states that there are two different values. 
\( \text{Specinalg}(i) \) is not an algebra correctly specified by the 
specification.

DEFINITION 4.15 - A quotient algebra is said to 
\textit{satisfy} a specification with inequalities \( S(I) \) 
if and only if (1) it is an epimorphic image of 
\( \text{specalg} \) and (2) it is not an epimorphic image of 
any \( \text{specinalg}(i) \) for all inequalities \( i \in I \).

***

THEOREM 4.16 - A data abstraction \( A \) is 
correctly specified by a specification with 
inequalities \( S(I) \) if and only if \( A \) satisfies 
\( S(I) \). ***
PROOF - Every axiom in $S(I)$ is true in $A$, because $A$ is an epimorphic image of $specalg$.

Every inequality $i$ in $I$ is true in $A$, because $A$ is not an epimorphic image of $specineq(i)$. Thus, $A$ is a model for $S(I)$.

If $A$ is a model for $S(I)$, then the axioms are true, and the inequalities are true. So, $A$ satisfies $S(I)$. ***

Adding inequalities to a specification, then, potentially removes some data abstractions from the collection that satisfy the specification. If the original collection formed a lattice, what does the new collection look like?

THEOREM 4.17 - The collection of data abstractions that are correctly specified by a specification with inequalities $S(I)$ forms a complete lower sub-semilattice of $L_w$ if and only if $specalg$ is not an epimorphic image of any $specinalg(i)$, for all inequalities $i \in I$.

If it is, the collection is void. We denote the semilattice $S_s$. ***

PROOF - When $specalg$ is an epimorphic image of some $specinalg(i)$, every epimorphic image of
specalg is an epimorphic image of
specinalg(i), because epimorphisms compose.
Hence, there is no model for $S(I)$ in this case.
Otherwise, the collection of correctly specified
data abstractions is closed at the bottom by
specalg. ***

Figure 4-6 contains the semilattice of data
abstractions that satisfy the specification in Figure 4-5.
By adding the inequality:

$$\text{Zero} \not\preceq \text{Succ}_{4}(\text{Zero})$$

the semilattice may be reduced to a single abstraction.

If a specification contains at least one axiom it is
possible to delete the whole lattice by one inequality: the
inequality of any substitution instance of the two words in
the axiom. For example,

$$\text{Zero} \not\preceq \text{Succ}_{12}(\text{Zero})$$

reduces Nat_{12} to nothing: there are no data abstractions
that satisfy the specification. Notice that any inequality
that defines (by changing the inequality into an equality) a
quotient algebra of which specineq is an epimorphic image,
reduces the lattice to nothing. If no one inequality in a
collection (possibly infinite) reduces the lattice to
nothing, then the whole collection does not.
Figure 4-6. Semilattice $S_8$ for Nat12 with inequalities
The final algebra of [Wand 79] and [Kamin 79] is the top element of $S_\mathbb{S}$, which is guaranteed to be a lattice by choosing a particular subalgebra for all domains except the type-of-interest. That is, an interpretation is chosen for each of the base types of the language. Usually, this interpretation is described by the initial algebra of a specification. (E.g., there are two values in type \texttt{bool}, an infinite number of values in type \texttt{int}, etc.) Each new type in a hierarchical data abstraction is defined by the final algebra of the specification, given the preceding definitions for all of the old types in the specification.

Figure 4-7 shows the relationship between the initial and final algebra interpretations, using the lattice of all interpretations, $L_\mathcal{W}$. 
Figure 4-7. Initial vs. Final Algebra
5. Implementations

Just as the collection of semantic interpretations of a specification can be described by quotient algebras of a word algebra, the collection of semantic interpretations of a class can be described by quotient algebras of the same word algebra. That is, every class has an associated signature, which yields a lattice of quotient algebras. All the correct semantic interpretations of the class are contained in that lattice.
5.1. Classes

**Definition 5.1** - A class consists of:

1. A signature \((S,F,V)\).
2. Function bodies in a programming language that implement each function in the signature.
3. Additional functions and procedures of the programming language as needed. ***

Figures 5-1 and 5-2 contain classes Nat18 and Stack. The programming language used is SIMPL-D [Gannon & Rosenberg 79]. Objects of each domain are implemented by a domain record of previously-defined types. Thus, Nat18 is implemented by a record with one field: an int variable. The type int is the implementation-defined integer type. Values of type Stack are implemented by an array of Nat and an int pointer. Note that Stack is bounded: there may only be 100 values in the stack.

The functions and procedures of a class are assumed to terminate. Also, all of the operations in the signature of the class must be functions. Global variables and side effects are disallowed.
class Nat = Zero, Succ

unique int Val

Nat func Zero
Nat Result
Result.Val := 0
return(Result)

Nat func Succ(Nat Arg)
Nat Result
if Arg.Val = 17
  then Result.Val := 0
else Result.Val := Arg.Val + 1
end
return(Result)

class
class Stack = Newstack, Push, Pop, Top

unique Nat array Vals(100)
unique int Tops

Stack func Newstack
    Stack Result
    Result.Tops := 0
    return(Result)

Stack func Push(Nat N, Stack S)
    Stack Result
    if S.Tops = 99
        then return(S)
    else Result.Vals(S.Tops) := N
        Result.Tops := S.Tops + 1
        return(Result)
    end

Stack func Pop(Stack S)
    Stack Result
    if S.Tops = 0
        then return(S)
    else Result := S
        Result.Tops := Result.Tops - 1
    end

Nat func Top(Stack S)
    Nat Result
    if S.Tops = 0
        then return(Zero)
    else Result.Val := S.Vals(S.Tops)
        return(Result)
    end

class
5.2. Correctness

The lattice of semantic interpretations, \( L_w \), contains more data abstractions than are correctly implemented by a given class. In particular, the data abstraction defined by \( \text{modl3} \) (see Section 3.3) is not correctly implemented by the class in Figure 5-1. In order to find the abstractions that are correctly implemented, we must formalize what it means for the code of a class to evaluate a constant word.

**DEFINITION 5.2** - The \texttt{exec} function is a mapping from \( w_c \) into the semantic domain used to define the base types of the programming language, using the meaning of the functions appearing in the constant:

\[
\text{exec}(f(w_1, \ldots, w_n)) = [f([w_1], \ldots, [w_n])].
\]

***

We assume that the given programming language has a semantics that defines domains of values for all the base types of the language. These domains may be sequences of bits or lattices in a denotational semantics. Any given constant expression must have a value in one of these domains. The bracket notation, due to [Kleene 52], denotes the function computed by the code for the named operation on the underlying domain. We extend the notation to words by:

\[
w = f(w_1, \ldots, w_n)
\] implies
[w] = [f]([[w_1]], ..., [[w_n]]).

Because we have assumed totality of all operations, the function denoted always exists.

For example, the operation Zero in Nat18 (Figure 5-1) yields a certain constant bit string (the constant 0 in the base type int). The operation Succ in Nat18, when applied to the result of Zero yields another bit string (the constant 1 in the base type int). It is possible to discover the functions [Zero] and [Succ] in Nat18 by enumerating all their possible values under the exec mapping (one value for [Zero] , eighteen for [Succ] ). In general, a function might have an infinite number of values.

**DEFINITION 5.3** - Let A be a data abstraction with signature (D_1 , ..., D_n, f_1 , ..., f_m) and C a class with the same signature, but written (D_1', ..., D_n', f_1', ..., f_m'). We say that A is correctly implemented by C if and only if there is an epimorphism R: exec(W_c) --> A. That is,

R(exec(f_1'(d_1', ..., d_k'))) = f_1(R(exec(d_1')) , ..., R(exec(d_k')))

for all d_1', ..., d_k' in D_1', ..., D_k' for all f_1'. We call R a representation mapping.

***
This notion of correctness originated in [Hoare 72]. Figure 5-3 shows a commutative diagram illustrating the relationship between A and C.
Figure 5-3. Correctness of implementations
Now we draw a parallel between specifications and classes. In particular, we can define a derivation*.

**DEFINITION 5.4** - A derivation* is a derivation (see chapter 4) with

(4') If \( \text{exec}(w_1) = \text{exec}(w_2) \) then \( w_1 = w_2 \) is an equation

substituted for rule (4). The last equation in a derivation* is the equation derived*. ***

The collection of words derived* equal defines a relation on the constant word algebra.

**DEFINITION 5.5** - Two elements, \( w_1 \) and \( w_2 \), of the constant word algebra of the signature of a class \( C \) are in the relation conceq if and only if the equation \( w_1 = w_2 \) can be derived* from \( C \). We say that \( w_1 \) and \( w_2 \) are equal in conceq. ***

Not unexpectedly, we get the following result.

**LEMMA 5.6** - Conceq is a congruence on \( W_c \).

***

**PROOF** - This proof is identical to the proof that speceq is a congruence on \( W_c \) (see chapter 4), since rule (4) for derivations is not used in that
proof. ***

**DEFINITION 5.7** - The quotient algebra \( \text{concalg} \)
of a class \( C \) is defined by the congruence\n\[ \text{conceq of } C : \]
\[ \text{concalg} = W_C / \text{conceq} . \]

Because \( \text{concalg} \) contains only one value for every\ncollection of words that evaluate to the same value in the\nunderlying domain of the programming language, \( \text{concalg} \) is\ncorrectly implemented.

**LEMMA 5.8** - Given a class \( C \), its quotient\nalgebra \( \text{concalg} \) is correctly implemented by \( C \).

***

**PROOF** - The representation mapping from \( C \) to\n\( \text{concalg} \) is simply the identity mapping, which is\nan epimorphism.

The algebra \( \text{concalg} \) is the algebra of the underlying\nbit strings or denotational semantics of the code of the\nclass. It must be correctly implemented because it is\n**exactly** implemented. However, there are other algebras that\nare correctly implemented. For example, \( \text{concalg} \) of the\nclass in Figure 5-2 contains different values for the words\n\( \text{Pop(Push(Newstack,Zero))} \) and\n\( \text{Pop(Push(Newstack,Succ(Zero)))} \), because the values of the
arrays are different (at location Vals(0)). The normal algebra of a stack would have one value for these two words. That is, the representation mapping to concalg is an isomorphism. However, any epimorphic image is correct.

**DEFINITION 5.9** - A quotient algebra is said to satisfy a class if and only if it is an epimorphic image of concalg of that class. ***

**THEOREM 5.10** - A data abstraction A is correctly implemented by a class C if and only if A satisfies C. ***

**PROOF** - If A is correctly implemented then there exists a representation mapping

\[ R: \text{exec}(W_c) \to A \]. But, \( \text{exec}(W_c) = \text{concalg} \). So, R is an epimorphism: \( R: \text{concalg} \to A \).

Thus, A satisfies C.

If A satisfies C then there exists an epimorphism E: concalg \( \to A \). But, E is then a representation mapping: \( E: \text{exec}(W_c) \to A \).

So, A is correctly implemented. ***

For example, the algebra with exactly three values in domain Nat:

\[
\begin{align*}
\text{Zero} &= \text{Succ}^3(\text{Zero}) = \text{Succ}^6(\text{Zero}) = \\
\text{Succ}(\text{Zero}) &= \text{Succ}^4(\text{Zero}) = 
\end{align*}
\]
\[ \text{Succ}^2(\text{Zero}) = \text{Succ}^5(\text{Zero}) = \ldots \]
is correctly implemented by \text{Nat18} (see Figure 5-1) under the representation mapping:

\[
\begin{align*}
\text{Val} & = 0,3,6,9,12 \text{ or } 15 & \rightarrow & \text{Zero} \\
\text{Val} & = 1,4,7,10,13 \text{ or } 16 & \rightarrow & \text{Succ(Zero)} \\
\text{Val} & = 2,5,8,11,14 \text{ or } 17 & \rightarrow & \text{Succ}^2(\text{Zero})
\end{align*}
\]

Notice that it is up to the user of the class to define and consistently use the correct representation mapping.

The collection of correctly implemented data abstractions is closed.

**THEOREM 5.11** - The collection of data abstractions that are correctly implemented by a class form a complete sublattice of \( L_w \). We denote the sublattice \( L_c \). ***

**PROOF** - This proof is identical to the proof for \( L_s \) (see chapter 4), except that \text{concalg} is used instead of \text{specalg}. ***

Figure 5.4 contains the lattice of all correctly implemented data abstractions for the class in Figure 5-1.
Figure 5-4. The lattice $L_c$ for Nat18
There are only six abstractions in the lattice. Comparing this lattice to Figure 3-3, we see that it is a sublattice of the appropriate $L_w$, but a different sublattice from Figure 4-4. The box at the bottom of Figure 5-4 represents the data abstraction with 18 different values in $\text{Nat}$. The box at the top represents the trivial algebra, with one value in $\text{Nat}$.
6. Specification/Implementation Intersection

If a class and a specification share the same signature they can be compared. In particular, if the same collection of data abstractions was intended to be described by both the class and the specification, then the success or failure of that intention can be described in terms of the overlap of the two lattices $L_S$ and $L_C$. When the overlap is perfect, i.e. the two collections are identical, then total success may be claimed. To measure the overlap, we use the following result:

**Theorem 6.1** - Given a specification $S$ with signature $(D,F,V)$ and a class $C$ with signature $(D,F)$, the collection of data abstractions that are correctly specified by $S$ and correctly implemented by $C$ form a complete sublattice of $L_S$ and $L_C$. We denote the common sublattice $S_L$. ***

**Proof** - Viewing the lattices $L_S$ and $L_C$ as sets, the intersection of $L_S$ and $L_C$ is the set of data abstractions that satisfy both the specification and the class. Let $S_L$ be that set. $S_L$ is not empty, because the trivial
algebra is in both $L_s$ and $L_c$.

The lattice operations, meet and join, are the same for $L_s$ and $L_c$. So, they are defined on $SL$. We need to show that $SL$ is closed under them. Every data abstraction $A_i$ in $SL$ is defined by a congruence relation $C_i$.

The meet of any two algebras is defined by their congruence intersection. Join is congruence-closure union. These operations preserve containment. So, the meet and join of two congruences containing both speceq and conceq is a congruence containing both speceq and conceq. ***

The sublattice $SL$ contains all those data abstractions correctly implemented and correctly specified. There are four possibilities: (1) $SL$ may be equal to $L_s$, but not equal to $L_c$, (2) $SL$ may be equal to $L_c$, but not equal to $L_s$, (3) $SL$ may be equal to both $L_s$ and $L_c$ or (4) $SL$ may not be equal to either $L_s$ or $L_c$.

In case (1) ($SL = L_s$, $SL \neq L_c$) the collection of correctly specified data abstractions are all correctly implemented, but some data abstractions that are correctly implemented are not correctly specified: the quotient algebra concalg contains more distinct values than the quotient algebra specalg. If agreement was intended and
the specification is not in error, two remedies are available: (1) the collection of representation mappings used may be restricted to those that map onto specalg or an algebra contained in $L_s$, or (2) the implementation may be changed to make fewer distinctions between values.

If the implementation is correct, but the specification is wrong, then a different set of axioms is needed. Removing axioms will (in general) increase the size of $L_s$, but may not produce the desired specalg. Changing variables to constants and increasing the lengths of words have a similar effect.

In case (2) ($SL = L_c$, $SL \neq L_s$) the collection of correctly implemented data abstractions are all correctly specified, but some data abstractions that are correctly specified are not correctly implemented: the quotient algebra specalg contains more distinct values than the quotient algebra concalg. If agreement was intended, and the specification is in error, then $L_s$ may be reduced in size by adding axioms. Changing constants to variables and shortening words in axioms will also reduce $L_s$. If the implementation is wrong, $L_c$ may be increased by various means. Adding new fields to the records that implement types, increasing the number of control paths in function bodies and lengthening the expressions that appear in function bodies are all modifications that potentially
In case (3) \((SL = L_s = L_c)\) the collections of correctly specified and correctly implemented data abstractions are identical: concalg and specalg are isomorphic. If the specification is wrong, then so is the implementation, and vice versa. If either one is known to be correct, then so is the other.

In case (4) \((SL \neq L_s, SL \neq L_c)\) some data abstractions are correctly specified but not correctly implemented, and some data abstractions are correctly implemented but not correctly specified: neither concalg nor specalg is in \(SL\). If the desired collection of data abstractions is contained in \(SL\), then both \(L_s\) and \(L_c\) may be reduced. If the desired collection is in \(L_s\) or \(L_c\), but not the other, then the erroneous lattice must be expanded. Otherwise, both lattices must be expanded.

Figure 6-1 shows the lattices \(L_s\), \(L_c\) and \(SL\) for the specification Natl2 (Figure 4-1) and the class Natl8 (Figure 5-1). It is an example of case (4) \((SL \neq L_s, SL \neq L_c)\).
Figure 6-1. The lattices $L_S$, $L_C$ and $SL$ for Nat.
If the data abstraction with six values in Nat was intended, then both the specification and the class may be changed. Adding the axiom:

\[ \text{Succ}^6(\text{Zero}) = \text{Zero} \]

shrinks \( L_s \) to \( SL \). Changing the test

\[
\text{if Arg.Val = 17}
\]

to

\[
\text{if Arg.Val = 5}
\]

shrinks \( L_c \) to \( SL \).

Although the techniques for changing specifications and classes discussed above may not always work to produce the desired lattice, there is hope that some sequence of changes will work.
7. Table Specifications

Software maintenance is facilitated by localization: the confinement of required changes to small syntactic units. Changes to axiomatic specifications do not seem to have this property. All the axioms in a specification need to be considered together in making any change. We propose, therefore, an alternate form of specification—table specifications.

Table specifications are weaker than axiomatic specifications in the sense that the set of all data abstractions specifiable by tables is a subset of the set of all data abstractions specifiable by axioms. Table specifications have two properties that axiomatic specifications do not have, however. First, changes to table specifications are more clearly localized than changes to axiomatic specifications. Second, there is a natural correspondence between table specifications and implementations of data abstractions that preserves the localization property.

Since every data abstraction is uniquely determined by a congruence on the constant word algebra of its signature, a presentation of the congruence suffices to describe the data abstraction. Furthermore, the division of types into
congruence classes is often mirrored in implementations by division of functions into control paths: each control path of a function handles a subset of congruence classes.

A congruence may have an infinite number of congruence classes. (An implementation almost always defines a finite number of congruence classes, but this number is usually too large to consider explicit enumeration of the classes.) On the other hand, most data abstractions decompose into a very small number of "patterns" of simple structures. That is, each value of the data abstraction may be constructed from a subset of the operations in the signature, and the other operations may be defined in terms of this subset by a small set of simple rules. It is this property that facilitates "data type induction" [Guttag, Horowitz & Musser 78].

The rows of table specifications are the patterns of congruence classes that suffice to define the congruence. The columns are operations. The entries in a table provide the simple rules that define the operations in terms of the rows. When a congruence has a finite number of classes of some type the table for that type can be completely elaborated, although it might be impractical to do so. In this case there is a one-to-one correspondence between rows and distinct values of the type. When a congruence has an infinite number of classes of some type, a small number of rows (less than 10) is usually sufficient to describe the
patterns of congruence classes. However, some perverse types do not have any finite description of their congruence classes.
7.1. T-Grammars

It will be convenient (for describing congruence-class patterns) to use a new notation for words in the word algebra of a signature. We will drop the use of parentheses in words and use angle brackets, < and >, to delimit subwords. This does not introduce any ambiguity, since the leading symbol of each subword has a constant arity. Thus, 
f(a, g(b, c)) becomes <f><a><g><b><c> . Where there is no ambiguity we will drop the angle brackets, also. So, 
<f><a><g><b><c> becomes fagbc . When a symbol is repeated an exponent will sometimes be used. For example, 
f^3a is an abbreviation for ffffa . This new notation suggests two kinds of patterns.
DEFINITION 7.1 - Let \( w \) be a word of type \( T \) in the word algebra \( W_v(S,F,V) \) containing variables \( V_1, \ldots, V_n \) of types \( T_1, \ldots, T_n \) in \( S \).

The set of all words of form \( w \) is the set of all words in \( W_v(S,F,V) \) obtained by substituting words of types \( T_1, \ldots, T_n \) for variables \( V_1, \ldots, V_n \) in \( w \). We denote this set by \( \text{form}(w) \). The set \( \text{form}(R_1,R_2) \) is equal to the product set \( \text{form}(R_1) \times \text{form}(R_2) \). Note that \( w \in \text{form}(w) \). The set of all words of rightform \( w \), denoted by \( \text{rform}(w) \), is the set of all words in \( W_v(S,F,V) \) of the form \( Xw \), where \( X \) is any string, including the null string. ***

Figure 7-1 contains some examples of \( \text{form}(w) \) and \( \text{rform}(w) \), using the signature of Stack in Figure 4-2.

A useful property of the word algebra of any signature is that it may be divided into a collection of languages, generated by context-free grammars.
form(<Succ><N>), an infinite set, includes:
   <Succ><Zero>,
   <Succ><Zero>,
   <Succ><Zero>, etc.

rform(<Succ><N>), an infinite set, includes:
   <Succ><N>,
   <Succ><N>,
   <Succ><N>, etc.

form(<Push><Zero><S>), an infinite set, includes:
   <Push><Zero><Newstack>,
   <Push><Pop><Push><N><S>,
   <Push><Zero><S>, etc.

rform(<Push><Zero><S>), an infinite set, includes:
   <Push><Zero><S>,
   <Push><Zero><S>,
   <Push><Zero><S>, etc.

Figure 7-1. Examples of form(ω) and rform(ω)
**Definition 7.2** - Let $T$ be a type in the signature $(S,F,V)$ of a data abstraction. The 
*T-grammar of $(S,F,V)$* is the 4-tuple $(S,F,P,T)$, where

1. The set of nonterminals is $S$, the set of type names.
2. The set of terminals is $F$, the set of operation names.
3. The set of productions $P$ is defined as follows:
   a. For each operation $f : T_1 \times \ldots \times T_n \rightarrow T$, $f \in F$, there is a production $T ::= f_{T_1 \ldots T_n}$.
   b. For each constant operation $f : \rightarrow T$, $f \in F$, there is a production $T ::= f$.
4. The start symbol is $T$, the type name. The language generated by $(S,F,P,T)$ is called the *$T$-language of $(S,F,V)$*. *****

The form of the productions guarantees that every $T$-grammar is context-free. The *Nat-grammar of Stack* (Figure 4-2) is shown in Figure 7-2.
Nat-grammar of Stack = (S,F,P,Nat)

S = Nat, Stack

F = Zero, Succ, Newstack, Push, Pop, Top

P = <Nat> ::= <Zero>
    <Nat> ::= <Succ><Nat>
    <Nat> ::= <Top><Stack>
    <Stack> ::= <Newstack>
    <Stack> ::= <Push><Nat><Stack>
    <Stack> ::= <Pop><Stack>

Figure 7-2. The Nat-grammar of Stack
THEOREM 7.3 - Let $L_i$ be the $T_i$-language for each type $T_i$ in signature $(S,F,V)$. The union of the $L_i$ is isomorphic to $W_c(S,F,V)$. ***

PROOF -

( $W_c(S,F,V) \subseteq T$-language )

(Proof by induction on the length of words)

(Basis step) Let $f$ be a word of type $T$ in $W_c(S,F,V)$. Because $f$ is a constant, there is a production $T :: f$ in the $T$-grammar of $(S,F,V)$. So, $f$ is an element of the $T$-language.

(Induction step) Now, let $f(w_1, \ldots, w_n)$ be a word of type $T$ of length $k$ and $w_1, \ldots, w_n$ be words (of length less than $k$) of types $T_1, \ldots, T_n$ in $W_c(S,F,V)$. We assume, by induction, that the words $w_1, \ldots, w_n$ are in the $T_1$-, $\ldots$, $T_n$-languages. There is a production $T ::= fT_1\ldots T_n$ in the $T$-grammar. So, there is a word $f w_1 \ldots w_n$ in the $T$-language.

( $T$-language $\subseteq W_c(S,F,V)$ )

(Proof by induction on the length of words)

(Basis Step) Let $f$ be a word in the $T$-language
of \((S,F,V)\). Since every production contains a leading terminal symbol, there must be a production \(T ::= f\) in the T-grammar of \((S,F,V)\). But, that means that there must be an operation \(f:\rightarrow T\) in \((S,F,V)\). So, \(f\) is in \(W_c(S,F,V)\).

(Induction Step) Now, let \(fw_1...w_n\) be a word in the T-language of length \(k\), so \(w_1, ..., w_n\) are words of length less than \(k\) of types \(T_1, ..., T_n\). We assume, for induction, that \(w_1, ..., w_n\) are in \(W_c(S,F,V)\). There must be a production \(T ::= fT_i...T_n\) in the T-grammar of \((S,F,V)\), because there is a production for each function in the signature. That means that there is an operation \(f: T_1 x ... x T_n \rightarrow T\) in \((S,F,V)\). So, \(fw_1...w_n\) is in \(W_c(S,F,V)\). ***

More useful for our purposes than context-free languages are regular languages. They have a convenient notation, regular expressions, that may be exploited in constructing tables and in generating implementations. Unfortunately, not every signature guarantees regularity of T-languages. Two special cases guarantee regularity.

**Theorem 7.4** - All T-languages of a signature with order \(\leq 1\) are regular. ***
PROOF - Each operation $f$ of type $T$ in the signature is constant, $f: \rightarrow T$, or has one parameter $f: T' \rightarrow T$. The productions in the T-grammar, then, are of the form $T ::= f$ or $T ::= fT'$. So, each T-grammar is right-linear.

***

THEOREM 7.5 - If a hierarchical specification has TOI-order $\leq 1$ for each TOI, and each TOI parameter is the rightmost parameter in its parameter list, then the TOI-languages are regular. ***

PROOF - Each production in the $T_i$-grammar is of the form $T_i ::= f$ or $T_i ::= fT_i \ldots T_n$ TOI, where $T_1, \ldots, T_n$ are old types. This grammar is right-linear. ***

The Nat-grammar of Stack (Figure 7-2) is regular. The Nat-grammar of Natplus (Figure 7-3) is not regular, because the production

$<Nat> ::= <Plus><Nat><Nat>$

makes the Nat-order two.
Nat-grammar of Natplus = (S,F,P,Nat)

S = Nat

F = Zero: --> Nat
   Succ: Nat --> Nat
   Plus: Nat x Nat --> Nat

P = <Nat> ::= <Zero>
   <Nat> ::= <Succ><Nat>
   <Nat> ::= <Plus><Nat><Nat>

Figure 7-3. The Nat-grammar of Natplus
7.2. Constructors

**DEFINITION 7.6** - A set of constructors of type $T$ in a specification $(S, F, V)$ is a subset $R = \{ R_1, R_2, \ldots \}$ of $W_v(S, F, V)$, such that every constant of type $T$ in $W_c(S, F, V)$ is equal to some word in: $\text{form}(R_1) \cup \text{form}(R_2) \cup \ldots$. When $\text{form}(R_i) \cap \text{form}(R_j)$ is empty for all $i \neq j$ the set of constructors is called disjoint. The singleton set $V$ containing only a variable with domain $T$ is the least set of constructors, called the trivial set of constructors. A set of constructors of a product of types $T_1 \times T_2$ is the product $R_1 \times R_2$ of sets of constructors for the individual types $R_i$ for $T_i$. A set of constructors for the empty set is the empty set. ***

A type may have many different sets of constructors in a specification. Figure 7-4 contains a specification of a type with three different nontrivial sets of constructors. As is evident from the example, no least nontrivial set of constructors need exist. The greatest set of constructors of $T$ is the subset of all elements of type $T$ in $W_v(S, F, J)$. 
Types: \( T \)

Functions:
\[
\begin{align*}
f &: \rightarrow T \\
g &: \rightarrow T \\
h &: \rightarrow T
\end{align*}
\]

Axioms: \( f = g \)

Sets of constructors:
\[
\begin{align*}
f, g, h \\
f, h \\
g, h
\end{align*}
\]

Figure 7-4. A type with three nontrivial sets of constructors
Although some specifications are not regular, an equivalent regular specification of the constructors can often be found for such specifications. It is an open question whether a regular set of constructors exists for every data abstraction.
7.3. Tables

A disjoint set of constructors of a type may be used to describe the congruence classes of that type. To complete the description of the congruence each operation must be defined in terms of the set of constructors. When the set is finite a table may be constructed.
DEFINITION 7.7 - A table specification is a regular signature \((S,F,V)\) and a finite set of triples, called tables, one triple for each domain arity in the signature. For each triple \(T = (R,C,E)\):

1. \(R = \{R_1, R_2, \ldots\}\) is a nontrivial, finite disjoint set of constructors of \(T\), called rows.
2. \(C = \{f_1, f_2, \ldots\}\) is the finite set of functions of domain arity \(T\), called columns.
3. \(E\) is a function: \(R \times C \rightarrow R'\), where \(R'\) is the union over all rows in all tables of the specification. \(E(r,c)\), where \(r \in R\) and \(c \in C\), is called the \((r,c)\)-entry of the table. When the entry is the word \(cr\), it is called trivial.

In addition, the following constraints must be met:

4. For each nonconstant row \(R_i = f\omega_1\ldots\omega_n\), where \(f: T_1 \times \ldots \times T_n \rightarrow T\) is a function in \(F\), \(E(\omega_1\ldots\omega_n,f) = R_i\).
5. Each variable in an entry appears in that entry's row name. ***

Figure 7-5 contains a table specification for Stack.
Types: Nat, Stack

Functions:  
Zero: \( \rightarrow \) Nat  
Succ: Nat \( \rightarrow \) Nat  
Newstack: \( \rightarrow \) Stack  
Pop: Stack \( \rightarrow \) Stack  
Top: Stack \( \rightarrow \) Nat  
Push: Nat \( \times \) Stack \( \rightarrow \) Stack

Variables:  
\( N: \) Nat, \( S: \) Stack

<table>
<thead>
<tr>
<th>Nat</th>
<th>Succ</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;Zero&gt;)</td>
<td>(&lt;\text{Succ}&gt;&lt;Zero&gt;)</td>
</tr>
<tr>
<td>(&lt;\text{Succ}&gt;&lt;N&gt;)</td>
<td>(&lt;\text{Succ}^2&gt;&lt;N&gt;)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack</th>
<th>Pop</th>
<th>Top</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;\text{Newstack}&gt;)</td>
<td>(&lt;\text{Newstack}&gt;)</td>
<td>(&lt;\text{Zero}&gt;)</td>
</tr>
<tr>
<td>(&lt;\text{Push}&gt;&lt;N&gt;&lt;S&gt;)</td>
<td>(&lt;S&gt;)</td>
<td>(&lt;N&gt;)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nat ( \times ) Stack</th>
<th>Push</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (\langle N&gt;,\langle S&gt;) )</td>
<td>(&lt;\text{Push}&gt;&lt;N&gt;&lt;S&gt;)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \emptyset )</th>
<th>Zero</th>
<th>Newstack</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;\text{Zero}&gt;)</td>
<td>(&lt;\text{Newstack}&gt;)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 7-5. A table specification of Stack
There are four tables: Nat, Stack, Nat x Stack and \(\emptyset\). The Nat table has two rows, \(<\text{Zero}>\) and \(<\text{Succ}><N>\), and one column, Succ. The rows are disjoint, because \(\text{form}(<\text{Zero}>) \neq \text{form}(<\text{Succ}><N>)\). Both entries in this table are trivial. The Stack table has two rows, \(<\text{Newstack}>\) and \(<\text{Push}><N><S>\), and two columns, Pop and Top. None of the entries in this table are trivial. Note that the variables \(N\) and \(S\) that appear in entries also appear in those entries' row name, \(<\text{Push}><N><S>\) (satisfying condition (5)). The Stack x Nat table has one row, the trivial set of constructors, \((<N>,<S>)\), and one column, Push. The entry in this table is trivial. The last table contains two constant functions, Zero and Newstack. Their entries are trivial.

A table specification defines a unique congruence on the constant word algebra of its signature. Thus, it defines a unique data abstraction. The rows of the tables define the congruence classes. The columns and entries define the functions.
DEFINITION 7.8 - Let T be a table specification with signature \((S,F,V)\). Let \(w\) be a word in \(W_v(S,F,V)\). The T-value of \(w\), denoted by \(\text{eval}(T,w)\), or \(\text{eval}(w)\) when the T is obvious from context, is defined recursively:

1. If \(w\) is a variable, then \(\text{eval}(w) = w\).
2. If \(w\) is a 0-ary function and appears as a column in the constant table of T, then \(\text{eval}(w)\) is the entry in that column. If it is not in the table, then \(\text{eval}(w)\) is undefined.
3. Otherwise, \(w = f(x_1, \ldots, x_n)\). Let \(y_i = \text{eval}(x_i)\), \(i = 1, \ldots, n\). If any \(y_i\) is undefined, then \(\text{eval}(w)\) is undefined. If \(f\) is not a column in a table in T, \(\text{eval}(w)\) is undefined. Let \(R = (z_1, \ldots, z_n)\) be a row in a table in T such that \(y_i \in \text{form}(z_i)\), \(i = 1, \ldots, n\). If no such row exists, \(\text{eval}(w)\) is undefined. Otherwise,

\[
\text{eval}(w) = E(R,f) \frac{[y_1, \ldots, y_n / z_1, \ldots, z_n]}{A[B_1, \ldots, B_n / C_1, \ldots, C_n]},
\]

where \(A[B_1, \ldots, B_n / C_1, \ldots, C_n]\) is the expression formed by substituting \(B_i\) for every occurrence of \(C_i\) in \(A\),
DATA ABSTRACTION TRANSFORMATIONS (U)
AUG 80 M A ARDIS
CSC-TR-925
AFOSR-TR-80-1061
F49620-80-C-0001
NL
END
DATE
FEB 80
DTIC
i = 1, ..., n. ***

For example, using the table specification of Stack in Figure 7-5,
\[
\text{eval}(\text{Succ}(\text{Top}(\text{Pop}(\text{Push}(\text{Zero}, \text{Newstack})))))) = \text{Succ}(\text{Zero}).
\]
The significance of the eval function is that it reduces words to forms that appear in the table specification entries.

**DEFINITION 7.9** - A derivation" is a derivation (see chapter 4) with

(4'') If eval\(w_1\) = eval\(w_2\) then \(w_1 = w_2\)
is an equation substituted for rule (4). The last equation in a derivation" is the equation derived". ***

**DEFINITION 7.10** - Two elements, \(w_1\) and \(w_2\), of the constant word algebra of the signature of a table are in the relation \(\text{tableq}\) if and only if the equation \(w_1 = w_2\) can be derived". We say that \(w_1\) and \(w_2\) are equal in \(\text{tableq}\) and write \(w_1 \equiv_{T} w_2\). ***

**LEMMA 7.11** - Tableq is a congruence on \(W_c(S,F,V)\). ***

**PROOF** - This proof is identical to the proof that speceq is a congruence (see chapter 4), since
rule (4) for derivations is not used in that
proof. ***

**DEFINITION 7.12** - The quotient algebra tablalg
of a table specification with signature (S,F,V)
is defined by the congruence tableq:

\[ \text{tablalg} = W_{c}(S,F,V)/\text{tableq} \]

We say that the table specification describes
tablalg. ***

A table specification describes a data abstraction. We need
to show that a table specification exists for many useful
data abstractions.

**THEOREM 7.13** - There exists a table specification
T for the constant word algebra of any regular
signature (S,F,V). ***

**PROOF** - Let the signature of T be (S,F,V).

(1) For each domain arity \( T_i \) construct a table:

(a) Let each constant function \( f: \longrightarrow T_i \)
be a row.

(b) Let each function \( f: T_i \longrightarrow T_j \) be a
column.

(c) For each function

\[ f: T_1 \times \ldots \times T_n \longrightarrow T_i \]
add a row

\( fV_1\ldots V_n \), where each \( V_j \) is a
variable with domain $T_j$.

(d) Let $E(R,f) = fR$, for all entries in $T_i$.

(2) For each domain arity $(T_1, \ldots, T_n)$ construct a table:

(a) Let the rows be the set $R_1 \times \cdots \times R_n$, where $R_i$ is the set of rows in table $T_i$.

(b) Let each function $f: T_1 \times \cdots \times T_n \rightarrow T_i$ be a column.

(c) Let $E(R,f) = fR$, for all entries in $(T_1, \ldots, T_n)$.

(3) For the empty domain arity $(\ )$ construct a table:

(a) Let each function $f: \rightarrow T_1$ be a column.

(b) Let $\emptyset$ be the only row.

(c) Let $E(\emptyset,f) = f$, for all entries in $(\ )$. ***

Figure 7-6 contains a table specification of the constant word algebra of Stack.
Types: Nat, Stack

Functions:
- Zero: \( \rightarrow \text{Nat} \)
- Succ: Nat \( \rightarrow \) Nat
- Newstack: \( \rightarrow \) Stack
- Pop: Stack \( \rightarrow \) Stack
- Top: Stack \( \rightarrow \) Nat
- Push: Nat \( \times \) Stack \( \rightarrow \) Stack

Variables:
- N: Nat, S: Stack

<table>
<thead>
<tr>
<th>Nat</th>
<th>Succ</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Zero&gt;</td>
<td>&lt;Succ&gt;&lt;Zero&gt;</td>
</tr>
<tr>
<td>&lt;Succ&gt;&lt;N&gt;</td>
<td>&lt;Succ&gt;&lt;N&gt;</td>
</tr>
<tr>
<td>&lt;Top&gt;&lt;S&gt;</td>
<td>&lt;Succ&gt;&lt;Top&gt;&lt;S&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack</th>
<th>Pop</th>
<th>Top</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Newstack&gt;</td>
<td>&lt;Pop&gt;&lt;Newstack&gt;</td>
<td>&lt;Top&gt;&lt;Newstack&gt;</td>
</tr>
<tr>
<td>&lt;Pop&gt;&lt;S&gt;</td>
<td>&lt;Pop&gt;&lt;S&gt;</td>
<td>&lt;Top&gt;&lt;Pop&gt;&lt;S&gt;</td>
</tr>
<tr>
<td>&lt;Push&gt;&lt;N&gt;&lt;S&gt;</td>
<td>&lt;Pop&gt;&lt;Push&gt;&lt;N&gt;&lt;S&gt;</td>
<td>&lt;Top&gt;&lt;Push&gt;&lt;N&gt;&lt;S&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nat ( \times ) Stack</th>
<th>Push</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;N&gt;,&lt;S&gt;)</td>
<td>&lt;Push&gt;&lt;N&gt;&lt;S&gt;</td>
</tr>
</tbody>
</table>

\( \emptyset \) | Zero: <Zero> | Newstack: <Newstack>

Figure 7-6. The table specification of \( W_c \) of Stack
Comparing the table specifications of Stack in Figures 7-5 and 7-6, we see that there are no rows of the form 
\(<\text{Pop}\><\text{S}\>\) or \(<\text{Top}\><\text{S}\>\) in Figure 7-5, but there are in Figure 7-6. The entries in Figure 7-6 are all trivial, but the entries in the Pop and Top columns of Figure 7-5 are not trivial. The Stack data abstraction described in Figure 7-5 is smaller (it has fewer values) than the data abstraction described in Figure 7-6.

In the next chapter we will show how to add an axiom to a table. This will demonstrate a technique for changing specifications in addition to constructing tables for axiomatic specifications.
8. Transformations of Tables

The set of all rows $R_i$ in a table specification with signature $(S,F,V)$ has four useful properties:

1. **(T1)** It is finite.
2. **(T2)** For all rows $R_i$, $R_j$
   $$\text{form}(R_i) \cap \text{form}(R_j) = \emptyset.$$  (Disjoint)
3. **(T3)** For every word $w \in W_c(S,F,V)$ there exists a row $R$ and a word $w'$, such that $w =_T w'$ and $w' \in \text{form}(R)$. (Complete)
4. **(T4)** For each row $R$ $\text{eval}(R) = R$. (Minimal)

The purpose of this chapter is to introduce transformations on table specifications that preserve these four properties.

In the course of transforming table specifications it will frequently be convenient to change the domains of variables that appear in tables. For example, restricting the domain of $V$ reduces the size of $\text{form}(V)$, as well as reducing the size of $\text{form}(fV)$, etc. It will always be possible to express the domain of a variable as a sum of rows. That is, for each variable $V$ of type $T$,
$$\text{dom}(V) = \text{form}(R_1) \cup \ldots \cup \text{form}(R_n),$$  where
$$\{R_1, \ldots, R_n\}$$ is a subset of the set of rows in the table $T$. 
Taking advantage of this fact, we will explicitly denote the domains of variables by extra columns in tables. Each variable will have a column in its type table with an entry of its name in every row of its domain. These extra columns will be written immediately after the row-name column. Figure 8-1 shows the Stack table specification of Figure 7-5, augmented with variable columns for variables N and S.
### Types: \( \text{Nat}, \text{Stack} \)

### Functions:
- **Zero: \( \rightarrow \text{Nat} \)**
- **Succ: \( \text{Nat} \rightarrow \text{Nat} \)**
- **Newstack: \( \rightarrow \text{Stack} \)**
- **Pop: \( \text{Stack} \rightarrow \text{Stack} \)**
- **Top: \( \text{Stack} \rightarrow \text{Nat} \)**
- **Push: \( \text{Nat} \times \text{Stack} \rightarrow \text{Stack} \)**

### Variables:
- **\( N: \text{Nat}, S: \text{Stack} \)**

#### Nat, Succ Table

<table>
<thead>
<tr>
<th>Nat</th>
<th>Succ</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;\text{Zero}&gt;)</td>
<td>(\text{N})</td>
</tr>
<tr>
<td>(&lt;\text{Succ}\text{\langle}N\text{\rangle})</td>
<td>(\text{N})</td>
</tr>
</tbody>
</table>

#### Stack, Pop, Top Table

<table>
<thead>
<tr>
<th>Stack</th>
<th>Pop</th>
<th>Top</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;\text{Newstack}\text{\rangle})</td>
<td>(\text{S})</td>
<td>(\text{Newstack}\text{\langle}\text{Zero}\text{\rangle})</td>
</tr>
<tr>
<td>(&lt;\text{Push}\text{\langle}N\text{\rangle}\text{\langle}S\text{\rangle})</td>
<td>(\text{S})</td>
<td>(\text{S}\text{\langle}\text{N}\text{\rangle})</td>
</tr>
</tbody>
</table>

#### Nat \(\times\) Stack, Push Table

<table>
<thead>
<tr>
<th>Nat (\times) Stack</th>
<th>Push</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\langle\text{N}\text{,}\text{S}\rangle)</td>
<td>(\text{Push}\text{\langle}\text{N}\text{\rangle}\text{\langle}S\text{\rangle})</td>
</tr>
</tbody>
</table>

#### \(\emptyset\), Zero, Newstack Table

<table>
<thead>
<tr>
<th>(\emptyset)</th>
<th>(\text{Zero})</th>
<th>(\text{Newstack})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\langle\text{Zero}\rangle)</td>
<td>(\langle\text{Newstack}\rangle)</td>
<td></td>
</tr>
</tbody>
</table>

---

**Figure 8-1. Table specification of Stack with variable columns**
We will observe the following rules for variables:

(V1) Any constant word $f_1 \ldots f_n g$ may be written in the form $f_1 \ldots f_n V$, where $\text{dom}(V) = \{g\}$. The reverse change (writing a word that contains a variable as a constant) may also be employed where appropriate.

(V2) If two variables in a table have the same domain they may be consolidated: one variable may be substituted for the other, leaving only one of the two variables in the table.

(V3) Variables that do not appear in any row names may be eliminated.

(V4) Whenever a row is removed from a table, that row is removed from all variable domains that included it. If a variable has a null domain as a result of such a change, all rows that include that variable must be removed from the table.

(V5) When a row is added to a table no variable domains change, unless explicitly noted.

The first three rules are for convenience. Rule (V1) makes it possible to treat all row names as if they contained a variable. Rules (V2) and (V3) cut down on the growth of new variables. Rules (V4) and (V5) are needed to explain the effects of transformations on the domains of variables.
8.1. Benign Transformations

**DEFINITION 8.1** - A change to a table specification that preserves properties (T1), (T2), (T3) and (T4) is called a benign transformation. ***

All of the transformations defined in this section are benign.

There is a table for each domain arity in a specification, not each type. If a type appears in a signature, but does not appear in the domain or range of any function, it will have no table in a table specification. We call these types null types. Addition and deletion of null types are benign transformations.

Addition of a function to a table specification requires defining the function over all rows in its domain arity table. This may be done by defining new rows in the result table: the result of applying the function $f$ to $(w_1, \ldots, w_n)$ is the word $fw_1\ldots w_n$. Since all the added entries are unique and distinct from all other entries, the transformation is benign. If a nontrivial function is desired axioms must be added by another transformation, described in Section 8.3.
ALGORITHM F: Add a Function

Input: T: a table specification with signature
(S,F,V)

f: \( T_1 \times \ldots \times T_n \rightarrow T_m \): a function
distinct from all those in F,
where none of \( T_1, \ldots, T_{n-1} \) are
the T0I

Output: T: a new table specification

Local Variables: R: a row name
k: an index to table names, as in \( T_k \)

(F1) Add new variables \( V_1, \ldots, V_n \) of types
\( T_1, \ldots, T_n \) to tables \( T_1, \ldots, T_n \) in
T. Let \( \text{dom}(V_i) \) be \( T_i, i=1, \ldots, n \).

(F2) Add a new variable \( W_i \) to each table \( T_i \) in
T. Let \( \text{dom}(W_i) \) be empty.

(F3) Let R be the row name \( fV_1 \ldots V_n \). Let k
be m.

(F4) Add R to table \( T_k \) with trivial entries.

(F5) Extend \( \text{dom}(W_k) \) to include R. If there is
a \( V_k \), extend it to include R, if
possible.

(F6) For each function
\[ f_i: T_a \times \ldots \times T_k \times \ldots T_b \rightarrow T_c \in F, \]
let R be the row name \( f_iW_a \ldots W_k \ldots W_b \). If
there is no such row in table \( T_c \), let k
be c and repeat steps (F4) through (F6).

(F7) Add the column $f$ with trivial entries to $T_m$.

(F8) Eliminate any variables $W_i$ with empty domains.

(F9) Add $f$ to $(S,F,V)$. ***

Adding a function $f: T_1 \times \ldots \times T_n \rightarrow T_m$ to a table specification implies adding new words (some of which are of the form $fV_1 \ldots V_n$) to the word algebra of its signature. Given variables $V_1, \ldots, V_n$ with domains $T_1, \ldots, T_n$, the row $fV_1 \ldots V_n$ is added to table $T_m$. This can cause a "ripple" effect in new word generation. Functions that include $T_m$ in their domain arity produce new words, which produce new words, and so on. To close this process a new variable, $W_i$, is introduced for each type $T_i$. $W_i$ "catches" all the new words introduced into type $T_i$. This limits the ripple effect to one pass through each domain in each domain arity, at worst. If no new words are added to a type $T_i$, the variable $W_i$ may be eliminated. An example of adding a function to a table specification is shown in Figure 8-2.
Original table specification:

Types: \( T_1 \)

Functions:

- \( Z: \rightarrow T_1 \)
- \( S: T_1 \rightarrow T_1 \)

Variables: \( N, V_1, W_1 : T_1 \)

\[
\begin{array}{ccc}
T_1 & N & S \\
Z & N & SZ \\
SN & N & S^2N \\
\end{array}
\]

Function to be added: \( P: T_1 \rightarrow T_1 \)

(F1):

\[
\begin{array}{ccc}
T_1 & N & V_1 & S \\
Z & N & V_1 & SZ \\
SN & N & V_1 & S^2N \\
\end{array}
\]

(F2):

\[
\begin{array}{ccc}
T_1 & N & V_1 & W_1 & S \\
Z & N & V_1 & W_1 & SZ \\
SN & N & V_1 & S^2N \\
\end{array}
\]

(F3): \( R = PV_1, k = 1 \)

Figure 8-2 (Part 1 of 3). Example of function addition
(F4.1): $T_1 \quad N \quad V_1 \quad W_1 \quad S$

<table>
<thead>
<tr>
<th>Z</th>
<th>N</th>
<th>V_1</th>
<th>S_Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>SN</td>
<td>N</td>
<td>V_1</td>
<td>S^2_N</td>
</tr>
<tr>
<td>PV_1</td>
<td></td>
<td></td>
<td>SPV_1</td>
</tr>
</tbody>
</table>

(F5.1): $T_1 \quad N \quad V_1 \quad W_1 \quad S$

<table>
<thead>
<tr>
<th>Z</th>
<th>N</th>
<th>V_1</th>
<th>S_Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>SN</td>
<td>N</td>
<td>V_1</td>
<td>S^2_N</td>
</tr>
<tr>
<td>PV_1</td>
<td></td>
<td></td>
<td>SPV_1</td>
</tr>
</tbody>
</table>

(F6.1): $R = SW_1, k = 1$

Figure 8-2 (Part 2 of 3). Example of function addition
(F4.2):

\[
\begin{array}{cccc}
T_1 & N & V_1 & W_1 & S \\
Z & N & V_1 & & SZ \\
SN & N & V_1 & & S^2N \\
PV & V_1 & W_1 & & SPV \\
SW & V_1 & W_1 & & S^2W \\
\end{array}
\]

(F5.2):

\[
\begin{array}{cccc}
T_1 & N & V_1 & W_1 & S \\
Z & N & V_1 & & SZ \\
SN & N & V_1 & & S^2N \\
PV & V_1 & W_1 & & SPV \\
SW & V_1 & W_1 & & S^2W \\
\end{array}
\]

(F6.2): \( R = SW_1 \), but \( SW_1 \) is already a row

(F7):

\[
\begin{array}{cccc}
T_1 & N & V_1 & W_1 & S & P \\
Z & N & V_1 & & SZ & PZ \\
SN & N & V_1 & & S^2N & PSN \\
PV & V_1 & W_1 & & SPV & P^2V \\
SW & V_1 & W_1 & & S^2W & PSW \\
\end{array}
\]

Figure 8-2 (Part 3 of 3). Example of function addition
Removal of a function is not a problem unless the function is used to form a set of constructors. If it is, then some new set of constructors would need to be chosen, and the table specification changed to reflect this, before the function could be removed.

**ALGORITHM D: Delete a Function**

Input: \( T \) : a table specification with signature 

\[ (S,F,V) \]

\[ f: T_1 \times \ldots \times T_n \rightarrow T_m \] : a function in 

\( F \) not found in any row name of 

\( T \)

Output: \( T \) : a new table specification

(D1) Remove the column with name \( f \) from table 

\( (T_1, \ldots, T_n) \) in \( T \).

(D2) Remove the function \( f \) from \( (S,F,V) \). ***

Since the function to be removed does not appear in any row name, its absence will not affect the set of constructors of any type. Also, it cannot appear in any entry if it does not appear in any row name. Removal is as simple as deleting one column from one table. Figure 8-3 shows the result of removing the function Pop from the table specification of Stack in Figure 8-1.
Types: Nat, Stack

Functions: Zero: \(\rightarrow\) Nat
Succ: Nat \(\rightarrow\) Nat
Newstack: \(\rightarrow\) Stack
Top: Stack \(\rightarrow\) Nat
Push: Nat \(\times\) Stack \(\rightarrow\) Stack

Variables: N: Nat, S: Stack

<table>
<thead>
<tr>
<th>Nat</th>
<th>N</th>
<th>Succ</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;Zero&gt;)</td>
<td>N</td>
<td>(&lt;\text{Succ}&lt;\text{Zero}&gt;)</td>
</tr>
<tr>
<td>(&lt;\text{Succ}&lt;\text{N}&gt;)</td>
<td>N</td>
<td>(&lt;\text{Succ}&lt;\text{N}&gt;)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack</th>
<th>S</th>
<th>Top</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;\text{Newstack}&gt;)</td>
<td>S</td>
<td>(&lt;\text{Zero}&gt;)</td>
</tr>
<tr>
<td>(&lt;\text{Push}&lt;\text{N}&lt;\text{S}&gt;)</td>
<td>S</td>
<td>(&lt;\text{N}&gt;)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nat (\times) Stack</th>
<th>Push</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\text{N},\text{S}))</td>
<td>(&lt;\text{Push}&lt;\text{N}&lt;\text{S}&gt;)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(\emptyset)</th>
<th>Zero</th>
<th>Newstack</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;\text{Zero}&gt;)</td>
<td>(&lt;\text{Newstack}&gt;)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 8-3. The result of deleting Pop from Figure 8-1.
Two more benign transformations that will be useful later are expansion and contraction of rows. These transformations have no semantic effect: the output table specification describes the same data abstraction as the input table specification.

**ALGORITHM** X: Expand a Row

*Input:* T : a table in a table specification

R = f₁...fₙ.V : a row in T

*Output:* T : a new table

(X1) For each row Rᵢ in dom(V) :

(a) Add a row f₁...fₙRᵢ to T.

(b) Add entries to fill the row:

\[ E(f₁...fₙRᵢ, f) = E(f₁...fₙV, f)[Rᵢ/V] \]

for each column f in T.

(c) Extend the domains of all variables defined over R to include Rᵢ.

(X2) Remove the row R from T. ***
This algorithm takes advantage of an equation that holds for all tables:

\[ \text{form}(fV_0) = \text{form}(fg_1V_1) U \ldots U \text{form}(fg_nV_n) \]

where

\[ \text{dom}(V_0) = \text{form}(g_1V_1) U \ldots U \text{form}(g_nV_n) \]

Since domains of variables are always defined in terms of rows, such an equation exists for every row \( V_0 \). The algorithm substitutes the rows on the right side of the equation for the row on the left side. The entries are formed by substitution of the appropriate \( g_iV_i \) for \( V_0 \). An example of expanding a row is shown in Figure 8-4.
Input:

\[
T = \begin{array}{cccc}
T_1 & N & V_1 & W_1 & S & P \\
Z & N & V_1 & S & P & PZ \\
SN & N & V_1 & S^2N & PSN & P^2V_1 \\
PV_1 & V_1 & W_1 & SPV_1 & P^2V_1 & P^2V_1 \\
SW_1 & V_1 & W_1 & S^2W_1 & PSW_1 & P^2V_1 \\
\end{array}
\]

\[R = SW_1\]

\[(X1.1) \quad R_1 = PV_1\]

\[
\begin{array}{cccc}
T_1 & N & V_1 & W_1 & S & P \\
Z & N & V_1 & S & P & PZ \\
SN & N & V_1 & S^2N & PSN & P^2V_1 \\
PV_1 & V_1 & W_1 & SPV_1 & P^2V_1 & P^2V_1 \\
SW_1 & V_1 & W_1 & S^2W_1 & PSW_1 & P^2V_1 \\
SPV_1 & V_1 & W_1 & S^{2PV_1} & PSPV_1 & P^2V_1 \\
\end{array}
\]

Figure 8-4 (Part 1 of 2). Example of row expansion
Figure 8-4 (Part 2 of 2). Example of row expansion

\[ R_1 = SW_1 \]

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>N</td>
<td>V</td>
<td>1</td>
<td>W</td>
<td>1</td>
<td>S</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P</td>
</tr>
<tr>
<td>SN</td>
<td>N</td>
<td>V</td>
<td>1</td>
<td>W</td>
<td>1</td>
<td>S^2N</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PSN</td>
</tr>
<tr>
<td>PV_1</td>
<td>V</td>
<td>1</td>
<td>W</td>
<td>1</td>
<td>SPV_1</td>
<td>P^2V_1</td>
</tr>
<tr>
<td>SW_1</td>
<td>V</td>
<td>1</td>
<td>W</td>
<td>1</td>
<td>S^2W_1</td>
<td>PSW_1</td>
</tr>
<tr>
<td>SPV_1</td>
<td>V</td>
<td>1</td>
<td>W</td>
<td>1</td>
<td>S^2PV_1</td>
<td>PSPV_1</td>
</tr>
<tr>
<td>S^2W_1</td>
<td>V</td>
<td>1</td>
<td>W</td>
<td>1</td>
<td>S^3W_1</td>
<td>PS^2W_1</td>
</tr>
</tbody>
</table>

\[ (X2): \]

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>N</td>
<td>V</td>
<td>1</td>
<td>W</td>
<td>1</td>
<td>S</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P</td>
</tr>
<tr>
<td>SN</td>
<td>N</td>
<td>V</td>
<td>1</td>
<td>W</td>
<td>1</td>
<td>S^2N</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PSN</td>
</tr>
<tr>
<td>PV_1</td>
<td>V</td>
<td>1</td>
<td>W</td>
<td>1</td>
<td>SPV_1</td>
<td>P^2V_1</td>
</tr>
<tr>
<td>SPV_1</td>
<td>V</td>
<td>1</td>
<td>W</td>
<td>1</td>
<td>S^2PV_1</td>
<td>PSPV_1</td>
</tr>
<tr>
<td>S^2W_1</td>
<td>V</td>
<td>1</td>
<td>W</td>
<td>1</td>
<td>S^3W_1</td>
<td>PS^2W_1</td>
</tr>
</tbody>
</table>
Expanding a row is useful when one of the new rows can be eliminated by another transformation. Such transformations will be described in Section 8.3.

The opposite of expansion of a row is contraction of rows. Contraction is possible whenever two rows have similar entries due to a common prefix. That is, a more "general" row could be defined, such that each of the two original rows is an "instance" of the general row. For example, rows that arise from expansion of a row $R$ may be contracted into the row $R$, since the expanded rows are all instances of $R$.

Let $R_1 = f_1 \ldots f_n g_1 \ldots g_m V_1$ and $R_2 = f_1 \ldots f_n h_1 \ldots h_k V_2$ be two rows to be contracted into $R = f_1 \ldots f_n V$. Then three properties must hold:

(P1) $\text{form}(R) = \text{form}(R_1) \cup \text{form}(R_2)$

(P2) For all columns $f$ in $T$ there exist words $e(R,f)$ in $W_v(S,F,V)$, such that:

(a) $E(R_1,f) = e(R,f)[g_1 \ldots g_m V_1/V]$

(b) $E(R_2,f) = e(R,f)[h_1 \ldots h_k V_2/V]$

(P3) There is no variable whose domain includes $R_1$ but not $R_2$, or vice versa.

Property (P1) guarantees that $R$ may be substituted for the
pair of rows \( R_1 \) and \( R_2 \) without changing the constructor set of the table. Property (P2) guarantees that the evaluation of any word will be the same before and after the contraction. Property (P3) prevents the side effect of increasing the size of the constructor set by increasing the domains of variables.

**ALGORITHM C : Contract Rows**

Input: \( T \) : a table in a table specification with signature \((S,F,V)\)

\[
\begin{align*}
R_1 &= f_1 \ldots f_n g_1 \ldots g_m V_1, \\
R_2 &= f_1 \ldots f_n h_1 \ldots h_k V_2: \text{rows in } T, \\
&\quad n \geq 0 \\
R &= f_1 \ldots f_n V: \text{a new row to replace } R_1 \\
&\quad \text{and } R_2, \text{such that properties } (P1), (P2) \text{ and } (P3) \text{ hold.}
\end{align*}
\]

Output: \( T \) : a new table

(C1) Add the row \( R \) to \( T \), with entries

\[
E(R,f) = e(R,f) \text{ for each column } f \text{ in } T.
\]

(C2) Extend the domains of variables defined over \( R_1 \) and \( R_2 \) to include \( R \).

(C3) Remove rows \( R_1 \) and \( R_2 \) from \( T \).

An example of contraction of rows is shown in Figure 8-5.

Note that the resulting table is the same as the input table of Figure 8-4.
Input:

<table>
<thead>
<tr>
<th>T =</th>
<th>T</th>
<th>N</th>
<th>V₁</th>
<th>W₁</th>
<th>S</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>N</td>
<td>V₁</td>
<td></td>
<td></td>
<td>SZ</td>
<td>PZ</td>
</tr>
<tr>
<td>SN</td>
<td>N</td>
<td>V₁</td>
<td></td>
<td></td>
<td>S²N</td>
<td>PSN</td>
</tr>
<tr>
<td>PV₁</td>
<td>V₁</td>
<td>W₁</td>
<td></td>
<td>SPV₁</td>
<td>P²V₁</td>
<td></td>
</tr>
<tr>
<td>SPV₁</td>
<td>V₁</td>
<td>W₁</td>
<td></td>
<td>S²PV₁</td>
<td>PSPV₁</td>
<td></td>
</tr>
<tr>
<td>S²W₁</td>
<td>V₁</td>
<td>W₁</td>
<td></td>
<td>S³W₁</td>
<td>PS²W₁</td>
<td></td>
</tr>
</tbody>
</table>

R₁ = SPV₁ (f₁ = S, ε₁ = P, V₁ = V₁)
R₂ = S²W₁ (f₁ = S, h₁ = S, V₂ = W₁)
R = SW₁ (f₁ = S, V = W₁)

Note: (1) form(SW₁) = form(SPV₁) U form(S²W₁)

(2a) E(SPV₁, S) = S²W₁ [PV₁/W₁], E(SPV₁, P) = PSW₁ [PV₁/W₁]

(2b) E(S²W₁, S) = S²W₁ [SW₁/W₁], E(S²W₁, P) = PSW₁ [SW₁/W₁]

(C1):

<table>
<thead>
<tr>
<th>T₁</th>
<th>N</th>
<th>V₁</th>
<th>W₁</th>
<th>S</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>N</td>
<td>V₁</td>
<td></td>
<td>SZ</td>
<td>PZ</td>
</tr>
<tr>
<td>SN</td>
<td>N</td>
<td>V₁</td>
<td></td>
<td>S²N</td>
<td>PSN</td>
</tr>
<tr>
<td>PV₁</td>
<td>V₁</td>
<td>W₁</td>
<td></td>
<td>SPV₁</td>
<td>P²V₁</td>
</tr>
<tr>
<td>SPV₁</td>
<td>V₁</td>
<td>W₁</td>
<td></td>
<td>S²PV₁</td>
<td>PSPV₁</td>
</tr>
<tr>
<td>S²W₁</td>
<td>V₁</td>
<td>W₁</td>
<td></td>
<td>S³W₁</td>
<td>PS²W₁</td>
</tr>
<tr>
<td>SW₁</td>
<td></td>
<td></td>
<td></td>
<td>S²W₁</td>
<td>PSW₁</td>
</tr>
</tbody>
</table>

Figure 8-5 (Part 1 of 2). Example of row contraction
### (C2):

<table>
<thead>
<tr>
<th></th>
<th>T_1</th>
<th>N</th>
<th>V_1</th>
<th>W_1</th>
<th>S</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>N</td>
<td>V_1</td>
<td></td>
<td></td>
<td>SZ</td>
<td>PZ</td>
</tr>
<tr>
<td>SN</td>
<td>N</td>
<td>V_1</td>
<td></td>
<td></td>
<td>S^2N</td>
<td>PSN</td>
</tr>
<tr>
<td>PV_1</td>
<td>V_1</td>
<td>W_1</td>
<td></td>
<td></td>
<td>SPV_1</td>
<td>P^2V_1</td>
</tr>
<tr>
<td>SPV_1</td>
<td>V_1</td>
<td>W_1</td>
<td></td>
<td></td>
<td>S^2PV_1</td>
<td>PSPV_1</td>
</tr>
<tr>
<td>S^2W_1</td>
<td>V_1</td>
<td>W_1</td>
<td></td>
<td></td>
<td>S^3W_1</td>
<td>PS^2W_1</td>
</tr>
<tr>
<td>SW_1</td>
<td>V_1</td>
<td>W_1</td>
<td></td>
<td></td>
<td>S^2W_1</td>
<td>PSW_1</td>
</tr>
</tbody>
</table>

### (C3):

<table>
<thead>
<tr>
<th></th>
<th>T_1</th>
<th>N</th>
<th>V_1</th>
<th>W_1</th>
<th>S</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>N</td>
<td>V_1</td>
<td></td>
<td></td>
<td>SZ</td>
<td>PZ</td>
</tr>
<tr>
<td>SN</td>
<td>N</td>
<td>V_1</td>
<td></td>
<td></td>
<td>S^2N</td>
<td>PSN</td>
</tr>
<tr>
<td>PV_1</td>
<td>V_1</td>
<td>W_1</td>
<td></td>
<td></td>
<td>SPV_1</td>
<td>P^2V_1</td>
</tr>
<tr>
<td>SW_1</td>
<td>V_1</td>
<td>W_1</td>
<td></td>
<td></td>
<td>S^2W_1</td>
<td>PSW_1</td>
</tr>
</tbody>
</table>

**Figure 8-5 (Part 2 of 2). Example of row contraction**
8.2. Rewrite Sets

By Lemma 7.11 a table specification defines a congruence on the constant word algebra of its signature. More than that, it defines a set of rewrite rules. Given two different words \( w_1 \) and \( w_2 \), such that
\[
eval(w_1) = eval(w_2) \quad \text{and} \quad w_1 = eval(\cdot^I),
\]
the rewrite rule \( w_2 \rightarrow w_1 \) may be derived. By changing table specifications we change the derived rewrite rules. Looking at it from the other direction, we need to know which rewrite rules we want before we can change the table specification. To do this, a total order on each type is defined.

**Definition 8.2** - Let \((S,F,V)\) be a regular signature. Let \( F \) be partitioned into the set of constant functions \( \{k_i\} \) and nonconstant functions \( \{f_i\} \). We define a total order \( \leq \) on each type in \( W_v(S,F,V) \) by first defining \( \prec \):

1. \( i < j \implies k_i \prec k_j \)
2. \( i < j \implies f_i \prec f_j \)
3. \( i < j \implies V_i \prec V_j \)
4. \( k_i \prec V_j \) for all \( k_i, V_j \)
5. \( V_i \prec f_j \) for all \( V_i, f_j \)

Let \( \leq \) be lexicographic ordering of strings, using \( \prec \) to compare individual symbols. ***
Note that it will never be necessary to compare function symbols of different types. Figure 8-6 shows the standard order on Stack (Figure 8-1), given the naming convention shown. Our algorithms will be defined so that all tables will follow the standard order in all implied rewrite rules.

**DEFINITION 8.3** - Let $T$ be a table specification. The *rewrite set of* $T$ is the set of all pairs $(fw_1...w_n, R)$, usually written in the form $fw_1...w_n \rightarrow R$, one pair for each nontrivial entry $R = E(w_1...w_n, f)$. ***

For example, the rewrite set of Stack (Figure 8-1) is

\[
\{ <\text{Pop}><\text{Newstack}> \rightarrow <\text{Newstack}> , \\
<\text{Pop}><\text{Push}><\text{N}><\text{S}> \rightarrow <\text{S}> ,  \\
<\text{Top}><\text{Newstack}> \rightarrow <\text{Zero}> , \\
<\text{Top}><\text{Push}><\text{N}><\text{S}> \rightarrow <\text{N}> \}
\]

The function eval uses the entries in a table specification to rewrite any word to a canonical term. By virtue of the table properties (T1) through (T4) the rewriting is always convergent [Musser 79b]. That is, a unique term is always produced after a finite number of rewriting steps.
Naming Convention:

Nat: \( k_1 = \langle \text{Zero} \rangle \)

\( f_1 = \langle \text{Succ} \rangle \)

\( f_2 = \langle \text{Top} \rangle \)

\( V_1 = \langle N \rangle \)

Stack: \( k_1 = \langle \text{Newstack} \rangle \)

\( f_1 = \langle \text{Push} \rangle \)

\( f_2 = \langle \text{Pop} \rangle \)

\( V_1 = \langle S \rangle \)

Standard Order:

Nat: \( \langle \text{Zero} \rangle \trianglelefteq \langle N \rangle \trianglelefteq \langle \text{Succ} \rangle \trianglelefteq \langle \text{Top} \rangle \)

Stack: \( \langle \text{Newstack} \rangle \trianglelefteq \langle S \rangle \trianglelefteq \langle \text{Push} \rangle \trianglelefteq \langle \text{Pop} \rangle \)

Examples:

\( \langle \text{Succ} \rangle^2 \langle N \rangle \trianglelefteq \langle \text{Top} \rangle \langle \text{Newstack} \rangle \)

\( \langle \text{Push} \rangle \langle \text{Zero} \rangle \langle \text{Pop} \rangle \langle \text{Push} \rangle \langle N \rangle \langle S \rangle \trianglelefteq \)

\( \langle \text{Pop} \rangle \langle \text{Newstack} \rangle \)

Figure 8-6. The standard order on Stack
Some changes to a table specification cause the rewrite set of some table(s) to become incomplete: they are no longer convergent. An algorithm for completing an incomplete set of rewrite rules is described in [Knuth & Bendix 70]. Unfortunately, this algorithm does not always terminate. We give here a modified version of part of the Knuth-Bendix algorithm. Our algorithm always terminates, but it may have to be used an infinite number of times to complete a rewrite set. A useful sub-algorithm is defined first.
ALGORITHM $ G : $ Generate Rewrite Rules

Input: $ T $ : a table specification with signature

$ (S,F,V) $ 

$ (x_1, x_2) : $ a pair of words in $ W_v(S,F,V) $ 

Output: $ P = \{ (L_i, R_i) \} : $ a set of pairs of words in $ W_v(S,F,V) $ to be used in rewriting: $ L_i \rightarrow R_i $ 

Local Variables: $ O = \{ (w_1, w_2) \} : $ a set of pairs of words in $ W_v(S,F,V) $ defined in $ T $ 

(G1) Let $ O $ and $ P $ be empty sets.

(G2) If both $ x_1 $ and $ x_2 $ are defined in $ T $ let $ O $ be the set $ \{ (x_1, x_2) \} $. Otherwise, substitute all rows in the domains of variables in $ x_1 $ and $ x_2 $ for those variables, generating a set $ O = \{ (w_1, w_2) \} $. 

(G3) For each pair $ (w_1, w_2) $ in $ O $:

(a) Compute $ w_1' = \text{eval}(w_1) $, $ w_2' = \text{eval}(w_2) $. 

(b) If $ w_1' \leq w_2' $ add $ w_2' \rightarrow w_1' $ to $ P $. 
If $ w_2' \leq w_1' $ add $ w_1' \rightarrow w_2' $ to $ P $. 
(Otherwise, $ w_1' = w_2' $, a trivial rewrite rule.) **
This algorithm consists of two major steps: (G2) and (G3). The first step generates a set of pairs of words. Each member of each pair is a word defined in the table specification. The second step orders the pairs by the standard order.

The first step is needed whenever a word in the input pair is in "too general" form for the table. That is, the input word is of the form $f_1 \ldots f_n V_1$, but there are rows in the table specification of the form $f_1 \ldots f_n g_1 \ldots g_m V_2$. In this case the variable $V_1$ must be expanded in exactly the same way that a row is expanded. Each row in the domain of $V_1$ is substituted for $V_1$, generating a set $O$ of pairs.
ALGORITHM K : Complete a Rewrite Set

Input:  T : a table specification with signature 
(\(S, F, V\))

\(O = \{ (L_i, R_l) \} \) : a set of pairs of 
words in \(W_{v}(S, F, V)\), the original 
rewrite set

Output: \(N = \{ L_i \rightarrow R_i \} \) : a new rewrite set, 
disjoint from \(O\)

Local Variables: \(P = \{ L_i \rightarrow R_i \} \) : a set of 
rewrite rules generated from 
Algorithm G

\(B = \{ (L_{i_1} \rightarrow R_{i_1}, L_{i_2} \rightarrow R_{i_2}) \} \) : a 
set of pairs of rewrite rules

\(C = \{ w_{i_1} \} \) : a set of words in \(W_{v}(S, F, V)\) 
representing overlapping of rewrite 
rules

(K1) Let \(N\) be an empty set.

(K2) Using all pairs \((L, R)\) in \(O\) generate a 
set of rewrite rules \(P\), using Algorithm G.

(K3) Add to \(N\) each rewrite rule in \(P\) that is 
not in \(O\).

(K4) Form the set \(B = \{ (L_{i_1} \rightarrow R_{i_1}, \)
\(L_{i_2} \rightarrow R_{i_2}) \} \) of all pairs of rewrite 
rules in \(O \cup N\).

(K5) If \(B\) is empty quit.
(K6) Select an element \((L_1 \rightarrow R_1, L_2 \rightarrow R_2)\) of \(B\), and remove it from \(B\).

(K7) Form the set \(C = \{f_1 \cdots f_n g_1 \cdots g_m h_1 \cdots h_k V_1, V_2 \}\) (where \(m \geq 1, n \geq 0, k \geq 0\)) of all "overlaps" of \(L_1\) and \(L_2\), where
\[
L_1 = f_1 \cdots f_n g_1 \cdots g_m V_1, \\
L_2 = g_1 \cdots g_m h_1 \cdots h_k V_2, \quad \text{and} \\
\text{dom}(V_1) \supset \text{form}(h_1 \cdots h_k V_2). \quad C \text{ may be empty.}
\]

(K8) For each element of \(C\) generate a rewrite set \(P\), using Algorithm G with \(w_1 = R_1\) and \(w_2 = f_1 \cdots f_n R_2\). Add to \(N\) each rule in \(P\) that is not in \(O\).

(K9) Repeat steps (K5) through (K9). ***

This algorithm finds implied rewrite rules, given a rewrite set and a table. Several invocations of the algorithm may be needed to complete a rewrite set. Algorithm G is used to generate a set of rewrite rules from input pairs of words. This is necessary for those cases where the table specification has a rewrite set that is different from the one input to the algorithm. Such cases arise in Algorithm A, defined in the next section.

To find implied rewrite rules, each pair of rewrite rules is checked for "overlap." If a word can be rewritten in two different ways (an overlap), a new rewrite rule is
generated. Overlapping in regular signatures can only occur in one form: \( fghV \rightarrow ahV \) and \( fghV \rightarrow fbV \), where \( fgV \rightarrow aV \) and \( ghV \rightarrow bV \) are rewrite rules, and \( f, g \) and \( h \) are subwords of any finite length.

An example of completing a rewrite set is shown in Figure 8-7. In the example Algorithm K is invoked twice. An example that does not converge is shown in Figure 8-8. In this example each invocation of Algorithm K generates a new rewrite rule that requires another invocation of Algorithm K.
Types: $T$

Functions:
- $Z: \rightarrow T$
- $S: T \rightarrow T$

$$T = \begin{array}{c|c}
Z & S \\
SZ & S^2Z \\
S^2Z & S^3Z \\
S^3Z & S^4Z \\
S^4Z & Z
\end{array}$$

$$O = \{<S^2Z, Z>, <S^5Z, Z>\}$$

First invocation of Algorithm K:

(K1) $N = \emptyset$

(K2.1) Invoke Algorithm G with $x_1 = S^2Z$, $x_2 = Z$

- (G1) $O = \emptyset$
- (G2) $O = <S^2Z, Z>$
- (G3) (a) $w_1' = S^2Z$, $w_2' = Z$
- (b) $P = \{S^2Z \rightarrow Z\}$

$$P = \{S^2Z \rightarrow Z\}$$

(K2.2) Invoke Algorithm G with $x_1 = S^5Z$, $x_2 = Z$

$$p = \{S^2Z \rightarrow Z, S^5Z \rightarrow Z\}$$

Figure 8-7 (Part 1 of 3). Example of rewrite set completion
(K3) \( N = \emptyset \)

(K4) \( B = \langle S^2Z \rightarrow Z, S^2Z \rightarrow Z \rangle, \langle S^2Z \rightarrow Z, S^5Z \rightarrow Z \rangle, \langle S^5Z \rightarrow Z, S^2Z \rightarrow Z \rangle, \langle S^5Z \rightarrow Z, S^5Z \rightarrow Z \rangle \)

(K6.1) Select \( \langle S^2Z \rightarrow Z, S^2Z \rightarrow Z \rangle \)

(K7.1) \( C = \langle S^2Z \rangle \) (\( S^2Z \) overlaps with itself trivially.)

(K8.1) \( P = \langle S^2Z \rightarrow Z \rangle, N = \emptyset \)

(K6.2) Select \( \langle S^2Z \rightarrow Z, S^5Z \rightarrow Z \rangle \)

(K7.2) \( C = \emptyset \) (No overlaps)

(K8.2) \( P = \emptyset, N = \emptyset \)

Figure 8-7 (Part 2 of 3). Example of rewrite set completion
(K6.3) Select $< s^5z \rightarrow z$, $s^2z \rightarrow z >$

(K7.3) $C = \{ s^5z \}$  $(f_1 f_2 f_3 = s^3$, $g_1 g_2 = s^2$, $V_1 = V_2 = Z)$

(K8.3) $P = \{ s^3z \rightarrow z \}$, $N = s^3z \rightarrow z$

(K6.4) Select $< s^5z \rightarrow z$, $s^5z \rightarrow z >$

$B = \emptyset$

(K7.4) $C = \{ s^5z \}$  (Trivial overlap)

(K8.4) $P = \{ s^5z \rightarrow z \}$, $N = \{ s^3z \rightarrow z \}$

(K5.5) $B = \emptyset \Rightarrow$ quit

Second invocation of Algorithm K:

$O = \{ s^2z \rightarrow z$, $s^5z \rightarrow z$, $s^3z \rightarrow z \}$

Result: $N = sz \rightarrow z$

$O \cup N$ is a complete rewrite set

Figure 8-7 (Part 3 of 3). Example of rewrite set completion
Types: D  
Functions:  
\( h: \rightarrow D \)  
\( f: D \rightarrow D \)  
\( g: D \rightarrow D \)  
Variables:  
\( V: D \)

\[
\begin{array}{c|c|c|c}
T & V & f & g \\
\hline
h & V & fh & gh \\
fV & V & f^2V & gfV \\
gV & V & fgV & g^2V \\
\emptyset & h & \\
\hline
\end{array}
\]

\( O = \{<fgfV, gfV>\} \)

Note: Standard order must be: \( h \leq f \leq g \)

Otherwise, \( gfV \rightarrow fgfV \rightarrow f^2gfV \rightarrow \ldots \)

First invocation of Algorithm K:

\( fgfgfV \rightarrow gfgfV \rightarrow g^2fV \) and

\( fgfgfV \rightarrow fg^2fV \)

So, \( N = \{<fg^2fV, g^2fV>\} \)

Second invocation:

\( fg^2fgfV \rightarrow g^2fgfV \rightarrow g^3fV \) and

\( fg^2fgfV \rightarrow fg^3fV \)

So, \( N = \{<fg^3fV, g^3fV>\} \)

\( \ldots \)

Figure 8-8. Example of non-convergent rewrite set
8.3. Adding Axioms

Adding an axiom to an axiomatic specification usually changes the lattice of specified data abstractions. When it does, it shrinks the lattice by eliminating some data abstractions. The corresponding change to a table specification is a change in the rows, the constructors of a type. Given an axiom, the appropriate row changes can sometimes be made so that \( \text{tablalg} = \text{specalg} \). This cannot be done when the use of Algorithm K fails to converge.

Two algorithms are defined for adding axioms to table specifications. The first algorithm changes the constructor set of a type by removing a row and all words that contain that row as a rightmost subword. It is invoked by the second algorithm, the algorithm to add an axiom.
**ALGORITHM** R: Remove a Row

**Input:** T: a table in a table specification

\[ T' \]

R: \[ f_1 \ldots f_nV \]: a row in T

**Output:** T: a new table

**Local Variables:**

k: an integer

W: a variable in the signature of T'

of type T

R': \[ g_1 \ldots g_m f_1 \ldots f_k X \]: a row in T

(R1) Let k be n.

(R2) If k = 0 quit.

(R3) Let W be a new variable with

\[ \text{dom}(W) = \text{form}(f_{k+1} \ldots f_n V) . \]

(R4) For each row \( R' = g_1 \ldots g_m f_1 \ldots f_k X \) in T, m \( \geq 1 \), where dom(X) \( \supseteq \) dom(W):

(a) Replace X with a new variable \( X' \):

\[ \text{dom}(X') = \text{dom}(X) - \text{dom}(W) . \]

(b) If \( \text{dom}(X') \) is empty remove \( R' \) from T.

(R5) Subtract 1 from k, and repeat steps (R2) through (R5). ***

Given a word \( f_1 \ldots f_n V \), Algorithm R eliminates words in

\[ \text{rform}(f_1 \ldots f_n V) \), \text{rform}(f_1 \ldots f_{n-1} V_{n-1}) \), \ldots \), \text{and} \]

\[ \text{rform}(f_1 V_1) \), in that order (where
dom(V_i) \supseteq form(f_{i+1} \ldots f_n V_n). It does this by changing the domains of variables in row names. Each word to be eliminated is removed from the domains of variables. For example, if fV_1 is to be eliminated, and there is a row gfV_1, where dom(V_1) = form(gfV_1) \cup form(hV_2), then a variable change is made:

gfV_1 \rightarrow gfV_3, \ dom(V_3) = form(hV_2).

If a variable's domain becomes empty as the result of such a change the row containing that variable is removed from the table.
**Algorithm A: Add an Axiom**

**Input:** $T$ : a table specification with signature $(S,F,V)$

$\langle w_1, w_2 \rangle$ : a pair of words in $W_v(S,F,V)$, the axiom to be added

**Output:** $T$ : a new table specification

**Local Variables:**
- $P = \{ L_i \rightarrow R_i \}$ : a set of rewrite rules generated from the axiom
- $O = \{ L_i \rightarrow R_i \}$ : the original rewrite set generated from $T$ extended by calls to Algorithm $K$
- $L \rightarrow R$ : a particular pair from $P$
- $T'$ : a table in $T$

1. Let $O$ be the rewrite set of $T$.
2. Using Algorithm $G$, generate a set $P$ of rewrite rules from the axiom $\langle w_1, w_2 \rangle$.
3. If $P$ is empty quit.
4. Remove a rule $L \rightarrow R$ from $P$. Add the selected rule to $O$.
5. Expand rows (Algorithm $X$) until $L$ is a row in some table $T'$.
6. Remove $L$ from $T'$ (Algorithm $R$).
7. Substitute $R$ for $L$ in all entries of $T$.
8. Using Algorithm $K$, complete the rewrite set.
0 with the table specification T. Add the new set, N, of rewrite rules to P and to 0.

(A9) Repeat steps (A3) through (A9). ***

Each axiom to be added to a table is transformed into a set of rewrite rules, using Algorithm G. Each rewrite rule is processed, yielding a new rewrite set. Algorithm K is then used to complete the rewrite set. Because new rewrite rules may need to be added, an iterative process is necessary. In those cases where no convergent rewrite rule set can be formed, the algorithm does not terminate.

For each rewrite rule to be processed the table specification is changed by eliminating all occurrences of the left side of the rewrite rule. Algorithm R is used to eliminate occurrences in row names. Substitution of the right side of the rewrite rule is used for entries. An example of axiom addition is shown in Figure 8-9.
Types: $T_1$

Functions: $Z: \rightarrow T_1$
$P: T_1 \rightarrow T_1$
$S: T_1 \rightarrow T_1$

Variables: $V_i: T_1$, $i=1,2,...$

$$
\begin{array}{|c|c|c|c|}
\hline
T & V_1 & P & S \\
\hline
Z & V_1 & PZ & SZ \\
PV_1 & V_1 & P^2V_1 & SPV_1 \\
SV_1 & V_1 & PSV_1 & S^2V_1 \\
\hline
\emptyset & Z & \text{Note: This table will not change.} \\
\hline
\end{array}
$$

$\langle w_1, w_2 \rangle = \langle S^4Z, Z \rangle$

(A1) $\emptyset = \emptyset$ (Every rewrite rule in $T$ is trivial.)

(A2) Use Algorithm G to generate rewrite rules:

$P = \{ S^4Z \rightarrow Z \}$

(A4) Select $S^4Z \rightarrow Z$ from $P$

Figure 8-9 (Part 1 of 9). Example of axiom addition
(A5) Use Algorithm X to expand rows until $S^4Z$ is a row. Because Algorithm X is not as "smart" as it could be the result is a larger table than necessary. This will be corrected at the end by contraction. Altogether, Algorithm X is invoked three times.

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$V_1$</th>
<th>$P$</th>
<th>$S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>$V_1$</td>
<td>PZ</td>
<td>SZ</td>
</tr>
<tr>
<td>PV</td>
<td>$V_1$</td>
<td>$P^2V_1$</td>
<td>SPV</td>
</tr>
<tr>
<td>S</td>
<td>$V_1$</td>
<td>PSZ</td>
<td>S$^2$Z</td>
</tr>
<tr>
<td>SPV</td>
<td>$V_1$</td>
<td>PSPV</td>
<td>S$^2$PV</td>
</tr>
<tr>
<td>$S^2$</td>
<td>$V_1$</td>
<td>PS$^2$Z</td>
<td>S$^3$Z</td>
</tr>
<tr>
<td>$S^2$PV</td>
<td>$V_1$</td>
<td>PS$^2$PV</td>
<td>S$^3$PV</td>
</tr>
<tr>
<td>$S^3$</td>
<td>$V_1$</td>
<td>PS$^3$Z</td>
<td>S$^4$Z</td>
</tr>
<tr>
<td>$S^3$PV</td>
<td>$V_1$</td>
<td>PS$^3$PV</td>
<td>S$^4$PV</td>
</tr>
<tr>
<td>$S^4$</td>
<td>$V_1$</td>
<td>PS$^4$Z</td>
<td>S$^5$Z</td>
</tr>
<tr>
<td>$S^4$PV</td>
<td>$V_1$</td>
<td>PS$^4$PV</td>
<td>S$^5$PV</td>
</tr>
<tr>
<td>$S^5$</td>
<td>$V_1$</td>
<td>PS$^5$Z</td>
<td>S$^6$Z</td>
</tr>
<tr>
<td>$S^5$PV</td>
<td>$V_1$</td>
<td>PS$^5$PV</td>
<td>S$^6$PV</td>
</tr>
<tr>
<td>$S^6$</td>
<td>$V_1$</td>
<td>PS$^6$Z</td>
<td>S$^7$Z</td>
</tr>
<tr>
<td>$S^6$PV</td>
<td>$V_1$</td>
<td>PS$^6$PV</td>
<td>S$^7$PV</td>
</tr>
<tr>
<td>$S^7$</td>
<td>$V_1$</td>
<td>PS$^7$Z</td>
<td>S$^8$Z</td>
</tr>
<tr>
<td>$S^7$PV</td>
<td>$V_1$</td>
<td>PS$^7$PV</td>
<td>S$^8$PV</td>
</tr>
<tr>
<td>$S^8$</td>
<td>$V_1$</td>
<td>PS$^8$V</td>
<td>S$^9$V</td>
</tr>
</tbody>
</table>

Figure 8-9 (Part 2 of 9). Example of axiom addition
(A6) Remove \( S^4Z \) from \( T_1 \): Use Algorithm R

(R1) \( k = 4 \)

(R3.1) \( \text{dom}(W) = \text{form}(Z) \)

(R4.1.1) \( R' = S^4Z, X = Z \)

(a) Remove \( Z \) from \( \text{dom}(Z) \)

(b) Remove \( S^4Z \) from table

<table>
<thead>
<tr>
<th>( T_1 )</th>
<th>( V_1 )</th>
<th>( P )</th>
<th>( S )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Z )</td>
<td>( V_1 )</td>
<td>( PZ )</td>
<td>( SZ )</td>
</tr>
<tr>
<td>( PV_1 )</td>
<td>( V_1 )</td>
<td>( P^2V_1 )</td>
<td>( SPV_1 )</td>
</tr>
<tr>
<td>( SZ )</td>
<td>( V_1 )</td>
<td>( PSZ )</td>
<td>( S^2Z )</td>
</tr>
<tr>
<td>( SPV_1 )</td>
<td>( V_1 )</td>
<td>( PSPV_1 )</td>
<td>( S^2PV_1 )</td>
</tr>
<tr>
<td>( S^2Z )</td>
<td>( V_1 )</td>
<td>( PS^2Z )</td>
<td>( S^3Z )</td>
</tr>
<tr>
<td>( S^2PV_1 )</td>
<td>( V_1 )</td>
<td>( PS^2PV_1 )</td>
<td>( S^3PV_1 )</td>
</tr>
<tr>
<td>( S^3Z )</td>
<td>( V_1 )</td>
<td>( PS^3Z )</td>
<td>( S^4Z )</td>
</tr>
<tr>
<td>( S^3PV_1 )</td>
<td>( V_1 )</td>
<td>( PS^3PV_1 )</td>
<td>( S^4PV_1 )</td>
</tr>
<tr>
<td>( S^4PV_1 )</td>
<td>( V_1 )</td>
<td>( PS^4PV_1 )</td>
<td>( S^5PV_1 )</td>
</tr>
<tr>
<td>( S^5Z )</td>
<td>( V_1 )</td>
<td>( PS^5Z )</td>
<td>( S^6Z )</td>
</tr>
<tr>
<td>( S^5PV_1 )</td>
<td>( V_1 )</td>
<td>( PS^5PV_1 )</td>
<td>( S^6PV_1 )</td>
</tr>
<tr>
<td>( S^6Z )</td>
<td>( V_1 )</td>
<td>( PS^6Z )</td>
<td>( S^7Z )</td>
</tr>
<tr>
<td>( S^6PV_1 )</td>
<td>( V_1 )</td>
<td>( PS^6PV_1 )</td>
<td>( S^7PV_1 )</td>
</tr>
<tr>
<td>( S^7Z )</td>
<td>( V_1 )</td>
<td>( PS^7Z )</td>
<td>( S^8Z )</td>
</tr>
<tr>
<td>( S^7PV_1 )</td>
<td>( V_1 )</td>
<td>( PS^7PV_1 )</td>
<td>( S^8PV_1 )</td>
</tr>
<tr>
<td>( S^8Z )</td>
<td>( V_1 )</td>
<td>( PS^8Z )</td>
<td>( S^9Z )</td>
</tr>
</tbody>
</table>

Figure 8-9 (Part 3 of 9). Example of axiom addition
(R4.1.2) Remove $S^5Z \cdot (g_1 = S)$

(R4.1.3) Remove $S^6Z \cdot (g_1g_2 = S^2)$

(R4.1.4) Remove $S^7Z \cdot (g_1g_2g_3 = S^3)$

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$V_1$</th>
<th>$P$</th>
<th>$S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Z$</td>
<td>$V_1$</td>
<td>$PZ$</td>
<td>$SZ$</td>
</tr>
<tr>
<td>$PV_1$</td>
<td>$V_1$</td>
<td>$P^2V_1$</td>
<td>$SPV_1$</td>
</tr>
<tr>
<td>$SZ$</td>
<td>$V_1$</td>
<td>$PSZ$</td>
<td>$S^2Z$</td>
</tr>
<tr>
<td>$SPV_1$</td>
<td>$V_1$</td>
<td>$PSPV_1$</td>
<td>$S^2PV_1$</td>
</tr>
<tr>
<td>$S^2Z$</td>
<td>$V_1$</td>
<td>$PS^2Z$</td>
<td>$S^3Z$</td>
</tr>
<tr>
<td>$S^2PV_1$</td>
<td>$V_1$</td>
<td>$PS^2PV_1$</td>
<td>$S^3PV_1$</td>
</tr>
<tr>
<td>$S^3Z$</td>
<td>$V_1$</td>
<td>$PS^3Z$</td>
<td>$S^4Z$</td>
</tr>
<tr>
<td>$S^3PV_1$</td>
<td>$V_1$</td>
<td>$PS^3PV_1$</td>
<td>$S^4PV_1$</td>
</tr>
<tr>
<td>$S^4PV_1$</td>
<td>$V_1$</td>
<td>$PS^4PV_1$</td>
<td>$S^5PV_1$</td>
</tr>
<tr>
<td>$S^5PV_1$</td>
<td>$V_1$</td>
<td>$PS^5PV_1$</td>
<td>$S^6PV_1$</td>
</tr>
<tr>
<td>$S^6PV_1$</td>
<td>$V_1$</td>
<td>$PS^6PV_1$</td>
<td>$S^7PV_1$</td>
</tr>
<tr>
<td>$S^7PV_1$</td>
<td>$V_1$</td>
<td>$PS^7PV_1$</td>
<td>$S^8PV_1$</td>
</tr>
<tr>
<td>$S^8PV_1$</td>
<td>$V_1$</td>
<td>$PS^8PV_1$</td>
<td>$S^9PV_1$</td>
</tr>
</tbody>
</table>

Figure 8-9 (Part 4 of 9). Example of axiom addition
(R4.1.5) \[ R' = S^8 V_1 \quad (S_1 S_2 S_3 S_4 = S^4), \quad X = V_1 \]

(a) Replace \( V_1 \) with \( V_2 \):

\[ \text{dom}(V_2) = \text{dom}(V_1) - \text{form}(Z) \]

<table>
<thead>
<tr>
<th>T</th>
<th>V_1</th>
<th>V_2</th>
<th>P</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>V_1</td>
<td></td>
<td>PZ</td>
<td>SZ</td>
</tr>
<tr>
<td>PV_1</td>
<td>V_1</td>
<td>V_2</td>
<td>P^2 V_1</td>
<td>SPV_1</td>
</tr>
<tr>
<td>SZ</td>
<td>V_1</td>
<td>V_2</td>
<td>PSZ</td>
<td>S^2 Z</td>
</tr>
<tr>
<td>SPV_1</td>
<td>V_1</td>
<td>V_2</td>
<td>PSPV_1</td>
<td>S^2 PV_1</td>
</tr>
<tr>
<td>S^2 Z</td>
<td>V_1</td>
<td>V_2</td>
<td>PS^2 Z</td>
<td>S^3 Z</td>
</tr>
<tr>
<td>S^2 PV_1</td>
<td>V_1</td>
<td>V_2</td>
<td>PS^2 PV_1</td>
<td>S^3 PV_1</td>
</tr>
<tr>
<td>S^3 Z</td>
<td>V_1</td>
<td>V_2</td>
<td>PS^3 Z</td>
<td>S^4 Z</td>
</tr>
<tr>
<td>S^3 PV_1</td>
<td>V_1</td>
<td>V_2</td>
<td>PS^3 PV_1</td>
<td>S^4 PV_1</td>
</tr>
<tr>
<td>S^4 PV_1</td>
<td>V_1</td>
<td>V_2</td>
<td>PS^4 PV_1</td>
<td>S^5 PV_1</td>
</tr>
<tr>
<td>S^5 PV_1</td>
<td>V_1</td>
<td>V_2</td>
<td>PS^5 PV_1</td>
<td>S^6 PV_1</td>
</tr>
<tr>
<td>S^6 PV_1</td>
<td>V_1</td>
<td>V_2</td>
<td>PS^6 PV_1</td>
<td>S^7 PV_1</td>
</tr>
<tr>
<td>S^7 PV_1</td>
<td>V_1</td>
<td>V_2</td>
<td>PS^7 PV_1</td>
<td>S^8 PV_1</td>
</tr>
<tr>
<td>S^8 V_2</td>
<td>V_1</td>
<td>V_2</td>
<td>PS^8 V_2</td>
<td>S^9 V_2</td>
</tr>
</tbody>
</table>

(R5.1) \( k = 3 \)

(R3.2) \( \text{dom}(W) = \text{form}(SZ) \)

Figure 8–9 (Part 5 of 9). Example of axiom addition
(R4.2) \( R' = S^8V_2 \quad (g_1 \ldots g_4 = S^4, f_1f_2f_3 = S^3) \)
\( X = SV_2 \)

(a) Replace \( V_2 \) with \( V_3 \): remove form(SZ)

<table>
<thead>
<tr>
<th>( T_1 )</th>
<th>( V_1 )</th>
<th>( V_3 )</th>
<th>( P )</th>
<th>( S )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Z )</td>
<td>( V_1 )</td>
<td>( V_3 )</td>
<td>( PZ )</td>
<td>( SZ )</td>
</tr>
<tr>
<td>( PV_1 )</td>
<td>( V_1 )</td>
<td>( V_3 )</td>
<td>( P^2V_1 )</td>
<td>( SPV_1 )</td>
</tr>
<tr>
<td>( SZ )</td>
<td>( V_1 )</td>
<td>( V_3 )</td>
<td>( PSZ )</td>
<td>( S^2Z )</td>
</tr>
<tr>
<td>( SPV_1 )</td>
<td>( V_1 )</td>
<td>( V_3 )</td>
<td>( PSPV_1 )</td>
<td>( S^2PV_1 )</td>
</tr>
<tr>
<td>( S^2Z )</td>
<td>( V_1 )</td>
<td>( V_3 )</td>
<td>( PS^2Z )</td>
<td>( S^3Z )</td>
</tr>
<tr>
<td>( S^2PV_1 )</td>
<td>( V_1 )</td>
<td>( V_3 )</td>
<td>( PS^2PV_1 )</td>
<td>( S^3PV_1 )</td>
</tr>
<tr>
<td>( S^3Z )</td>
<td>( V_1 )</td>
<td>( V_3 )</td>
<td>( PS^3Z )</td>
<td>( S^4Z )</td>
</tr>
<tr>
<td>( S^3PV_1 )</td>
<td>( V_1 )</td>
<td>( V_3 )</td>
<td>( PS^3PV_1 )</td>
<td>( S^4PV_1 )</td>
</tr>
<tr>
<td>( S^4PV_1 )</td>
<td>( V_1 )</td>
<td>( V_3 )</td>
<td>( PS^4PV_1 )</td>
<td>( S^5PV_1 )</td>
</tr>
<tr>
<td>( S^5PV_1 )</td>
<td>( V_1 )</td>
<td>( V_3 )</td>
<td>( PS^5PV_1 )</td>
<td>( S^6PV_1 )</td>
</tr>
<tr>
<td>( S^6PV_1 )</td>
<td>( V_1 )</td>
<td>( V_3 )</td>
<td>( PS^6PV_1 )</td>
<td>( S^7PV_1 )</td>
</tr>
<tr>
<td>( S^7PV_1 )</td>
<td>( V_1 )</td>
<td>( V_3 )</td>
<td>( PS^7PV_1 )</td>
<td>( S^8PV_1 )</td>
</tr>
<tr>
<td>( S^8V_3 )</td>
<td>( V_1 )</td>
<td>( V_3 )</td>
<td>( PS^8V_3 )</td>
<td>( S^9V_3 )</td>
</tr>
</tbody>
</table>

(R5.2) \( k = 2 \)

(R3.3) \( \text{dom}(W) = \text{form}(S^2Z) \)

(R4.3) Replace \( V_3 \) with \( V_4 \): remove form(S^2Z)

(R5.3) \( k = 1 \)

(R3.4) \( \text{dom}(W) = \text{form}(S^3Z) \)

Figure 8-9 (Part 6 of 9). Example of axiom addition
(R4.4) Replace $V_4$ with $V_5$; remove form($S^3Z$)

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$V_1$</th>
<th>$V_5$</th>
<th>$P$</th>
<th>$S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Z$</td>
<td>$V_1$</td>
<td>$V_5$</td>
<td>$PZ$</td>
<td>$SZ$</td>
</tr>
<tr>
<td>$PV_1$</td>
<td>$V_1$</td>
<td>$V_5$</td>
<td>$P^2V_1$</td>
<td>$SPV_1$</td>
</tr>
<tr>
<td>$SZ$</td>
<td>$V_1$</td>
<td></td>
<td>$PSZ$</td>
<td>$S^2Z$</td>
</tr>
<tr>
<td>$SPV_1$</td>
<td>$V_1$</td>
<td>$V_5$</td>
<td>$PSPV_1$</td>
<td>$S^2PV_1$</td>
</tr>
<tr>
<td>$S^2Z$</td>
<td>$V_1$</td>
<td></td>
<td>$PS^2Z$</td>
<td>$S^3Z$</td>
</tr>
<tr>
<td>$S^2PV_1$</td>
<td>$V_1$</td>
<td>$V_5$</td>
<td>$PS^2PV_1$</td>
<td>$S^3PV_1$</td>
</tr>
<tr>
<td>$S^3Z$</td>
<td>$V_1$</td>
<td></td>
<td>$PS^3Z$</td>
<td>$S^4Z$</td>
</tr>
<tr>
<td>$S^3PV_1$</td>
<td>$V_1$</td>
<td>$V_5$</td>
<td>$PS^3PV_1$</td>
<td>$S^4PV_1$</td>
</tr>
<tr>
<td>$S^4PV_1$</td>
<td>$V_1$</td>
<td>$V_5$</td>
<td>$PS^4PV_1$</td>
<td>$S^5PV_1$</td>
</tr>
<tr>
<td>$S^5PV_1$</td>
<td>$V_1$</td>
<td>$V_5$</td>
<td>$PS^5PV_1$</td>
<td>$S^6PV_1$</td>
</tr>
<tr>
<td>$S^6PV_1$</td>
<td>$V_1$</td>
<td>$V_5$</td>
<td>$PS^6PV_1$</td>
<td>$S^7PV_1$</td>
</tr>
<tr>
<td>$S^7PV_1$</td>
<td>$V_1$</td>
<td>$V_5$</td>
<td>$PS^7PV_1$</td>
<td>$S^8PV_1$</td>
</tr>
<tr>
<td>$S^8PV_1$</td>
<td>$V_1$</td>
<td>$V_5$</td>
<td>$PS^8PV_1$</td>
<td>$S^9PV_1$</td>
</tr>
</tbody>
</table>

(R5.4) $k = 0$

(R2.5) Quit Algorithm R

Figure 8-9 (Part 7 of 9). Example of axiom addition
(A7) Substitute $Z$ for $S^4Z$ in $E(S^3Z, S)$

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$V_1$</th>
<th>$V_5$</th>
<th>$P$</th>
<th>$S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Z$</td>
<td>$V_1$</td>
<td></td>
<td>$PZ$</td>
<td>$SZ$</td>
</tr>
<tr>
<td>$PV_1$</td>
<td>$V_1$</td>
<td>$V_5$</td>
<td>$P^2V_1$</td>
<td>$SPV_1$</td>
</tr>
<tr>
<td>$SZ$</td>
<td>$V_1$</td>
<td></td>
<td>$PSZ$</td>
<td>$S^2Z$</td>
</tr>
<tr>
<td>$SPV_1$</td>
<td>$V_1$</td>
<td>$V_5$</td>
<td>$PSPV_1$</td>
<td>$S^2PV_1$</td>
</tr>
<tr>
<td>$S^2Z$</td>
<td>$V_1$</td>
<td></td>
<td>$PS^2Z$</td>
<td>$S^3Z$</td>
</tr>
<tr>
<td>$S^2PV_1$</td>
<td>$V_1$</td>
<td>$V_5$</td>
<td>$PS^2PV_1$</td>
<td>$S^3PV_1$</td>
</tr>
<tr>
<td>$S^3Z$</td>
<td>$V_1$</td>
<td></td>
<td>$PS^3Z$</td>
<td>$Z$</td>
</tr>
<tr>
<td>$S^3PV_1$</td>
<td>$V_1$</td>
<td>$V_5$</td>
<td>$PS^3PV_1$</td>
<td>$S^4PV_1$</td>
</tr>
<tr>
<td>$S^4PV_1$</td>
<td>$V_1$</td>
<td>$V_5$</td>
<td>$PS^4PV_1$</td>
<td>$S^5PV_1$</td>
</tr>
<tr>
<td>$S^5PV_1$</td>
<td>$V_1$</td>
<td>$V_5$</td>
<td>$PS^5PV_1$</td>
<td>$S^6PV_1$</td>
</tr>
<tr>
<td>$S^6PV_1$</td>
<td>$V_1$</td>
<td>$V_5$</td>
<td>$PS^6PV_1$</td>
<td>$S^7PV_1$</td>
</tr>
<tr>
<td>$S^7PV_1$</td>
<td>$V_1$</td>
<td>$V_5$</td>
<td>$PS^7PV_1$</td>
<td>$S^8PV_1$</td>
</tr>
<tr>
<td>$S^8V_5$</td>
<td>$V_1$</td>
<td>$V_5$</td>
<td>$PS^8V_5$</td>
<td>$S^9V_5$</td>
</tr>
</tbody>
</table>

(A8) Use Algorithm $K$ with $O = \{ S^4Z \rightarrow Z \}$
No new rewrite rules generated.

(A3) Quit Algorithm $A$

Figure 8-9 (Part 8 of 9). Example of axiom addition
Using Algorithm C, rows $SPV_1$, $S^2PV_1$, $S^3PV_1$, ..., $S^7PV_1$ and $S^8V_5$ can be contracted into $SV_5$:

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$V_1$</th>
<th>$V_5$</th>
<th>$P$</th>
<th>$S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Z$</td>
<td>$V_1$</td>
<td></td>
<td>$PZ$</td>
<td>$SZ$</td>
</tr>
<tr>
<td>$PV_1$</td>
<td>$V_1$</td>
<td>$V_5$</td>
<td>$P^2V_1$</td>
<td>$SPV_1$</td>
</tr>
<tr>
<td>$SZ$</td>
<td>$V_1$</td>
<td></td>
<td>$PSZ$</td>
<td>$S^2Z$</td>
</tr>
<tr>
<td>$S^2Z$</td>
<td>$V_1$</td>
<td></td>
<td>$PS^2Z$</td>
<td>$S^3Z$</td>
</tr>
<tr>
<td>$S^3Z$</td>
<td>$V_1$</td>
<td></td>
<td>$PS^3Z$</td>
<td>$S^4Z$</td>
</tr>
<tr>
<td>$SV_5$</td>
<td>$V_1$</td>
<td>$V_5$</td>
<td>$PSV_5$</td>
<td>$S^2V_5$</td>
</tr>
</tbody>
</table>

Figure 8-9 (Part 9 of 9). Example of axiom addition.
Table specifications correspond nicely with implementations in two ways: (1) The partitioning of a type into table rows is often mirrored by the partitioning of a function's input into disjoint cases, treated by disjoint control paths. (2) The existence of a table specification ensures the existence of an implementation—the implementation of tabla1g.
9.1. Program Partitions

From the control structure statements of a function in a class, a set of control paths may be determined.

**DEFINITION 9.1** - Let \( f: T_1 \times \ldots \times T_n \rightarrow T \) be a function in a class. Let \( t_1, \ldots, t_n \) be constants of types \( T_1, \ldots, T_n \).

\[ \text{Path}(f, (t_1, \ldots, t_n)) \] is the sequence of non-control statements executed by \( f \) on input \( (t_1, \ldots, t_n) \). ***

Because we have assumed totality of all class functions, \( \text{path}(f, (t_1, \ldots, t_n)) \) is always a finite sequence of statements. Because no side effects are allowed, execution of control statements does not change the values of any variables.

**DEFINITION 9.2** - Let \( f: T_1 \times \ldots \times T_n \rightarrow T \) be a function in a class. Let \( t_1, \ldots, t_n \) be constants of types \( T_1, \ldots, T_n \).

\[ \text{Func}(f, (t_1, \ldots, t_n)) \] is the constant function defined by execution of the sequence of statements \( \text{path}(f, (t_1, \ldots, t_n)) \). ***

An implementation of \( \text{func}(f, (t_1, \ldots, t_n)) \) is easily constructed by generating assignments of the values of \( t_1, \ldots, t_n \) to the parameters of \( f \) and generating the
sequence of statements in $P$. The values of $t_1, \ldots, t_n$ must be expressed in terms of the primitive types of the language. For example

func(Push,(Zero,Newstack)) (see Figure 5-2) is the sequence

Result.Vals(0) := 0

Result.Tops := 0 + 1.

Such functions are not very interesting by themselves, but combine with each other in a nice way.
DEFINITION 9.3 - Let \( f: T_1 \times \ldots \times T_n \rightarrow T \) and \( g: T_1' \times \ldots \times T_n' \rightarrow T \) be functions with disjoint domain arities:

\[
T_i \cap T_i' = \emptyset \quad i = 1, \ldots, n.
\]

The sum of \( f \) and \( g \) is a function with meaning:

\[
[f + g]: (T_1 \times \ldots \times T_n) + (T_1' \times \ldots \times T_n') \rightarrow T,
\]

where \( T + T' \) denotes the disjoint union of types \( T \) and \( T' \), such that

\[
[f + g](t_1, \ldots, t_n) = f(t_1, \ldots, t_n) \text{ when } t_i \in T_i, \quad i = 1, \ldots, n
\]

\[
[f + g](t_1, \ldots, t_n) = g(t_1, \ldots, t_n) \text{ when } t_i \in T_i', \quad i = 1, \ldots, n.
\]

We denote the meaning of the sum of all functions \( f_i: T_1 \times \ldots \times T_n \rightarrow T, \quad i \in I \), an index set, by

\[
\left[ \sum_{i \in I} f_i \right].
\]

This notation allows us to express a useful lemma.

LEMMA 9.4 - Let \( f: T_1 \times \ldots \times T_n \rightarrow T \) be a function in a class. The sum of the control paths of \( f \) uniquely defines \( f \):

\[
[f] = \left[ \sum_{(z_1, \ldots, z_n) \in (T_1, \ldots, T_n)} \text{func}(f_i(z_1, \ldots, z_n)) \right].
\]
**PROOF** - The set of functions in the sum is the set of all constant functions generated by considering every constant word in form\((fV_1 \ldots V_n)\), where \(V_i\) has domain \(T_i\). The set of constants is disjoint, so the sum is defined. The set of constants covers all values of \(f\), so the equality holds. ***
9.2. Table Partitions

Very few useful functions are written as giant case-statements, with each case a constant function. So, very few useful functions are syntactically divided by the set of all \( \text{func( )} \) functions. On the other hand, very few functions are written with straight-line code, with no control statements. A good programmer strikes a balance between these extremes. Table specifications help define such a balance.

We extend the definition of the path function to sets of input values.

**DEFINITION** 9.5 - Let \( f: T_1 \times \ldots \times T_n \rightarrow T \) be a function in a class. Let \( R = \{ R_i \} \) be a set of constants in \( T_1 \times \ldots \times T_n \). \( \text{Path}(f, R) \) is the set of sequences \( \{ \text{path}(f, R_i) \} \). ***

We intend to use rows of tables for the sets \( R \). Since the rows of a table are disjoint, it is natural to expect the paths of rows to be disjoint.
**DEFINITION 9.6** - Let $f$ be a function in a class of domain arity $T$. Let $R_1$ and $R_2$ be subsets of domain $T$. $R_1$ and $R_2$ are $f$-independent if and only if

$$ \text{path}(f, R_1) \cap \text{path}(f, R_2) = \emptyset. $$

$R_1$ and $R_2$ are independent if and only if they are $f$-independent for all functions $f$ of domain arity $T$ in the class. 

The rows `<Newstack>` and `<Push><N><S>` in the table specification of Stack (see Figure 7-5) are independent in the implementation in Figure 5-2. The sets `{<Zero>}` and `{<Succ><Zero>}` are not independent in the implementation in Figure 5-1.

The degree to which the control structure of a function $f$ in a class corresponds to the row structure of the corresponding table in a table specification is measured by the $f$-independence of the rows in the table.
9.3. Relative Correctness

The division of a function by rows, or sums of rows, leads to a division of its correctness. Since a function is determined by the sum of its components, its correctness is similarly divisible.

**DEFINITION 9.7** - let \( f: T_1 \times \ldots \times T_n \rightarrow T \) be a class function. Let \( F: T'_1 \times \ldots \times T'_n \rightarrow T' \) be a function in a data abstraction. \( \text{Correct}(f,F) = \text{True} \) if there exists an epimorphism

\[
h: (T_1 \times \ldots \times T_n) \rightarrow (T'_1 \times \ldots \times T'_n),
\]

\[
h: T \rightarrow T', \text{ such that}
\]

\[
h(f(t_1, \ldots, t_n)) = F(h(t_1, \ldots, t_n)).
\]

This definition merely adds notation to the notion of correctness defined in chapter 5.
THEOREM 9.8 – Let $f$ be a class function of domain arity $T$. Let $F$ be a function in a data abstraction of domain arity $T'$. Let $R_1$ and $R_2$ be $f$-independent sets of domain $T$.

Correctness of $f$ with respect to $F$ over the disjoint union of $R_1$ and $R_2$ may be factored into its correctness over each:

$$\text{correct}(\text{func}(f,R_1) + \text{func}(f,R_2), F)$$

if and only if

$$\text{correct}(\text{func}(f,R_1), F) \text{ AND } \text{correct}(\text{func}(f,R_2), F).$$

***

PROOF – Since $R_1$ and $R_2$ are $f$-independent, we may construct functions $f_1$ and $f_2$, such that

$$[f] = [f_1 + f_2]$$

and

$$f(R_1) = f_1(R_1), \quad f(R_2) = f_2(R_2).$$

Suppose $f_1$ and $f_2$ are both correct.

Then, there exist epimorphisms $h_1$, $h_2$, such that

$$h_i(f_i(R_i)) = F(h_i(R_i)) \quad i = 1, 2.$$ But, that means there exists an epimorphism

$$h = h_1 + h_2.$$ So $f$ is correct.
Conversely, let \( f \) be correct. Then there exists an epimorphism \( h \), such that \[ h(f(t)) = F(h(t)) \quad \text{for all } t \in T. \]

But, the division of \( T \) into \( f \)-independent sets \( R_1 \) and \( R_2 \) yields

\[ h(f(t_1)) = F(h(t_1)) \quad \text{for all } t_1 \in R_1, \]
\[ h(f(t_2)) = F(h(t_2)) \quad \text{for all } t_2 \in R_2. \]

But,

\[ f(t_1) = f_1(t_1), \quad i=1,2. \]

So,

\[ h(f_1(t_1)) = F(h(t_1)) \quad i=1,2. \]

Therefore, \( f_1 \) and \( f_2 \) are correct. ***

The significance of the theorem is that identification of \( f \)-independent sets allows decomposition of correctness proofs by control paths. When the sets are rows, this means that a function can be proved correct, row-by-row.
10. Summary

Correctness is a relationship between a real object, a specification or a class, and an abstract object, an intended data abstraction. The correctness of two real objects, a specification and a class, with respect to the same abstract object yields a relationship between the two real objects. We have shown that this relationship may be described by a lattice.

Each element of the lattice has a structure—congruence classes. These are used in table specifications. Each row in a table describes a collection, or pattern of congruence classes. The partitioning of congruences into patterns of congruence classes is often mirrored by the partitioning of implementations into control paths. This is useful in software maintenance.

Axiomatic specifications are useful in design of data abstractions, but they are awkward to use in software maintenance for two reasons: (1) The effect of a change is determined by the total context of the specification. That is, all axioms must be considered in making any change. (2) The syntax of a change to a specification provides little assistance in making a corresponding change to an implementation.
The first problem is familiar to programmers who write highly-dependent code: each statement in a program affects and is affected by practically every other statement. A one-line change to such a program may have quite unpredictable results. Structured programming is an attempt to avoid such problems by separation of concerns. Table specifications are an example of "structured specification," where the rows of the tables are the concerns separated.

The second problem with axiomatic specifications is caused by the first. Since most programmers (we hope) use structured programming techniques in implementing data abstractions, changes may be localized. Changes to specifications should also be localized, and in the same way.
11. Bibliography

[ADJ 73]
J. A. Goguen, J. W. Thatcher, E. G. Wagner and J. B. Wright "A junction between computer science and category theory, I (Parts 1 & 2)" IBM RC 4526 1973

[ADJ 75a]

[ADJ 75b]

[ADJ 76c]

[ADJ 77a]
J. A. Goguen, J. W. Thatcher, E. G. Wagner and J. B. Wright "An initial algebra approach to the specification, correctness, and implementation of abstract data types" IBM RC 6487 April 1977

[ADJ 78a]

[ADJ 79a]
J. W. Thatcher, E. G. Wagner and J. B. Wright "More on advice on structuring compilers and proving them correct" IBM RC 7588 April 1979

[Aho & Ullman 72]
Alfred V. Aho and Jeffrey D. Ullman The Theory of Parsing, Translation, and Compiling Prentice-Hall 1972

[Arbib & Manes 75]
Michael A. Arbib and Ernest G. Manes Arrows, Structures, and Functors Academic Press 1975

[Basili 76]
V. R. Basili "The design and implementation of a family of
application-oriented languages" Proc. Fifth Texas Conf. on
Comp. Sci. October 1976 pp 6-12

[Bartussek & Parnas 77]
Wolfram Bartussek and David L. Parnas "Using traces to
write abstract specifications for software modules" Univ.
N. Carolina TR 77-012 1977

[Bauer et al. 79]
F. L. Bauer, M. Broy, H. Partsch, P. Pepper and H.
Woessner "Systematics of transformation rules" in Program
Construction (ed. Bauer and Broy) pp 273-289
Springer-Verlag 1979

[Birkhoff 35]
Garrett Birkhoff "On the structure of abstract algebras"
Proc. Cambridge Phil. Society v 31 part 4 pp 433-454
October 1935

[Birkhoff 67]
Garrett Birkhoff Lattice Theory AMS Colloq. Pub.s v 25
(3rd ed.) 1967

[Birkhoff 70]
Garrett Birkhoff "What can lattices do for you?" in Trends
in Lattice Theory (ed. by J. C. Abbott) 1970

[Birkhoff 73]
Monthly v 80 pp 760-782 1973

[Birkhoff & Frink 48]
Garrett Birkhoff and Orrin Frink, Jr. "Representations of
lattices by sets" Trans. AMS 64 (1948) pp 299-316 1948

[Birkhoff & Lipson 70]
G. Birkhoff and J. Lipson "Heterogeneous Algebras" J.
Combinatorial Theory v. 8 pp. 115-133 1970

[Birkhoff & Lipson 71]
Garrett Birkhoff and John Lipson "Universal algebra and
automata" Proc. Tarski Symp. (AMS) 1971

[Broy et al. 79]
M. Broy, W. Dosch, H. Partsch, P. Pepper and M. Wirsing
"Existential quantifiers in abstract data types" 6th
Colloq. on Automata, Lang. and Programming (ICALP) 1979

[Burstall 69]
R. M. Burstall "Proving properties of programs by

[Burstall 74]
R. M. Burstall "Program proving as hand simulation with a little induction" IFIP 74 1974

[Burstall & Darlington 77]
R. M. Burstall and John Darlington "A transformation system for developing recursive programs" JACM v. 24 n. 1 pp. 44-67 January, 1977

[Burstall & Goguen 77]
R. M. Burstall and J. A. Goguen "Putting theories together to make specifications" IJCAI 77 pp. 1045-1058 1977

[Chang, Kaden & Elliott 78]

[Cohn 65]
Paul M. Cohn Universal Algebra Harper and Row (out of print) 1965

[Correll 76]
C. Correll "Proving programs in several steps of refinement" IBM Research Report RC 6279 November, 1976

[Cremers & Hibbard 76]

[Cremers & Hibbard 77]
Armin B. Cremers and Thomas N. Hibbard "On the formal definition of dependencies between the control and information structure of a data space" Theoretical Computer Science v 5 pp 113-128 1977

[Cremers & Hibbard 78]
Armin B. Cremers and Thomas N. Hibbard "Orthogonality of information structures" Acta Infor. v 9 pp 243-261 1978

[Dahl et al. 70]

[Dahl, Dijkstra & Hoare 72]
[Darringer & King 77]
John A. Darringer and James C. King "Applications of symbolic execution to program testing" IBM RC 6965 1977

[Ehrich 77]
H. -D. Ehrich "Algebraic semantics of type definitions and structured variables" Fund. of Computation Theory Conf. (Springer LN in CS 56) 1977

[Ehrich 78]

[Ehrig, Kreowski & Padawitz 78]
H. Ehrig, H. -J. Kreowski and P. Padawitz "Stepwise specification and implementation of abstract data types" Fifth ICALP (Inter. Conf. on Automata Languages and Programming) Udine, Italy July, 1978

[Ehrig, Kreowski & Padawitz 79]

[Ehrig, Kreowski & Weber 78]

[Ehrig, Kreowski & Weber 79]

[Evans 51]
Trevor Evans "On multiplicative systems defined by generators and relations" Proc. Camb. Phil. Soc. v 47 1951

[Furtado & Kerschberg 76]
A. L. Furtado and L. Kerschberg "An algebra of quotient relations" (draft) PUC/RJ 10/76 September 1976

[Gannon & Rosenberg 79]
J. D. Gannon and J. Rosenberg "Implementing data abstraction features in a stack-based language" Software--Practice and Experience v 9 pp 547-560 1979

[Gerhart 75]
Susan L. Gerhart "Correctness-preserving program
transformations" 2nd ACM POPL 1975

[Giarratana, Gimona & Montanari 76]

[Goguen 75]

[Goguen 77]

[Goguen 78a]

[Goguen 78b]

[Goguen 79a]

[Goguen 79b]
Joseph A. Goguen "Evaluating expressions and proving inductive hypotheses about equationally defined data types and programs" unpublished 1979

[Goguen & Burstall 78]

[Goguen & Nourani 78]
Joseph Goguen and F. Nourani "Some algebraic techniques for proving correctness of data type implementations" UCLA Extended Abstract June 1978

[Goguen & Tardo 77]
[Goguen & Tardo 79]
Joseph A. Goguen and Joseph J. Tardo "An introduction to the implementation of OBJ ..." Proc. SRS Conf. 1979

[Graetzer 68]
George Graetzer Universal Algebra Van Nostrand 1968

[Graetzer 70]
George Graetzer "Universal algebra" in Trends in Lattice Theory (ed. by J. C. Abbott) 1970

[Graetzer 78]
George Graetzer General Lattice Theory Academic Press 1978

[Guessarian 79]
Irene Guessarian "Program transformations and algebraic semantics" Theoretical Computer Science v 9 pp 39-65 1979

[Guttag 75]

[Guttag 77]
John V. Guttag "Abstract data types and the development of data structures" CACM v. 20 n. 6 pp. 396-404 June, 1977

[Guttag & Horning 78]

[Guttag, Horowitz & Musser 76a]
John V. Guttag, Ellis Horowitz and David R. Musser "Abstract data types and software validation" ISI/RR 76-48 August, 1976

[Guttag, Horowitz & Musser 76b]
John V. Guttag, Ellis Horowitz and David R. Musser "The design of data type specifications" ISI/RR 76-49 November, 1976

[Guttag, Horowitz & Musser 77]

[Guttag, Horowitz & Musser 78]
J. V. Guttag, E. Horowitz and D. R. Musser "Abstract data types and software validation" CACM v 21 December 1978
[Hamlet et al. 79]

[Henke 76]

[Henke 77]

[Higgins 63]
Philip J. Higgins "Algebras with a scheme of operators" Math. Nachr. 27 1963

[Hoare 72]

[Hoare 75]

[Howden 78a]

[Kamin 79c]
Sam Kamin "Final data type specifications: a new data type specification method" Submitted to POPL 1979 October 1978

[Katz & Manna 75]
Shmuel Katz and Zohar Manna "A closer look at termination" Acta Informatica v. 5 pp. 333-352 1975

[Knuth & Bendix 70]

[Lampson et al. 77]

[Lankford 75]
Dallas S. Lankford "Canonical inference" Univ. Texas
ATP-32 December 1975

[Lankford 79a]
Dallas S. Lankford "A complete unification algorithm for abelian group theory" Louisianna Tech. MTP-1 January 1979

[Lankford 79b]
Dallas S. Lankford "Mechanical theorem proving in field theory" Louisianna Tech. MTP-2 January 1979

[Lankford & Ballantyne 77]
D. S. Lankford and A. M. Ballantyne "Decision procedures for simple equational theories with commutative-associative axioms ..." Univ. Texas ATP-39 August 1977

[Linger, Mills & Witt 79]
R. C. Linger, Harlan Mills and B. I. Witt Structured Programming Theory and Practice Addison-Wesley -- Chapters 5 and 6 1979

[Liskov & Berzins 77]
B. H. Liskov and V. Berzins "An appraisal of program specifications" MIT CSG Memo 141 April, 1977

[Liskov et al. 77]

[Loveman 76]
David B. Loveman "Program improvement by source to source transformation" 3rd ACM POPL 1976

[MacLane 71]
Saunders MacLane Categories for the Working Mathematician Springer-Verlag 1971

[MacLane & Birkhoff 67]
Saunders MacLane and Garrett Birkhoff Algebra Macmillan 1967

[Majster 77]
Mila E. Majster "Limits of the 'algebraic' specification of abstract data types" SIGPLAN v. 12 n. 10 October, 1977

[Majster 79a]
Mila E. Majster "Data types, abstract data types and their specification problem" Theoretical Computer Science 8 1979

[Majster 79b]
Mila E. Majster "Treatment of partial operations in the algebraic spec. technique" Proc. Specifications for Reliable Software April 1979

[Manes 76]
Ernest G. Manes Algebraic Theories Springer 1976

[Mendelson 64]
Elliott Mendelson Introduction to Mathematical Logic Van Nostrand Reinhold 1964

[Musser 77]

[Musser 79a]
David R. Musser "Abstract data type specification in the AFFIRM system" Proc. Specifications for Reliable Software April 1979

[Musser 79b]

[Nakahara 78]
Hayao Nakahara "Data type logic, a logical basis for hierarchical programming with abstract data types" 3rd IBM Symp. on Math. Foun. Comp. Sci. Japan August 1978

[Nakajima 77]
Reiji Nakajima "Sypes -- partial types -- for program and specification/structuring and a first order system iota logic" Univ. Oslo Inst. Info. RR 22 November 1977

[Nakajima, Nakahara & Honda 78]
R. Nakajima, H. Nakahara and M. Honda "Hierarchical program specification and verification -- a many-sorted approach" Kyoto Univ. RIMS-265 October 1978

[Neumann 62]
B. H. Neumann Special Topics in Algebra: Universal Algebra Courant Inst. NYU Lecture Notes 1962

[Ore 42]

[Partsch & Broy 79]
H. Partsch and M. Broy "Examples for change of types and object structures" in Program Correctness (ed. Bauer and Broy) pp 421-463 Springer-Verlag 1979
[Pepper 79]
P. Pepper "A study on transformational semantics" in
Program Correctness (ed. Bauer and Broy) pp 322-405
Springer-Verlag 1979

[Pierce 68]
Richard S. Pierce Introduction to the Theory of Abstract
Algebras Holt, Rinehart and Winston 1968

[Polajnar 78]
Jernej Polajnar "An algebraic view of protection and
extendability in abstract data types" Ph. D. diss. USC
September 1978

[Robinson & Levitt 77]
L. Robinson and K. Levitt "Proof techniques for
hierarchically structured programs" CACM v. 20 n. 4 pp.
271-283 April, 1977

[Rosenberg & Gannon 77]
Jon Rosenberg and John Gannon "SIMPL-D documentation"
SIN-56 May, 1977

[Scott 70]
Dana Scott "Outline of a mathematical theory of

[Scott 72]
Dana Scott "Mathematical concepts in programming language
semantics" SJCC 1972 1972

[Scott & Strachey 71]
Dana Scott and Christopher Strachey "Towards a
mathematical semantics for computer languages" Proc. Symp.
on Computers and Automata, Poly. Inst. Brooklyn April 1971

[Shaw, Wulf & London 77]
Mary Shaw, William A. Wulf and Ralph L. London
"Abstraction and verification in Alphard: defining and
specifying iteration and generators" CACM v. 20 n. 8 pp.
553-564, August, 1977

[Spitzen & Wegbreit 75]
Jay Spitzen and Ben Wegbreit "The verification and
synthesis of data structures" Acta Info. 4 pp 127-144 1975

[Veloso & Pequeno 79]
Paulo A. S. Veloso and Tarcisio H. C. Pequeno "Don't write
more axioms than you have to: a methodology for the
complete and correct specification of abstract data types;
with examples" (revised and expanded) TR 10/79 May 1979
[Wand 79]
Mitchell Wand "Final algebra semantics and data type extensions" JCSS v 19 n 1 pp 27-44 August 1979

[Wand 80]
Mitchell Wand "Continuation-based program transformation strategies" JACM v 27 n 1 pp 164-180 Jan. 1980

[Wegbreit 76a]

[Wegbreit 76b]
Ben Wegbreit "Goal-directed program transformation" 3rd ACM POPL 1976

[Wegbreit & Spitzen 76]
B. Wegbreit and J. M. Spitzen "Proving properties of complex data structures" JACM v. 23 n. 2 pp. 389-396 April, 1976

[Woessner et al. 79]

[Wulf, London & Shaw 76]

[Wulf et al. 78]
Wulf et al. "An informal definition of Alphard" CMU-CS-78-105 1978

[Zilles 74]

[Zilles 75]
S. N. Zilles "Algebraic specification of data types" MIT CSG Memo 119 pp. 1-12 July, 1975