TYPE REDUCTION
AND
PROGRAM VERIFICATION

by
David R. Jefferson

April 1980

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Submitted to the faculty of Carnegie-Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Copyright (C) 1980 David R. Jefferson

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-78-C-1551.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.
# Table of Contents

ACKNOWLEDGMENTS

SHORT ABSTRACT

LONG ABSTRACT

1. Thesis Summary

1.1 Program Verification Using the Inductive Assertion Method
1.2 Motivation and History
1.3 Summary of Chapter 2: Programming Using Production Systems
1.4 Summary of Chapter 3: The Three-valued Error Logic PFC3
1.5 Summary of Chapter 4: The Formal Language BITZV
1.6 Summary of Chapter 5: Type Reduction and Related Concepts
1.7 Summary of Chapters 6 and 7: The Type Reduction BITZV → BITZ → BIT
1.8 Summary of Chapter 8: An example
1.9 Summary of Chapter 9: Conclusions

2. Production Systems and Tree-Replacement Systems

2.1 Definition of Basic Production System
2.2 Definition of Basic Tree-Replacement Systems
2.3 Weak Correctness Rule for Basic Production Systems
   2.3.1 Proof of Weak Correctness of SCRT
   2.3.2 Weak Correctness of the DNF algorithm
2.4 Markov Production Systems

3. Three-valued Error Logic for Program Verification

3.1 The problem of partial functions and partial predicates
3.2 Three-valued propositional logic
   3.2.1 Constants
   3.2.2 Negation
   3.2.3 The T-, F- and E-functions
   3.2.4 The a-function
   3.2.5 Equivalence and equality
   3.2.6 Conjunction and Disjunction
   3.2.7 Wand, xor, card, cor
   3.2.8 Material Implication
   3.2.9 Conditionals
   3.2.10 Complete sets of three-valued propositional connectives
3.3 Semantics of PC3
   3.3.1 The relationship between Validity and Satisfiability
   3.3.2 Conjunction, Disjunction, Validity, Satisfiability
   3.3.3 Normal Forms
3.4 The Three-valued Partial Functional Calculus: PFC3
   3.4.1 Types
   3.4.2 Generic functions
   3.4.3 Syntax for PFC3: Three-valued, multisorted first order languages
   3.4.4 Semantics of the logic
6.1.5 Stage 1.5: Replace occurrences of \( y[i] \) with occurrences of the elem function
6.1.5.1 Specifications
6.1.5.2 The Algorithm
6.1.5.3 Proof of Weak Correctness
6.1.5.4 Termination

6.1.6 Stage 1.6: Remove nested occurrences of the Restriction function
6.1.6.1 Specifications
6.1.6.2 The Algorithm
6.1.6.3 Proof of Weak Correctness
6.1.6.4 Termination

6.1.7 Step 1.7: Rewrite certain miscellaneous expressions
6.1.7.1 Specifications
6.1.7.2 The Algorithm
6.1.7.3 Proof of Weak Correctness
6.1.7.4 Termination

6.1.8 Put elem terms into canonical form
6.1.8.1 Specifications
6.1.8.2 The Algorithm
6.1.8.3 Proof of Weak Correctness
6.1.8.4 Termination

6.2 Step 2: Put formula into DNF; separate the disjuncts
6.2.1 Specifications
6.2.2 The Algorithm
6.2.3 Proof of Correctness
6.2.4 Termination

6.3 Step 3: Remove \( EQ^n, EQ''^n, ORD^n, ORD''^n \)
6.3.1 Specifications
6.3.2 The Algorithm
6.3.3 Proof of Weak Correctness
6.3.4 Termination

6.4 Step 4: Remove \( EQ^n \) and \( ORD^n \)
6.4.1 Specifications
6.4.2 Notation
6.4.3 The Algorithm
6.4.4 Proof of Weak Correctness
6.4.5 Termination

6.5 Step 5: Hypothesize all possible index order configurations
6.5.1 Simplification of \( T \)-terms; Index Sets
6.5.2 Index Constraint Sets and Index Constraint Formulae
6.5.3 Summary of Step 5
6.5.4 Specifications
6.5.5 The Algorithm
6.5.6 Weak Correctness
6.5.7 Termination

6.6 Step 6: Collapse undefined \( V \)-terms to \( E_v \)
6.6.1 Specifications
6.6.2 The Algorithm
6.6.3 Proof of Weak Correctness
6.6.4 Termination
6.7 Step 7: Remove all occurrences of $E_v$
6.7.1 Specifications
6.7.2 The Algorithm
6.7.3 Proof of Weak Correctness
6.7.4 Termination
6.8 Step 8: Remove zero-length vectors; canonicalize expressions based on $E_v$
6.8.1 Specifications
6.8.2 The Algorithm
6.8.3 Proof of Weak Correctness
6.8.4 Termination
6.9 Step 9: Remove vector overlap
6.9.1 Specifications
6.9.2 Notation
6.9.3 The Algorithm
6.9.4 Proof of Weak Correctness
6.9.4.1 Invariant
6.9.4.2 Postcondition
6.9.5 Termination
6.10 Step 10: Remove occurrences of the restriction function
6.10.1 Specifications
6.10.2 Notation
6.10.3 The Algorithm
6.10.4 Proof of Weak Correctness
6.10.4.1 Invariance
6.10.4.2 Postcondition
6.10.5 Termination
6.11 Step 11: Remove $E_v$
6.11.1 Specifications
6.11.2 Notation
6.11.3 The Algorithm
6.11.4 Proof of Weak Correctness
6.11.5 Termination
6.12 Step 12: Remove (AND)
6.12.1 Specifications
6.12.2 The Algorithm
6.12.3 Proof of Weak Correctness
6.12.4 Termination
6.13 Step 13: Remove elem
6.13.1 Specifications
6.13.2 Notation
6.13.3 The Algorithm
6.13.4 Proof of Weak Correctness
6.13.5 Termination
6.14 Step 14: Remove $\{v\}$
6.14.1 Specifications
6.14.2 The Algorithm
6.14.3 Proof of Weak Correctness
6.14.4 Termination
6.15 Step 15: Remove $ub$ and $ub$
6.15.1 Specifications
6.15.2 The Algorithm
6.15.3 Proof of Weak Correctness
6.15.4 Terminal
6.16 The Type Reduction is Complete

7. Type Reduction: BITZ to BIT

7.1 Step 1: Make arguments to \( \max_z \) and \( \min_z \) be simple \( Z \)-variables
   7.1.1 Specifications
   7.1.2 The Algorithm
   7.1.3 Proof of Weak Correctness
   7.1.4 Termination

7.2 Step 2: Make sure that the \( \min(z), \max(z), z \lt ; \) and \( \{t\} \) functions do not occur nested in one another
   7.2.1 Specifications
   7.2.2 The Algorithm
   7.2.3 Proof of Weak Correctness
   7.2.4 Termination

7.3 Step 3: Remove \( \sim \) and the \( Z \)-conditional functions
   7.3.1 Specifications
   7.3.2 The Algorithm
   7.3.3 Proof of Weak Correctness
   7.3.4 Termination

7.4 Step 4: Put \( \neg T(w) \) in DNF and separate the disjuncts
   7.4.1 Specifications
   7.4.2 The Algorithm
   7.4.3 Proof of Correctness

7.5 Step 5: Remove \( \text{EQ}^n, \text{EQ}^\sim, \text{LEQ}^n, \text{LEQ}^\sim \)
   7.5.1 Specifications
   7.5.2 The Algorithm
   7.5.3 Proof of Correctness

7.6 Step 6: Remove \( \text{EQ}^n \) and \( \text{LEQ}^\sim \)
   7.6.1 Specifications
   7.6.2 The Algorithm
   7.6.3 Proof of Weak Correctness
   7.6.4 Termination

7.7 Step 7: Remove \( z \)
   7.7.1 Specifications
   7.7.2 The Algorithm
   7.7.3 Proof of Weak Correctness
   7.7.4 Termination

7.8 Step 8: Make \( Z \)-arguments to the \( \text{Size} \) and the \( z \lt > \) functions be \( Z \)-variables
   7.8.1 Specifications
   7.8.2 The Algorithm
   7.8.3 Proof of Weak Correctness
   7.8.4 Termination

7.9 Step 9: Hypothesize all possible definedness and equality relations between \( TX \)-terms
   7.9.1 The sets \( TX(w), \text{TCS}(w) \) and \( \text{TCF}(w) \)
   7.9.2 Summary of Step 9
   7.9.3 Specifications
   7.9.4 The Algorithm
   7.9.5 Weak Correctness
   7.9.6 Termination

7.10 Step 10: Remove those terms in \( TX(w) \) which are undefined
   7.10.1 Specifications
7.10.2 The Algorithm  
7.10.3 Proof of Weak Correctness  
7.10.4 Termination  
7.11 Step 11: Remove E
7.11.1 Specifications  
7.11.2 The Algorithm  
7.11.3 Proof of Weak Correctness  
7.11.4 Termination  
7.12 Step 12: Divide TX(w) into equivalence classes; make distinct members of TX(w) take distinct values  
7.12.1 Specification  
7.12.2 Notation  
7.12.3 The Algorithm  
7.12.4 Proof of Weak Correctness  
7.12.5 Termination  
7.13 Step 13: Remove neg and -; put EQ and LEQ into Polynomial Normal Form  
7.13.1 Polynomial Normal Form  
7.13.2 Specifications  
7.13.3 The Algorithm and Proof of Correctness  
7.14 Step 14: Remove all occurrences of size, z\(\Rightarrow\) min and max, thereby removing all occurrences of Z-variables as well  
7.14.1 Notation  
7.14.2 Specifications  
7.14.3 Summary of Step 14  
7.14.4 A note on the Strength of the Result in this Stage  
7.14.5 The Algorithm  
7.14.6 Proof of Correctness  
7.14.6.1 The Precondition Holds  
7.14.6.2 Proof of Invariance for all except Conjunct 2  
7.14.6.3 Proof of Invariance for Conjunct 2  
7.14.6.4 The Postcondition  
7.14.6.5 Termination  
7.14.7 The Introduction of Integer Multiplication  
7.15 Step 15: Remove all remaining Z-terms and Z-related function symbols: =, ≤, 0, +, *, and [f]  
7.15.1 Specifications  
7.15.2 The Algorithm  
7.15.3 Proof of Weak Correctness  
7.15.4 Termination  

8. An Example  
8.1 Step 6.1: Simplify the formula, canonicalize V-terms  
8.1.1 Stage 1: Remove all occurrences of len, min_v, max_v, \(\v v\) and the one-way and two-way conditional functions  
8.1.2 Stage 2: Remove all occurrences of \(\!\!\)  
8.1.3 Stage 3: Remove occurrences of the three-way conditional function  
8.1.4 Stage 4: Remove occurrences of the Assignment function  
8.1.5 Stage 5: Replaces occurrences of v[i] with occurrences of the elem function  
8.1.6 Stage 6: Remove nested occurrences of v[i, i2]  
8.1.7 Stage 7: Rewrite certain miscellaneous expressions  
8.2 Step 2: Put formula into DNF; generate the disjuncts  
8.3 Step 3: Remove EQ, EQ, ORD, ORD
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.4 Step 4:</td>
<td>357</td>
</tr>
<tr>
<td>8.5 Step 5:</td>
<td>359</td>
</tr>
<tr>
<td>8.6 Step 6:</td>
<td>364</td>
</tr>
<tr>
<td>8.7 Step 7:</td>
<td>364</td>
</tr>
<tr>
<td>8.8 Step 8:</td>
<td>365</td>
</tr>
<tr>
<td>8.9 Step 9:</td>
<td>366</td>
</tr>
<tr>
<td>8.10 Step 10:</td>
<td>374</td>
</tr>
<tr>
<td>8.11 Step 11:</td>
<td>376</td>
</tr>
<tr>
<td>8.12 Step 12:</td>
<td>377</td>
</tr>
<tr>
<td>8.13 Step 13:</td>
<td>378</td>
</tr>
<tr>
<td>8.14 Step 14:</td>
<td>380</td>
</tr>
<tr>
<td>8.15 Step 15:</td>
<td>382</td>
</tr>
<tr>
<td>9. Evaluation and Reflections</td>
<td>385</td>
</tr>
<tr>
<td>9.1 Introduction</td>
<td>385</td>
</tr>
<tr>
<td>9.2 Production Systems and Tree-Replacement Systems</td>
<td>385</td>
</tr>
<tr>
<td>9.3 The three-valued error logic PFC3</td>
<td>387</td>
</tr>
<tr>
<td>9.4 The Language BITZV</td>
<td>389</td>
</tr>
<tr>
<td>9.5 Type Reduction and Related Concepts</td>
<td>391</td>
</tr>
<tr>
<td>1. Index of Notation, abbreviation, terminology</td>
<td>393</td>
</tr>
<tr>
<td>1.1 Notation Introduced in Chapter 2</td>
<td>393</td>
</tr>
<tr>
<td>1.2 Notation Introduced in Chapter 3</td>
<td>394</td>
</tr>
<tr>
<td>1.3 Notation Introduced in Chapter 4</td>
<td>397</td>
</tr>
<tr>
<td>1.4 Notation Introduced in Chapter 6</td>
<td>398</td>
</tr>
<tr>
<td>1.5 Notation Introduced in Chapter 7</td>
<td>401</td>
</tr>
<tr>
<td>II. Truth Tables for Three-valued Logic (PFC3)</td>
<td>403</td>
</tr>
<tr>
<td>III. The Language BITZV</td>
<td>405</td>
</tr>
<tr>
<td>III.1 Functions returning Type-B</td>
<td>405</td>
</tr>
<tr>
<td>III.2 Functions returning Type-I</td>
<td>406</td>
</tr>
<tr>
<td>III.3 Functions returning Type-T</td>
<td>407</td>
</tr>
<tr>
<td>III.4 Functions returning Type-Z</td>
<td>408</td>
</tr>
<tr>
<td>III.5 Functions returning Type-V</td>
<td>409</td>
</tr>
<tr>
<td>IV. Example programs written and annotated using BITZV</td>
<td>411</td>
</tr>
<tr>
<td>IV.1 Binary Insertion Sort Program</td>
<td>412</td>
</tr>
<tr>
<td>IV.2 Two-way Merge Program</td>
<td>414</td>
</tr>
<tr>
<td>IV.3 Linear Search Program</td>
<td>416</td>
</tr>
</tbody>
</table>
List of Figures

Figure 3-1: Truth Tables for PC3  
Figure 5-1: BITZV and its sublanguages  
Figure 5-2: Decision Procedure for BITZV
For Sue
ACKNOWLEDGMENTS

I am pleased to acknowledge the help and inspiration that many people have contributed toward this thesis.

First of all I thank my primary advisor, Nori Suzuki, whose earlier paper with me was the immediate ancestor of this thesis. His insistence on mathematical precision at a time when I was offering only half-baked and vague ideas was an important lesson for me. The fact that I have gone overboard in mathematical detail in this thesis shows only that I have learned the lesson too well. This is my fault, not his.

Bill Wulf, who was my advisor for several years, has been an unfailing supporter of me and my work from the beginning. I especially thank him for releasing me goodnaturedly from certain software commitments I had made to the Hydra project so that I could pursue the theoretical areas which led to this thesis. He has since been available often, giving the kind of excellent advice that my future students probably won't take either.

I thank Peter Andrews for exposing me to higher-order logics and for helping me to a deeper understanding of logic in general. He spent many patient hours with me, discussing a wide range of logical topics only sometimes related to this thesis. After Nori Suzuki left C-MU, Peter was often the only one around who understood and cared about the details of some of my questions, and this thesis would have been much more shallow without his advice.

Jon Bentley, my friend and colleague, was at times an active participant in this research. I will long remember the late-night hours that he and Jim Saxe and I spent trying to learn how to construct well-founded sets for use in proving the convergence of the rule systems used in this thesis. We never came to any final understanding of the mathematical issues, but as partial repayment for his help I promise to pay him $1000.00 provided that he never reads, or refers to, any copy of this sentence.

Raj Reddy has my thanks for using his patented arm-around-the-shoulder technique at the
critical time, convincing me to stop extending and to start finishing my thesis.

Although he probably doesn't know it, Robert Floyd of Stanford University asked a critical question of me during a question period at the Conference on Theoretical Computer Science in Waterloo, Ontario in August, 1977. The question was not one I hadn't thought of; but the fact that someone besides me asked it was enough to isolate it for me from the background, and it was the basic question this thesis set out to answer: "Is there a decidable assertion language rich enough to express both the permutation and the ordering properties of simple sorting programs?". Prof. Floyd, the answer is "Yes".

The vast majority of the manuscript was keyed and edited by Sharon Carmack. A quick perusal will reveal that this is surely one of the more lengthy, painful and unpleasant typing jobs that anyone has ever done,¹ and she did it with great care and accuracy. Sadly, I can never do the same for her.

To Brian Reid, the sole author of the Scribe system which formatted this document goes my everlasting praise and admiration. The thought of having to use Pub or some other obsolete document formatter would prevent me from attempting such a difficult formatting job ever again. What formatting consistency that there is in this thesis is entirely due to Scribe, as I have had to put very little thought into it (which is the whole point). I also thank David Lamb and Andy Palay who, from time to time, have helped me do the character drawing that I needed for certain of the mathematical symbols.

Many others have contributed to the extraordinarily productive and friendly atmosphere here at C-MU which has helped me over the years. I just want to mention quickly a few of my closest friends and classmates: Doug Clark, Therese Flaherty, Loretta Guarino, John Gaschnig, Brian Reid, Elaine Rich, Mike Shamos (who thinks this thesis is basically worthless) and Richard Swan.

¹I agree. --s.c.
And finally, above all else, I must do what people have done from time immemorial at the completion of any large task. I must, with deepest gratitude, thank my lover and wife, Sue Brienza. Words cannot.
SHORT ABSTRACT

One of the failures of program verification research has been the inability to build automatic theorem-provers powerful enough to handle formulae that arise as verification conditions for even very simple programs. In this thesis we describe a theorem-proving algorithm which applies to the limited domain of simple searching, merging and sorting programs, but is a decision procedure within that domain.

We first define a formal assertion language called BITZV which is based on five types: Boolean, Integer, Type-T (a totally ordered type), Type-Z (zsets-of-T, like multisets except that elements may occur with negative multiplicities) and Type-V (finite length vectors-of-T of all lengths, and with arbitrary upper and lower index bounds). BITZV is unquantified, but contains a fairly large set of function and predicate symbols including all of the following (and more): all propositional connectives for type Boolean, +, -, = and < for Integers, = and < for Type-T, +, -, = and size for Type-Z, and lb (lower bound), ub (upper bound), min (least element), max (greatest element), \( o[<] \) (vector access), \( <^o, o^o> \) (vector element assignment), \( o[<, .., o>] \) (restriction to a subinterval), ord (orderedness predicate) and = for Type-V. The BITZV language is sufficiently rich to express all of the verification conditions for a variety of simple searching, merging and sorting programs, such as linear and binary search, insertion, selection and bubble sort, and simple k-way merge, recursive merge-sort and Quicksort programs. In the case of the sorting programs it can express both the permutation and the orderedness properties.

The main result of this thesis is a decision procedure for the language BITZV. We present both the algorithm and a lengthy proof of its correctness. The significance of the result is that any program whose expressions and inductive assertions can be written as terms and wffs of BITZV can now in principle be automatically verified, including all of the programs mentioned above.

The decision procedure is organized according to a principle we call Type Reduction. To apply Type Reduction to a multisorted language \( L(T_1, \ldots, T_n) \) having types \( T_1, \ldots, T_n \), is to reduce the validity problem for \( L(T_1,\ldots,T_n) \) to the validity problem for the smaller language
$L(T_1...T_{n-1})$. In the case of BITZV we apply Type Reduction once to reduce its validity problem to that of the language BITZ (which contains no Type-V terms), and then we apply it a second time to further reduce the problem to validity in the language BIT (which contains no Type-V terms or Type-Z terms). It is then a comparatively simple matter to show that BIT is decidable as a consequence of the decidability of the unquantified theories of Presburger arithmetic and of total order.

The decision procedure for BITZV, as written in this thesis, is not really practical. It was optimized for shortness of correctness proof, not runtime efficiency. Much algorithmic engineering would be necessary before a really useful implementation is feasible.
LONG ABSTRACT

The subject of this thesis is the theory of program verification in the Floyd-Hoare-Dijkstra tradition of the inductive assertion method. The most significant result is the definition of a method called Type Reduction for structuring the problem of deciding the validity of formulae that typically arise in program verification. We illustrate Type Reduction twice in solving the validity problem for an unquantified theory called BITZV which is designed specifically to be able to express the verification conditions for a variety of simple searching, merging, sorting and counting programs.

BITZV is formalized in a three-valued version of the predicate calculus called PFC3 which we develop to handle the problem of partial functions and exception behavior (such as zerodivide or array bounds violation.)

The Type Reduction algorithms are written in terms of production systems, and therefore we devise and describe methods for proving the correctness of such programs. We apply this methodology in providing a lengthy proof sketch that our Type Reduction algorithms actually are correct.

The idea of Type Reduction is basically simple. The verification conditions for real programs typically contain terms of several related data types such as integer, Boolean, real, multiset-of-real and vector-of-real. In general, the verification conditions will all be wffs in a multityped language denoted $L(T_1,...,T_n)$, with types $T_1,...,T_n$. Applying the Type Reduction method to the validity problem for a wff $w \in L(T_1,...,T_n)$ means constructing a finite set of wffs $\Omega$ such that the wffs in $\Omega$ involve only types $T_1,...,T_{n-1}$, and hence $\Omega \subseteq L(T_1,...,T_{n-1})$, and such that $w$ is valid if and only if all of the wffs in $\Omega$ are valid. Thus, Type Reduction means reducing the validity problem for a language $L(T_1,...,T_n)$ to that for language $L(T_1,...,T_{n-1})$.

Most of this thesis is devoted to illustrating Type Reduction by applying it to an unquantified language called BITZV having five types (logical sorts). Type-B is Boolean; Type-I is Integer; Type-T is some (any) Totally-ordered type; Type-Z is the set of all finite $\mathbb{Z}$sets-of-T (like multisets except that elements may have negative multiplicity); and Type-V is
the set of all finite-length Vectors-of-T. These five types are related by a large number of function symbols for handling such notions as vector access and assignment, maximum and minimum elements of vectors and zsets, orderedness of vectors, permutation of vectors, equality of vectors and zsets, etc. There is, in fact, a rich enough set of functions in the language BITZV to be able to express, without quantifiers, the inductive assertions and verification conditions for such programs as linear and binary search, simple K-way merge, and various simple sorting programs including versions of merge-sort and Quicksort.

To "solve" the decision problem for BITZV we perform two successive Type Reductions. The first one (Chapter 6) transforms a formula w∈BITZ into a set Ω∈BITZ (the same language without Type-V) in such a way that w is valid if and only if every wff of Ω is. The second Type Reduction (Chapter 7) reduces BITZ to BIT. Both algorithms are lengthy, so we intersperse the steps of the algorithms with proofs that they are correct. We also show, in Chapter 5, how the validity problem for BIT is reducible to the two problems BI and BT. BI is just an unquantified theory of the Integers, and BT is an unquantified theory of total order, so that decision procedures for these two theories can be used to decide any sentence in BITZV.

We thus have the chain of reductions BITZV ⇒ BITZ ⇒ BIT ⇒ (BI, BT). One very important property of our algorithms is that if non-constant multiplication or division is not present in the original w∈BITZV, then it is never introduced by any reduction stage, and consequently none of those formulae of BI to which w is reduced contain non-constant multiplication. The resulting sentences of BI will be unquantified wffs from Presburger arithmetic, and thus decidable. Since BT, the unquantified theory of total order, is also decidable, our algorithms constitute a decision procedure for those wffs of BITZV containing no nonconstant multiplication. It is significant that none of the searching, merging and sorting programs referred to above require non-constant multiplication, either in the programs or in the assertions. Hence the weak correctness problem for such programs, suitably annotated with specifications and loop invariants, is decidable.

Besides defining and illustrating Type Reduction this thesis treats certain other topics as
well. For example, the language BITZV contains some symbols that represent partial functions rather than total functions. The vector access and assignment functions are partial because they are "undefined" when the index is out of bounds, and the zset-minimum function is also partial because it is "undefined" when its argument is the empty zset. The usual predicate calculus is not well-suited to handling partial functions and "error" behavior. We have therefore devised a modification of the predicate calculus in which every type (logical sort) is provided with a distinguished error value for that type. In particular, Type Boolean has an error truth value, denoted error, which is distinct from true and false, and thus our partial function calculus, PFC3, has three truth values. In Chapter 3 we describe this logic, both the propositional and quantificational parts, and illustrate its application to program verification. In particular, we show that "firm correctness" of programs (which is the same as weak correctness but with the additional guarantee that runtime errors such as zerodivide or array bounds error never occur) is easily and naturally formalized using PFC3. We formalize the language BITZV in PFC3, and all programs verifiable in BITZV are in fact automatically verifiable to the firm correctness standard.

The Type Reduction algorithms are actually written in an informal programming language based on the notion of Production Systems. These are nondeterministic looping control structures with pattern-matching facilities. In order to prove the correctness of our Type Reduction algorithms it was essential to develop ways of reasoning about such production systems. Chapter 2 is devoted to these investigations. We give Hoare rules for proving weak correctness of several kinds of production systems, and survey ways of solving the much more difficult problem of proving termination.
I. Thesis Summary

1.1 Program Verification Using the Inductive Assertion Method

The subject matter of this thesis is the theory of program verification using the method of inductive assertions pioneered by Floyd, Hoare and others. Using this method, programs are verified in the following manner. The input to the verification process is the program whose correctness is in question, together with its specifications in the form of a precondition, a postcondition, and sufficiently many loop invariants to "cut" every loop in the program.

The first stage in the classical process is to transform the program and its specifications into a finite set of mathematical formulae called verification conditions. The verification conditions (VCs) are generally statements in a language formalized according to some variation of the Predicate Calculus. They will generally contain function and predicate symbols and free and bound variables corresponding to those found in the program and its specifications. The VCs are constructed in such a way that if all of the VCs are valid formulae (in an appropriate model) then the program is consistent with its specifications. Thus, the problem of proving the program's correctness is reduced to the problem of proving the validity of a set of formulae.

The second stage of the verification process is the theorem prover, which attempts to prove or disprove the validity of the VCs. If it succeeds in proving them all valid, then the program is verified as correct with respect to its specifications. If any of the VCs are disproved, then either the program is incorrect with respect to its pre- and postconditions, or else one or more of the loop invariants must be strengthened. There is a third possibility as well: the theorem prover may be unable to decide the validity of some of the VC's, in which case the correctness of the program is left in doubt.

Proving a program correct (with respect to its specifications) using the inductive assertion method usually means proving weak correctness. A proof of weak correctness is a proof of correctness subject to the provisions that (1) the program does not encounter any error condition such as zerodivide or stack underflow, and (2) that the program eventually halts for
every execution on every input. Weak correctness is opposed to firm correctness where the first provision is deleted, and to strong correctness, where both provisions are deleted.

We will not discuss the technical details of the inductive assertion method any further at this point. We will simply assume that the reader has a solid command of the issues in verification and of the predicate calculus and its variants. Instead, we will devote the next few sections of this summary chapter to a "top-down" description of the material in the thesis.
1.2 Motivation and History

The work in this thesis derives directly from earlier work by Suzuki and Jefferson [Suzuki 80]. In that paper we described a decision procedure for an unquantified formal language which was rich enough to express the inductive assertions needed to prove the permutation property for a wide variety of sorting programs. The formal language included integer variables together with the constants 0, 1, 2, etc. and the + and - operations, and also vector-of-integer variables, together with the vector access operation, v[i], and the assignment-to-a-vector-element operation, denoted <v, i1, i2>. The predicates permitted were ≤ and < for integers, and the perm predicate for vectors, where perm(v1,v2) means that vector v1 is a permutation of vector v2. Equality of vectors was not permitted, and there were severe restrictions on permitted uses of the perm predicate; nevertheless, it was not at all obvious (to me at least) that there should be a decision procedure for this language even under the restrictions imposed. (See Chapter 4).

Our success in finding a decision procedure for a language capable of expressing the VCs for the permutation property of sorting programs naturally led to the question of whether there was a decision procedure for the language which could express the VCs for the ordering property of (simple) sorting programs as well. In fact, these questions and others like them all derived from more general questions about the way the theorem prover for automatic program verifiers should be structured.

The traditional approach to building such theorem provers is to build a general-purpose, complete deduction engine which operates on formulae of the uninterpreted predicate calculus and tries to determine whether or not those formulae are universally valid. Frequently the deduction engine uses some variation of unification and resolution as its central operations. In order to make the general purpose theorem-prover decide interpreted formulae (for example those formulae whose interpretation was fixed to the domains of integers and vectors) the theorem-prover is fed not only the candidate theorem as input, but also a set of axioms defining (or at least constraining) the function and predicate symbols to
behave during deduction like those of the intended interpretation. The theorem-prover then
tries to connect the axioms to the candidate theorem by a chain of inferences, thereby (if
successful) proving the theorem.

Experience with this approach has led to only modest success at best. It seems that
proofs of even "simple" formulae are too long and that the proof-space to be searched is too
large for reasonable computing resources. The amount of progress toward the goal which is
made by a universally valid inference step, even with special knowledge of commutativity,
associativity and transitivity properties included, is simply too small to make a real dent in a
theorem-proving problem. It is apparently necessary to abandon the general-purpose
theorem prover approach in favor of a theorem-prover with detailed "knowledge" of the
mathematical domain, in this case integers and vectors.

This suggests a model of a theorem prover with numerous "packages" of mathematical
knowledge, enclosed in a superstructure that permits cooperation of the various packages in
the proof of a candidate theorem. For example, there might be a package having knowledge
of naive set theory, another with knowledge of vectors, and perhaps a third with knowledge
of stacks. With a rich enough collection of sufficiently competent knowledge packages, and a
reasonably intelligent superstructure, it might be possible to build a theorem-prover capable
of deciding most formulae in the domains of its knowledge within a reasonable amount of time,
at the expense of being completely incompetent outside those areas. It was with this
theorem-prover model in mind that the original work on the decision procedure for the
permutation-language was done. And it was also with this model in mind that that work was
extended to the present thesis.

This thesis is basically a "yes" answer to the question "Does there exist a decision
procedure for a language rich enough to express the verification conditions for the ordering
property of simple sorting programs." In fact, we present a language capable of expressing
both the ordering and the permutation properties of many simple sorting (and other)
programs, and we give a decision procedure for that language. From our experience with this
decision procedure we have abstracted a concept called Type Reduction which seems to hint
at the way the "superstructure" of a knowledge-based theorem-prover might work.
1.3 Summary of Chapter 2: Programming Using Production Systems

The goal of the thesis is to provide a "decision procedure" for the language BITZV as formalized in the logic PFC3. But the procedure must be written in some programming language, and the proof that the procedure is correct must be written in some meta-language. In Chapter 2 we define informally an appropriate programming language and give methods for reasoning about correctness of programs written in it.

When writing a program such as this decision procedure which operates on expressions in a formal language, the first thought is to use a Lisp-like language whose primary data structure (lists) can clearly represent expressions, and whose primary control is recursion on the structure of the expression. However, for the decision procedures I wanted to write such a language seemed too low-level and clumsy. For one thing, the primary operation required seemed to be pattern-replacement, not recursion on the structure of the expression; for another, I was inevitably drawn toward nondeterministic algorithms (the kind of nondeterminism involving a single arbitrary choice among a finite set of alternatives, not the kind in which all choices are selected), and such algorithms are not directly expressible in Lisp.

I finally chose to write the decision procedures in an informal language whose primary control structure is the "production system". Chapter 2 is devoted to defining four types of production systems and to describing ways of proving weak correctness and termination of production systems.

The first of the four kinds of production systems, the Basic Production System (BPS) is simply an unordered set of rules, each of which is of the form

\[ \text{pattern} \Rightarrow \text{action}. \]

The action-part of a rule (production) is simply any side-effect producing statement, including possibly another production system. The pattern-part of a rule is a Boolean expression (containing variables, functions and predicates over the prevailing data types) in which all
occurrences of some variables are distinguished as being **pattern variables**. (In our notation we underline pattern variables.) A pattern is said to be satisfiable if there are any values (of the appropriate types) which can be substituted for the pattern variables to make the pattern true. For example, the pattern

$0 < i \land i < 1$

is not satisfiable (ever) because there is no integer value which can be substituted for $i$ to make the expression true. By contrast, the pattern

$m < i \land i < n$

may or may not be satisfiable at some point in a computation; it all depends on the values of $m$ and $n$, which are not underlined and are thus ordinary variables, rather than pattern variables.

An entire BPS is written using `do(...)od` as delimiters and using `;` as a separator, for example:

```plaintext
do(
    pattern_1 := action_1;
    ...
    pattern_n := action_n
)od
```

Such a program represents a loop in which each iteration consists of selecting an applicable rule and performing the action. The loop continues until there are no applicable rules. More precisely, the interpretation algorithm is this:

1. Check the pattern sides of the rules to see if any of the patterns are satisfiable in the current state of the computation.

2. If none are satisfiable, the loop terminates. From among the patterns which are satisfiable, select one arbitrarily (nondeterministically); call it pattern$_k$.

3. From among the perhaps infinite number of ways that pattern$_k$ might be satisfied, select one, and bind the pattern variables to a set of values which satisfy the pattern.

4. Execute action$_k$ using the new bindings.
5. GOTO Step 1.

In Chapter 2 we discuss the BPS in more detail; in particular, we give a Hoare-like inference rule for proving weak correctness of a BPS. This rule is interesting in its own right, since it differs from the rule used to prove weak correctness of Dijkstra's iterative guarded commands only by the presence of one level of existential quantification in one verification condition. In fact we show that the Basic Production System as we define it is simply a mixture of Dijkstra's iterative guarded command and of the style of rule system or production system frequently used in artificial intelligence systems.

In addition to the BPS we also define three other kinds of production systems: the Markov production system, MPS; the Basic Tree-Replacement System, BTRS; and the Markov Tree-Replacement System. A Markov production system is similar to BPS except that priorities are attached to each production. Whenever more than one production is eligible to execute, the one selected is chosen (arbitrarily) from among those with highest priority. A BPS can thus be viewed as an MPS where every production has the same priority.

A tree-replacement system is a production system which is specialized for expression manipulation. The patterns are expression schema (tree schema), and are satisfiable if they "match" any subexpression (subtree) of the data being manipulated. The action-side of a tree-replacement production is always an expression (tree) schema which is substituted for that portion of the data expression matched by the pattern. Tree replacement systems come in two varieties, basic and Markov, again depending on whether or not the production rules are associated with priorities.

In addition to discussing methods for proving weak correctness of production systems, we also survey ways to prove termination. In the case of tree-replacement systems this is a very difficult subject, and we add nothing to the theory. We merely describe other people's results (especially those of Dershowitz and Manna) which we apply in this thesis.
1.4 Summary of Chapter 3: The Three-valued Error Logic PFC3

The goal of the research at the beginning was to produce a decision procedure for the language BITZV. But early on certain technical problems with the semantics of the language appeared. They all centered on the fact that some of the "functions" defined in BITZV are actually partial functions, undefined for certain values of their arguments. For example, the vector access function \( v[i] \) and the vector assignment function \( <v, i, t> \) are undefined when the index argument is out of range (i.e. when \( i<lb(v) \) or \( ub(v)<i \)), and the \( \min(v) \) and \( \max(v) \) functions are undefined when the argument is an empty vector, i.e. when \( lb(v)=ub(v)+1 \).

When partial function and predicate symbols occur in a formula, serious questions arise concerning the meaning of such semantic notions as "assignment of values to variables," valuation, satisfiability and validity (e.g. is undefined a value?). It is also not clear what meanings should be assigned to the quantifiers, or even to the propositional connectives. (What is the truth value of \( p(x) \land q(x) \) if \( p(x) \) is true and \( q(x) \) is not defined?) Further problems arise when considering questions of valid inference in a logic involving partial functions and predicates.

In order to deal with these issues in a clear and mathematically pleasing way I found it necessary to develop a version of the predicate calculus specifically tailored to the issue of partial functions and predicates. The logic is called PFC3 (Partial Function Calculus, three-valued), and its distinguishing feature is that each type is considered to have a special value in its interpretation called the error value (for that type.) A partial function which is "undefined" for some combination of arguments is interpreted as taking the error value (of the appropriate type) at those arguments. One of the key observations is that because Boolean is considered to be a type, it too is endowed with an error value called error, distinct from the values true and false. Hence, the logic is a three-valued logic.

Most of Chapter 3 is devoted to developing and explaining this logic. We first describe the propositional part of the logic, called PC3, and compare it to the usual two-valued propositional logic, PC2. We give several complete sets of three-valued propositional
connectives, and we show how a wff in the logic can be put in a normal form analogous to Conjunctive (or Disjunctive) Normal Form.

We then extend the propositional logic PC3 to a quantified functional calculus, PFC3. Our concern is mainly with getting consistent definitions of those semantic notions mentioned above which are problematic in the presence of partial functions. We provide numerous examples of valid formulae and valid inferences, but we do not actually axiomatize the logic since such an axiomatization would not be used in the rest of the thesis.

The development of BITZV provided one completely unanticipated side-benefit. One of the annoying defects of the theory of verification is that the inductive assertion method only addresses the issue of weak correctness, i.e. correctness subject to the stipulations that (1) the program does not encounter any run-time error such as array bounds violation, and that (2) the program eventually halts. The second stipulation is understandable, since proofs of convergence generally involve the notion of a well-founded ordering, a notion not conveniently formalizable in the same language as the inductive assertions. However, the first stipulation seems artificial. It has been known from the start that the inductive assertion method is in principle powerful enough to handle the issue of run-time errors; however, a clean and unified formalism for handling the issue has, to my knowledge, never been developed.

The logic PFC3 provides an attractive solution to this problem. In the later part of Chapter 3 we define a notion we call "firm correctness" which is intermediate in strength between weak and strong correctness. Roughly speaking, a program is firmly correct with respect to its specifications if it is weakly correct with respect to them and if it is guaranteed that there will be no run-time errors during its execution, i.e. at no time will a primitive partial function such as array access be applied to an argument for which its value is not defined. We present a set of Hoare-style inference rules for deducing firm correctness of while-programs, rules which are direct analogues of Hoare's rules for weak correctness.
1.5 Summary of Chapter 4: The Formal Language BITZV

The central object of study in the thesis is the formal language we call BITZV, which is a language involving variable symbols, function (and predicate) symbols, and propositional connective symbols combined in the usual way into well-formed formulae. Technically speaking it is an unquantified, multisorted, three-valued, first order language with equality, which comes equipped with an intended interpretation (or class of interpretations). It is unquantified because we do not permit quantifiers, either $\forall$ or $\exists$, in any wff of the language; this amounts to considering every variable to be implicitly universally quantified. The language is multisorted because each variable and expression has a logical sort (or type) associated with it which defines the semantic domain over which the value of the variable or expression ranges. In the case of BITZV there are five sorts (types): Type-$B$ is Boolean; Type-$I$ is Integer; Type-$T$ is some (any) totally-ordered domain; Type-$Z$ is the domain of finite Z-sets-of-$T$ (explained below); and Type-$V$ is the domain of finite-length vectors-of-$T$ (also explained below).

BITZV is three-valued because its semantics is formalized in a logic called PFC3 which has three truth values. These values are true, false and error where the error truth value usually arises as a result of some partial function or predicate being applied to an argument outside of its domain of definition. We will say more about this later.

The language is considered first-order because quantification is not permitted over function or predicate symbols; in fact, quantification is not permitted at all. It is, however, possible to quibble with the characterization of BITZV as first-order because the $Z$-variables and $V$-variables represent functions from Type-$T$ to Type-$I$ and from Type-$I$ to Type-$T$ respectively, and hence they may be viewed as second-order variables. Taking this view leads us to consider the function and predicate symbols taking $V$-terms or $Z$-terms as arguments (such as the vector access function $v[i]$) to be third-order symbols. There is some truth to both views; the precise position in the order hierarchy is not important.

The intended interpretations of BITZV are precisely defined in Chapter 4, but we can
summarize the material here. Type-B is interpreted as the set of Boolean truth values [true, false, error] together with a complete set of (three-valued) Boolean connectives, most of them variations on the familiar two-valued connectives.

Type-I is interpreted as the set \( \mathbb{Z} \) of integers, together with at least the operations of addition, subtraction, the constants 0, 1, 2, etc., and the = and < predicates. Multiplication and division are sometimes included as well, although the Type Reductions given in Chapters 6 and 7 no longer lead to actual decision procedures for BITZV in that case. Other (possibly partial) functions such as \( \log_2 \) may be included in the language of BITZV and in its interpretation, and most of the theory presented in this thesis remains unchanged.

Type-T is intended to be interpreted as a totally-ordered type. There is no single model in mind; rather, any totally ordered domain may serve as an interpretation for Type-T. The reason for this is the desire to express the verification conditions for "abstract" sorting algorithms, i.e. sorting algorithms that depend only on the total-order properties of the elements being sorted, and not on other properties such as the presence or absence of minimal and maximal elements or on the discreteness of the order type. Type-T includes at least the two predicates = and <, and may include other functions and predicates as well.

Type-Z is intended to be interpreted in the domain of finite zsets-of-T. A zset is a combinatorial object similar to a multiset (bag) except that elements may occur in them with negative multiplicities as well as with zero or positive multiplicities. A zset-of-T is a zset whose elements are all of Type-T. The restriction to finite zsets means that the multiplicity of each element must be finite (positive, negative or zero) and that only a finite number of elements may have nonzero multiplicity.

As described in Chapter 4, there are several operations involving zsets which are permitted in BITZV. The empty zset, \( \emptyset \) and zset addition and subtraction are defined in the natural way. We also allow "scalar multiplication" of a zset by an integer (where, for example, \( 4z \) is \( z+z+z+z \)). The only predicates defined for zsets are = and the partial ordering \( \leq \), where \( z_1 \leq z_2 \) if for every \( t \) of Type-T, \( z_1 \) has algebraically fewer copies of \( t \) than \( z_2 \) does.
There are several functions connecting Type-Z to the other types. For example, \( \{t\} \) is a function taking an argument \( t \) and returning a singleton zset with one copy of \( t \) and zero copies of all other elements. Another function with a similar notation is \( \{v\} \), which takes an element of Type-V (vector-of-T) as its argument and returns the zset of elements contained in the vector. Because elements do not occur with a negative multiplicity in vectors, \( \{v\} \) is always a true multiset, a fact which can be expressed as \( \phi \leq \{v\} \). The significance of the \( \{v\} \)-function from the point of view of sorting is that the permutation predicate can be expressed using it. A vector \( v_1 \) is a permutation of another vector \( v_2 \) if and only if \( \{v_1\} = \{v_2\} \).

Type-V is interpreted as the domain of all vectors-of-T. By a vector-of-T we mean a finite interval of integers (determined by a lower bound and an upper bound) and a mapping from the integers in the interval to elements of Type-T. According to this definition, the interval (or its bounds) are part of the value of a vector, and two vectors cannot be equal unless both the corresponding bounds are equal and the corresponding elements (map values) are equal.

Functions involving vectors which are permitted in BITZV include the usual vector access function, denoted \( v[i] \), and the vector element assignment function, denoted \( <v, i, t> \). There are also, of course, the functions \( lb, ub \) and \( len \) which return, respectively, the lower bound, upper bound, and length of the vector.

In order to express some of the ordering properties of vectors the BITZV-language includes the min and max functions, which return the least and greatest elements respectively of a vector, according to the total ordering of Type-T. It also includes the unary predicate \( ord \), whose interpretation is that the elements of its vector argument are sorted in nondecreasing order according to the Type-T predicate \(<\).

This discussion, of course, is only a summary of BITZV; the complete language is described in Chapter 4. But it should be enough to give the flavor of the language, and an indication of how it was designed with sorting programs in mind.
We take pains in Chapter 4 to try to characterize the expressive power of BITZV. About the best short summary we can make of that characterization is this: BITZV is rich enough to express versions of the following sorting algorithms together with their specifications (both the permutation and ordering properties), loop invariants and the resulting VC's: linear and binary insertion sort, selection sort, bubble sort, recursive merge sort and recursive Quicksort. Furthermore, it can express them without the use of multiplication or division (except by constants). Besides sorting programs, BITZV can express programs and assertions for linear and binary searching of a vector, for counting the number of occurrences of an element in a vector, for simple k-way merging (where k is a constant) and many other programs involving vectors and keys.
1.6 Summary of Chapter 5: Type Reduction and Related Concepts

The decision procedure for BITZV (or if multiplication and division are included, the reduction procedure) is very lengthy and relies on much detailed knowledge of the data types involved. But one general organizing principle emerged from the construction of the procedure: the notion of Type Reduction. Chapter 5 is devoted to defining and illustrating the concept of Type Reduction and giving an overall outline of the decision procedure for BITZV.

To apply Type Reduction to the language BITZV is to organize a decision procedure for it according to the following plan. We construct a procedure which inputs an arbitrary wff \( w \subseteq \text{BITZV} \), and outputs a finite set of wffs \( \Omega \subseteq \text{BITZV} \) with the following two properties:

1. None of the wffs in \( \Omega \) contain any terms of Type-V. They therefore contain no V-variables and no occurrences of \( v[i], <v, i, t> \) or \( =_v \) or any other V-related functions or predicates. We denote this by writing \( \Omega \subseteq \text{BITZ} \), where BITZ is the sublanguage of BITZV consisting of all wffs without any occurrences of Type-V terms.

2. \( \equiv w \iff \equiv \Omega \), i.e. the original wff \( w \) is valid in BITZV if and only if all of the wffs in \( \Omega \) are valid in BITZ. Since \( \text{BITZ} \subseteq \text{BITZV} \), validity in BITZ is the same as validity in \( \text{BITZV} \).

Combining a Type Reduction procedure which reduces BITZV to BITZ with a complete decision procedure for BITZ will yield a complete decision procedure for BITZV.

We can apply Type Reduction a second time to the language BITZ, this time to remove Type-Z. This time we give a procedure which takes an arbitrary wff \( w_0 \subseteq \text{BITZ} \) and constructs a finite set \( \Omega \subseteq \text{BIT} \) such that \( w_0 \) is valid if and only if all members of \( \Omega \) are valid. We can thus decide the validity of any wff on \( \text{BIT} \) (and hence \( \text{BITZV} \)) if we can produce a decision procedure for the language BIT.

The phrase "Type Reduction" is a triple entendre. The primary meaning of the phrase is in the sense of "problem reduction", i.e. the problem of deciding sentences in the language BITZV is reduced to the problem of deciding sentences in the smaller language BIT. The
phrase "Type Reduction" also refers to a reduction in the number of different types in the language, from five, in the case of BITZV, to three, in the case of BIT. The third connotation of "Type Reduction" is in the sense of reduction in the "order" or "level" of the type. In the Type Reductions from BITZV to BIT, types V and Z are of "higher order" than the more primitive types B, I and T because the interpretations of types Z and V are over function spaces of the other three types.

After reducing BITZV to BIT, the next step in the decision process is to construct a theorem prover for BIT. In this case we can use another technique which we have named Type Separation. Type Separation means reducing the decision problem for BIT to the two decision problems for BI and BT, thereby separating Type-I from Type-T. BT is simply the unquantified theory of equality (which is decidable) and BI is the unquantified theory of arithmetic which, although not actually decidable when both + and * are included, is reasonably amenable to special procedures designed to handle the cases which usually arise in practice.

The complete algorithm for deciding the language BITZV then can be diagrammed as follows, where \( \Rightarrow \) represents a Type-Reduction and \( \rightarrow \) represents a Type Separation.

\[
\text{BITZV} \Rightarrow \text{BiTZ} \Rightarrow \text{BIT} \rightarrow (\text{BI}, \text{BT})
\]

We take advantage of the well-known decidability of BT and of BI (when multiplication is excluded). Applying Type Reduction and Type Separation to a language such as BITZV does not necessarily lead to a complete decision procedure. But even so, it serves a useful organizational purpose. In this case it serves to isolate any undecidability of BITZV to the sublanguages BI and BT.

The Type Reduction and Separation procedures given in this thesis have one important property which makes them unusually robust. If the scalar multiplication function \( \times_2 \) and the integer functions \( + \) and \( / \) do not occur in the original wff \( w \), then they are not introduced into any of the resulting formulae by either the Type Reduction or the Type Separation procedures. Hence, if the original wff \( w \) does not contain any occurrence of scalar
multiplication, multiplication or division, its truth is decidable.

If a wff is not in BITZV because of occurrences of an integer function symbol (e.g. \(\log_2\)) which is not defined to be in the language, or perhaps occurrences of a function symbol over Type-T (e.g. successor) which is not in the language, the Type Reduction and Type Separation procedures still apply without change. The only difference is that the resulting wffs of language BI would contain occurrences of \(\log_2\) and the resulting wffs of BT would contain occurrences of the successor function. The final stage theorem-provers for BI and BT would have to be able to handle these additional functions, but the Type Reductions and Type Separation are unaffected. The Type Reduction which removes Type-V is sensitive only to those function symbols having Type-V in their signatures, and the Type Reduction from BITZ to BIT is sensitive only of function symbols involving Type Z. In fact, new types could be added to BITZV, and so long as the new type, say Q, did not add any new function symbol having Type-V in its signature, the Type Reduction procedures given in Chapters 6 and 7 could just as well be viewed as reducing BITQZV to BITQ.

For Type Reduction we can characterize this robustness by saying that the part of the language involving Type-V is maximal with respect to this procedure, but the part of the language not involving Type-V, i.e. BITZ, is minimal with respect to the same procedure. It is not clear to what extent such a minimal/maximal property characterizes of Type Reduction in general, but it is certainly a desirable property, and similar properties hold for the Type Reduction in Chapters 7. If it is indeed characteristic, then perhaps Type Reduction should be redefined to require such a minimal/maximal property.
1.7 Summary of Chapters 6 and 7: The Type Reduction \( \text{BITZV} \to \text{BITZ} \to \text{BIT} \)

Chapters 6 and 7, by far the largest of the thesis, are entirely devoted to presenting the Type Reduction procedures for the language \( \text{BITZV} \) and to proving that they are correct. Chapter 6 contains the Type Reduction from \( \text{BITZV} \) to \( \text{BITZ} \), and Chapter 7 contains the Type Reduction from \( \text{BITZ} \) to \( \text{BIT} \). There is no way we can summarize here the lengthy algorithms presented later. We will confine ourselves to a simple introductory discussion.

In each of the two Chapters the procedure is a sequence of about 15 steps, most of which are written as production systems of the kinds defined in Chapter 2. The proofs of correctness for the two procedures also follow (loosely) the methods developed in Chapter 2.

There is absolutely no way that we can summarize here the algorithms in Chapters 6 and 7; they are much too long, and the proofs are much too complex. About all we can do here is indicate some of the flavor of those chapters.

The proof of correctness is an application of the inductive assertion method at one meta-level above that at which program verification is usually done. Sentences in the language \( \text{BITZV} \), which themselves serve as assertions about sorting, searching and merging programs, are simply data objects to the Type Reduction procedures. The assertions which serve as specifications and invariants for the Type Reduction procedure are written (informally) in a metalanguage for \( \text{BITZV} \), and the proof of correctness is carried out (informally) in this metalanguage.

For example, the Precondition and Postcondition for the procedure in Chapter 6, in a somewhat expanded form from the way they are written there, are these:

**Precondition:**

1. \( w \text{ is } w_0 \)

2. \( w(\text{BITZV}) \) (unquantified)

**Postcondition:**
1. $\Omega \subseteq \text{BITZ}$ (unquantified)
2. $\Omega$ is finite
3. $\vdash w_0 \iff \vdash \Omega$

Here the variable $w$ ranges over the set of all wffs constructed from some (any) set of function symbols (including constants and predicates) and variable symbols. We might say that the "type" of $w$ is "wff." The first clause of the Precondition introduces a "constant" wff $w_0$, whose sole purpose is to represent the initial value of $w$, so that in later assertions when the value of $w$ has changed, we can still refer to its initial value. We write the condition $w$ is $w_0$ to assert that $w$ and $w_0$ denote the same wff; the word "is" serves as a meta-equality predicate. We cannot write this assertion as "$w=w_0$" because "$=$" is a symbol in the object language, and according to our notational conventions "$w=w_0$" is the unique wff whose main (root) symbol is $=$ and such that the left and right arguments to the $=$-symbol are the expressions which $w$ and $w_0$ denote (i.e. which are their values). Hence the assertion "$w=w_0$" would be meaningless; it would constitute merely the display of a value without asserting anything about it (rather like asserting "$n+1$" as the precondition to a sorting program).

The second clause of the Precondition simply requires that the wff $w$, whose truth we are trying to decide, be in the language BITZV (without quantifiers). We can actually weaken this condition to allow $w$ to be in certain extensions of BITZV, but in Chapter 6 we ignore this possibility.

In the Postcondition are the conditions which essentially define Type Reduction. $\Omega$ is a variable whose type is "set-of-wffs." The first conjunct requires that $\Omega$ only contain wffs from the language BITZ, i.e. none of the wffs in $\Omega$ contain any V-terms. The second conjunct also requires that $\Omega$ be finite; Type Reduction would be a useless concept if it reduced one problem to an infinite set of other problems.

Finally, in the third conjunct, we require that all of the wffs in $\Omega$ be valid if and only if the original wff $w_0$ was valid. Hence, the validity of $w_0$ can be decided using any procedure that can decide the validity of the wffs in $\Omega$, i.e. by any decision procedure for BITZ. Notice that
the symbol $\leftrightarrow$ is a metasymbol for "if and only if", and must not be confused with any symbol of BITZV, such as $\vDash$. Notice also that the logical symbol for truth (or validity) in a model, $\mathbb{V}$, is used as a metapredicate here to apply either to a single wff $w$, or to a set of wffs, $\Omega$.

To break the proof of correctness into manageable chunks we annotate the Type Reduction procedures with assertions between each pair of adjacent steps. We also, of course, supply a loop invariant for each of the production system steps. As a result, the correctness proof is divided into weak correctness proofs for each step, plus separate termination proofs for each step.

The correctness proofs we give for these procedures are necessarily somewhat informal. For one thing, we often leave logical gaps simply to avoid unbearable detail (or more than unbearable details), and we concentrate only on the central points. Another reason is that there is simply no established metatheory or notations for the concepts we deal with in the proofs, so that some parts of our assertions are written in English. We view the "proofs" of correctness essentially as proof sketches. It is our intention that the assertions and loop invariants be correct and sufficiently strong characterizations of the intermediate states of the algorithms so that a formal proof could be constructed with them as a foundation; but the arguments we supply as proof are somewhat abbreviated.
1.8 Summary of Chapter 8: An example

This chapter is devoted to a lengthy example illustrating a possible execution of the algorithm. We choose three sample formulae from the language BITZV and follow the action of the Type Reduction algorithms in Chapters 6 and 7. Considerable simplification is necessary in tracing the execution because of the two exponential explosions that occur, and because of the extreme amount of nondeterminism encoded in the algorithm.
1.9 Summary of Chapter 9: Conclusions

This final chapter offers an evaluation of the kind of work done in this thesis.

The choice of programming language, Production Systems (Chapter 2), has some advantages and disadvantages. On the one hand the language is powerful enough to express the algorithm in reasonably compact form, i.e. about 200 to 300 productions, depending on how you count the parts of the algorithms that are not written as production rules. Had I used any lower-level language, such as Lisp, it would have been impossible for me to construct Type Reduction algorithms of this size with any confidence at all in their correctness. The pattern-matching notion is natural to the problem and allows a surprisingly simple inference rule so that the weak correctness proofs can be relatively easy.

On the other hand, production systems as formalized here are extremely nondeterministic. This makes the convergence proofs extremely difficult, and in fact they are among the most difficult parts of the proof. Furthermore the large degree of nondeterminism permits some of the possible executions of the algorithm to be extremely inefficient (although "correct"). With nondeterminism there is a tradeoff: in general, the more nondeterminism there is in an algorithm, the easier it is to prove weak correctness and the harder it is to prove convergence. Our choice of Production Systems as a programming language is a de facto commitment in favor of nondeterminism.

The use of the three valued error logic PFC3 (Chapter 3) was dictated by circumstance; I was simply unable to make precise the algorithms and their proofs in any two-valued logic at the time the work started, and developing PFC3 seemed the only way out. I am now convinced that it was also the "correct" way, and that three- (or more) valued logics are inherently superior for program verification than two-valued logic.

There is no getting around the fact that there are at least four semantically different results that can happen when a Boolean expression is evaluated in a program (value true, value false, runtime error, or loop) and that when looping is ignored (in weak or firm
correctness) one still needs three truth values to model the other three possibilities. It is not true that the issues of error values and partial functions and predicates cannot be handled in a two-valued logic. In principle they can, but there is no advantage to doing so. When reasoning about partial functions in any two-valued system, exactly the same exceptions will have to be considered as in doing the same reasoning in a three-valued system. The metatheory, however, will be much more cluttered and ugly.

Perhaps the most satisfying part of the thesis is the attractiveness of the language BITZV. It really does seem to circumscribe a natural, interesting area of mathematics which is useful in program verification. However, it is of course extremely limited in many ways in its expressive power. It would be much more useful if data types involving finite sets and strings could be added.

The concept of Type Reduction is perhaps comparable to the notion of quantifier elimination in usefulness. Both techniques are intended as a tool for finding decision procedures. Both are very elegant when they apply, but there seems to be no way of telling in advance when they are likely to apply. Both of them seem to produce inefficient decision procedures, but examination of the procedures they lead to helps to identify the combinatorial issues in the target theory so that a more efficient decision procedure can be later constructed. In the case of our Type Reductions BITZV → BITZ → BIT, both stages involve an exponential explosion in one of the steps. The composition of the two processes therefore leads to a double exponential explosion. I conjecture that this is not inherent in the problem, and that an algorithm which decomposes the problem differently can reduce BITZV to BIT with only a single exponential explosion.

The Type Reduction algorithms given in this thesis have not been optimized to be space and time efficient; they were optimized, if at all, for shortness of correctness proof. Much work remains to be done before a really practical decision procedure for BITZV can be constructed. I do believe, however, that such a practical procedure is possible. The worst-case formulae for the algorithms presented here are simply not the kind that arise as verification conditions for actual programs, and there seems every reason to expect that this
must be the case.
2. Production Systems and Tree-Replacement Systems

The Type Reduction algorithms in Chapters 6 and 7 are written primarily in an informal programming language whose main control structures are the production system (PS), and a specialization of it, the tree-replacement system (TRS). These are sequential, nondeterministic, noneffective control structures, and in this chapter we define them, introduce some notation for writing them, and - most important - describe the main techniques used to prove the weak correctness of programs written in PS or TRS form.

2.1 Definition of Basic Production System

Both Production Systems and Tree-Replacement Systems will be defined in two forms, Basic and Markov. Here we will describe Basic Production Systems (BPS).

A BPS is a finite unordered set of rules (productions) each of which is of the form

\[ \text{pattern} \Rightarrow \text{action} \]

A collection of \( n \) rules is written

\[
\begin{align*}
\text{do} & \left( \\
\quad & \text{pattern}_1 \Rightarrow \text{action}_1 \\
\quad & \text{pattern}_2 \Rightarrow \text{action}_2 \\
\quad & \quad \ddots \\
\quad & \text{pattern}_n \Rightarrow \text{action}_n \\
\text{od} & 
\end{align*}
\]

The delimiters \( \text{do}(\ldots)\text{od} \) surround the productions, which are separated by the \( \left[ \right] \)-character. The rounded brackets used in the delimiter denote a Basic Production System, and are intended to connote unorderedness. Square brackets in the delimiters, as in \( \text{do}[[\ldots]]\text{od} \), indicate that the productions are ordered, and that the system is a Markov Production System (which we will describe soon).
The pattern-part of a production is similar to a Boolean expression except that the variables occurring in the expression are divided into two classes: pattern variables and non-pattern variables. The non-pattern variables are "free" and their values exist before and after execution of the production system; the pattern variables are "bound" by the production's execution, but do not exist before or after it.

A pattern is satisfiable if there exist values (of the appropriate types) for the pattern variables which make the pattern evaluate to true using the current values of the non-pattern variables. Thus, if \( p(u_1, \ldots, u_m, v_1, \ldots, v_n) \) is a pattern where \( u_1, \ldots, u_m \) are the non-pattern variables (whose values are set elsewhere in the program) and \( v_1, \ldots, v_n \) are the pattern variable, then \( p \) is satisfiable if

\[
\exists v_1 \ldots \exists v_n p(u_1, \ldots, u_m, v_1, \ldots, v_n)
\]

is true. We will use the \( \Delta \)-character to denote pattern-satisfiability, and we therefore define

\[
\Delta p(u_1, \ldots, u_m, v_1, \ldots, v_n) \equiv \exists v_1 \ldots \exists v_n p(u_1, \ldots, u_m, v_1, \ldots, v_n)
\]

The action-part of a production can be any statement in the programming language, including possibly another production system (but not a single production - a production is not a statement, although a production system is). The action-part may refer to any of the pattern or non-pattern variables of the pattern-part, or to any other variables inherited from outside the production system, but it may not perform side-effects on the pattern variables.

The interpretation of a Basic Production System is quite simple. Consider the BPS

\[
do(
\text{pattern}_1 \Rightarrow \text{action}_1)
\]

\[
\text{pattern}_n \Rightarrow \text{action}_n
\]

\) od

When sequential control reaches the BPS the following sequence of events occurs:

1. Some or all of the patterns are checked for satisfiability. A sufficient number
must be checked to find one pattern that is satisfiable, or to determine that none of the patterns are satisfiable. If more than one pattern is satisfiable, one of them is chosen arbitrarily (nondeterministically). We will call the selected pattern \( \text{pattern}_k \).

2. If none of the patterns are satisfiable then execution of the production system terminates, and control passes to whatever statement follows it.

3. The pattern variables in \( \text{pattern}_k \) are bound to values of the appropriate types which satisfy the pattern. If more than one set of values satisfies the pattern, then one set is chosen arbitrarily (nondeterministically) before the binding is performed.

4. The statement \( \text{action}_k \) is executed in the context of the bindings just performed.

5. GOTO step 1.

There are several tacit assumptions in this description. First, we assume that a pattern either is or is not satisfiable; there must be no third choice. We are not going to apply the three-valued error logic which we develop in Chapter 3 to the formalization of production systems (even though to be perfectly consistent we should). Instead, we will assume that no "errors" such as stack underflow or array bounds violation occur in the pattern checking process of Step 1. We will not bother to prove this in each instance, but will confine ourselves to cases where it is "obviously" possible to make the satisfiability test without errors.

We also assume that the question of whether or not a pattern is satisfiable is decidable, and furthermore that values which in fact satisfy a satisfiable pattern are computable. Once again we will not bother to prove this for each production that we write, but will confine ourselves to cases where it is obvious.

These assumptions are both simple and subtle and we will clarify them in the sorting example below.

There is a simple comparison between the notion of a BPS and that of an iterative guarded command as defined in [Dijkstra 76]. Guarded commands are similar to BPS except that instead of having patterns on the left side of rules, they have ordinary Boolean expressions.
Therefore, the task in Step 1 of the execution of a guarded command is to find an expression which is true, rather than a pattern which is satisfiable. Similarly, Step 3 in the execution process is omitted; no binding occurs because there are no pattern variables to bind. But aside from these differences the resemblance is quite strong and we can make the following generalization: the iterative guarded command statements are exactly like BPSs in which the patterns have no pattern variables. In this sense BPSs are richer than guarded commands.

Notice that there are in general two dimensions of nondeterminism in a BPS. First, there is the arbitrary choice of one production out of a set of possibly several with satisfiable pattern-parts. Second, there is the arbitrary choice of values to be bound to the pattern variables out of the possibly infinite set of bindings which might satisfy the pattern. Such heavy nondeterminism leads, as we will see, to compact programs with relatively straightforward weak-correctness proofs, but sometimes extremely difficult termination proofs.

To illustrate the notion of a BPS we now display a vector sorting program. We assume for the moment that the BPS control structure is embedded in an Algol- or Pascal-like language with the usual data types, data structures, control structures and declaration facilities.

```
SORT: array A[1..n] of integer;
patternvar u, v integer;
var temp integer;
do (1≤u ∧ u<v ∧ v≤n ∧ A[u]>A[v] ⇒
) od
```

The BPS at the heart of this program contains only one production, so some of the nondeterminism possible in a BPS is absent from this one. Nevertheless, the example is very instructive.

The pattern-part of the production is

It has the form of a Boolean expression, with pattern variables \( u \) and \( v \), and ordinary variable \( A \). This pattern is satisfiable if there are integer values for \( u \) and \( v \) which make the pattern true. It is satisfiable, therefore, if and only if some pair of elements of the array \( A \) are out of order.

We can follow the execution of this BPS according to the five-step cycle described above:

1. Determine whether or not some pair of elements of the vector \( A \) is out of order (i.e., whether the pattern is satisfiable.)
2. If not, then halt. (The array is sorted.)
3. If so, then bind \( u \) and \( v \) to any two indices such that \( 1 \leq u \wedge u < v \wedge v \leq n \wedge A[u] > A[v] \) is true. There may be many such pairs, so pick one pair arbitrarily.
4. Execute the action side of the rule, exchanging the two elements \( A[u] \) and \( A[v] \).
5. GOTO STEP 1

Later, when we give Hoare-style rules for proving correctness of BPSs, we will prove that this is a sort program. Now, however, we can only informally explain that the program sorts because it finds out of order elements and exchanges them repeatedly until there are no more elements out of order. Because this program does not specify how to find a pair of out-of-order elements, but chooses an arbitrary pair, it can be viewed as an abstraction of all comparison-exchange sorting programs.

2.2 Definition of Basic Tree-Replacement Systems

The production system formalism is frequently used to describe syntactic operations on expressions in some formal language. For this purpose a refined and abbreviated type of BPS called a Basic Tree-Replacement System (BTRS) is useful.

A BTRS is similar to a BPS in that it is an unordered collection of rules, and we denote it similarly, using \( \rightarrow \) instead of \( \Rightarrow \).
\begin{verbatim}
\textbf{do (}
\hspace{1em} \text{pattern}_1 \rightarrow \text{replacement}_1 \hspace{1em} \text{;}
\hspace{1em} \text{;}
\hspace{1em} \text{pattern}_n \rightarrow \text{replacement}_n
\text{)} \textbf{od}
\end{verbatim}

During its execution a BTRS has side-effects on only one variable, which in this thesis we will always call "w" (standing for the phrase "well-formed-formula"). The variable w contains a wff from some formal language. More precisely, it contains a finite ordered tree where the internal nodes represent operators and the immediate descendents of internal nodes represent operands. The execution of one production from a BTRS results in the replacement of some non-empty subtree of w by a new nonempty subtree, which accounts for the phrase "tree-replacement system". So that we will not have to draw tree-diagrams in this thesis we will always use parenthesized expressions instead of trees in our discussions, but it must be understood that replacements can be made only for well-formed subexpressions corresponding to subtrees.

We can best explain the notion of a BTRS by example. Here is a BTRS which puts an unquantified logical expression into Disjunctive Normal Form (DNF). We assume that w is the tree-form of an expression in a first order language.
DNF: do 

(a) \((\neg x)\) \implies x \top

(b) \((\neg (x \land y))\) \implies (\neg x) \lor (\neg y) \top

(c) \((\neg (x \lor y))\) \implies (\neg x) \land (\neg y) \top

(d) \((x \land (y \lor z))\) \implies (x \land y) \lor (x \land z) \top

(e) \((x \lor (y \land z))\) \implies (x \lor z) \lor (y \land z) \top

(f) \((x \land (y \land z))\) \implies (x \land y) \land z \top

(g) \((x \lor (y \lor z))\) \implies (x \lor y) \lor z \top

(h) \((x \lor x)\) \implies x \top

(i) \((x \land x)\) \implies x \top

) od

This BTRS applies (a) double negation elimination, (b)-(c) the DeMorgan laws, (f)-(g) the associative laws, (d)-(e) the left and right \(\land\)-over-\(\lor\) distributive laws, and (h)-(i) the idempotence laws as often as possible to the formula \(w\). The transformations are performed in arbitrary order and at arbitrary points in the expression tree where applicable.

Each pattern is an expression schema (tree schema) in which some of the symbols are symbols from the object language (\(\land\), \(\lor\) and \(\neg\)) and some are pattern variables (\(x\), \(y\), \(z\)) which represent entire subexpressions (subtrees). A pattern such as \(x \land (y \lor z)\) is satisfiable if there exists a subexpression of \(w\) which is a conjunction whose second argument is a disjunction. In other words, \(x \land (y \lor z)\) is satisfiable if it matches any subexpression of \(w\).

A pattern such as \(x \lor x\) is satisfiable if \(w\) has any subexpression which is a disjunction such that the left and right argument expressions (subtrees) are identical; once again, the pattern must match some subexpression of \(w\). Strictly speaking, productions (h) and (i) in which this kind of pattern with repeated pattern variables occurs are not necessary for the transformation to DNF; we include them only to illustrate the possibility. Notice that the
The participation of \( w \) is implicit; it is an implied operand for the entire system.

The replacement-part of a rule is an expression schema involving (possibly) the same pattern variables that occur in the pattern-part. The effect of the execution of a rule is to remove from \( w \) a subtree matching the pattern schema and replace it by a new subtree described by the replacement-part.

The execution cycle for a BTRS, then, is as follows. (Compare with that for a BPS.)

1. Search \( w \) for subtrees matching one of pattern \(_1\) to pattern \(_n\). There may be more than one satisfiable pattern, and if so, choose an arbitrary one. Call it pattern \(_k\).

2. If none of the patterns are satisfiable in \( w \), then the BTRS terminates.

3. Otherwise, bind the pattern variables of pattern \(_k\) to the entire subtrees that correspond to them positionally in \( w \). If there is more than one instance of pattern \(_k\) among the subtrees of \( w \), choose one instance arbitrarily, and bind the pattern variables accordingly.

4. Construct a new subtree represented by replacement \(_k\) using the bindings of Step 3, and substitute it in \( w \) for the subtree chosen in Step 3.

5. GOTO Step 1

Thus, the first production cycle of DNF could transform the following \( w \)

\[
((\neg\neg a) \land (\neg b \lor c)) \land (a \land ((\neg\neg d) \lor (\neg\neg d))) \lor e
\]

into any of the following results (where the new subtree is underlined):

\[
\begin{align*}
\text{a) } & \Rightarrow ((\neg\neg a) \land (\neg b \lor c)) \land (a \land ((\neg\neg d) \lor (\neg\neg d))) \lor e \\
\text{a) } & \Rightarrow ((\neg\neg a) \land (\neg b \lor c)) \land (a \land (\neg\neg d)) \lor e \\
\text{a) } & \Rightarrow ((\neg\neg a) \land (\neg b \lor c)) \land (a \land (\neg\neg d)) \lor e \\
\text{c) } & \Rightarrow ((\neg\neg a) \land (\neg b \lor c)) \land (a \land (\neg\neg d)) \lor e \\
\text{d) } & \Rightarrow ((\neg\neg a) \land (\neg b \lor c)) \land (a \land (\neg\neg d)) \lor e \\
\text{d) } & \Rightarrow ((\neg\neg a) \land (\neg b \lor c)) \land (a \land (\neg\neg d)) \lor e \\
\text{h) } & \Rightarrow ((\neg\neg a) \land (\neg b \lor c)) \land (a \lor (\neg\neg d)) \lor e
\end{align*}
\]
We illustrate here both dimensions of nondeterminism: any of the four productions with satisfiable patterns (a, c, d, h) may execute in the first iteration, and in the cases such as a) and d) where more than one subtree of the formula w matches the pattern, any of the matches may be selected for replacement by the right-hand side of the rule.

In Section 2.3.2 we will prove the correctness of the DNF algorithm.

The semantics of a BTRS is easily reducible to that of a BPS once a few pattern and tree operations are understood. Let P, P₁ and P₂ be tree schema (patterns) in the following definitions.

Definition 1: Let P ⊑ w mean that pattern P matches at least one subtree of w.

Definition 2: Let REPL(w, P₁ :→ P₂) be the result of replacing an arbitrary occurrence of P₁ (chosen nondeterministically) in w by the subtree constructed according to schema P₂ from those subtrees of w matched by the pattern variables of P₁. The command aborts if it is not the case that P₁ ⊑ w or if P₂ contains pattern variables not mentioned in P₁. REPL is a "nondeterministic function".

We take the notion ⊑ of a pattern occurring in a formula (or tree) and the notion REPL of replacing an occurrence of a pattern by another to be primitive. Any detailed examination of them would be belaboring the obvious and lead us astray. We define them because we want to introduce the notations which will be used repeatedly in this thesis, and to exhibit the reduction of the notion of a BTRS to that of a BPS. The following BTRS:

\[
\text{do } ( \\
\quad \text{pattern}_1 :\rightarrow \text{replacement}_1 \\
\quad \vdots \\
\quad \vdots \\
\quad \text{pattern}_n :\rightarrow \text{replacement}_n \\
\text{) od}
\]
has the same net effect as this BPS:

\[
\begin{align*}
\text{do} \quad ( & \\
\quad & \text{pattern}_1 \Rightarrow \mu \leftarrow \text{REPL}(\mu, \text{pattern}_1 \rightarrow \text{replacement}_1) \, \emptyset \\
\vdots & \\
\quad & \text{pattern}_n \Rightarrow \mu \leftarrow \text{REPL}(\mu, \text{pattern}_n \rightarrow \text{replacement}_n) \\
\text{od} & 
\end{align*}
\]

Because of the simplicity of this reduction we will describe proof rules for weak correctness for BPSs only, and we will reason about BTRSs informally through this reduction. Some methods of proving termination which are specifically suited for BTRSs will be described, however.

2.3 Weak Correctness Rule for Basic Production Systems

Because of the similarity between the action of a BPS and the action of Dijkstra's iterative guarded commands, we can modify the Hoare-style weak correctness rule for guarded commands to get one for BPS's. For an iterative guarded command such as

\[
\begin{align*}
\text{do} \\
\beta_1 \Rightarrow S_1 & \emptyset \\
\vdots & \\
\beta_n \Rightarrow S_n & \text{od} 
\end{align*}
\]

and precondition P, postcondition Q, the following proof rule is used:

\[
\begin{align*}
P & \Rightarrow I \\
I \wedge \beta_1 \{S_1\} & I \\
\vdots & \\
I \wedge \beta_n \{S_n\} & I \\
I \wedge \left( \land_{i=1..n} \lnot \beta_i \right) & Q \\
\hline
P \{\text{do } \beta_1 \Rightarrow S_1 \emptyset \ldots \| \beta_n \Rightarrow S_n\} & Q.
\end{align*}
\]
In other words, to prove the weak correctness of $\text{do } \beta_1 \Rightarrow S_1 \mid \cdots \mid \beta_n \Rightarrow S_n \text{ od}$ with respect to the specifications $(P,Q)$ we must construct a loop invariant $I$ which is weaker than precondition $P$, which is "invariant" over each production in the set, and which, when conjoined with the falseness of all of the guards, implies the postcondition, $Q$.

For the corresponding BPS of the form

$$\text{do } \begin{array}{c}
\text{pattern}_1 \Rightarrow \text{action}_1 \mid \\
\vdots \\
\vdots \\
\text{pattern}_n \Rightarrow \text{action}_n
\end{array} \text{ od}$$

the corresponding weak correctness rule which we will use is this:

$$(P \Rightarrow 1 \\
I \land \text{pattern}_1 \{\text{action}_1\} \mid \\
\vdots \\
\vdots \\
I \land \text{pattern}_n \{\text{action}_n\} \mid \\
I \land \land_{i \in 1..n} (\neg \text{pattern}_i) \Rightarrow Q)
$$

$$P \{\text{do } \text{pattern}_1 \Rightarrow \text{action}_1 \mid \cdots \mid \text{pattern}_n \Rightarrow \text{action}_n \} \text{ od } Q.$$ 

It is not hard to show by induction on the length of the computation that this rule is sound for reasoning about BPSs. We conjecture that it is also relatively complete in the sense defined by Cook, [Cook 78] but we have no proof. Fortunately, the use we make of the rule in this thesis depends only on its soundness.

We will apply this rule dozens of times in this thesis, both for BPSs and, through the reduction of the previous Section, to BTRSs. Rather than be completely formal we will apply it informally as illustrated in the next two subsections where we prove the weak correctness of the SORT and DNF algorithms using this rule.
2.3.1 Proof of Weak Correctness of SORT

We will now illustrate the use of rules just quoted to prove the weak correctness of the Sort and DNF programs. In the first case the Hoare-wff we must prove is this:

\[ n = n_0 \land n > 0 \land A[1..n] = A_0[1..n] \]

\[
\{ \text{do } \begin{align*}
1 \leq i \land i < j \land j \leq n \land A[i] > A[j] & \Rightarrow \\
\text{begin } & \text{temp} \leftarrow A[i]; A[i] \leftarrow A[j]; A[j] \leftarrow \text{temp end}
\end{align*} \text{od } \}
\]

\[ n = n_0 \land \text{perm}(A[1..n], A_0[1..n]) \land \text{ord}(A[1..n]). \]

The predicates in the pre- and postconditions have the following meanings:

- \( A[1..n] = A_0[1..n] \) means \( \forall x. ((1 \leq x \land x \leq n) \Rightarrow A[x] = A_0[x]) \);
- \( \text{perm}(A[1..n], A_0[1..n]) \) means that the elements \( A[1] \ldots A[n] \) are a permutation of the elements \( A_0[1] \ldots A_0[n] \);
- \( \text{ord}(A[1..n]) \) means \( \forall x. ((1 \leq x \land x < n) \Rightarrow A[x] \leq A[x+1]) \).

In Chapter 4 we will introduce a different notation for \( \text{perm} \), but this will do for now. Every assertion formula is to be interpreted using the usual two-valued predicate calculus semantics, since we have not yet introduced our three-valued error logic.

Before constructing a loop invariant and computing the verification conditions we really should check that those "tacit assumptions" hold which are required of any production system for it to be effective. First, there must be an algorithm for determining whether or not the pattern is satisfiable. In this case such an algorithm is obvious: one counts from 1 to \( n-1 \) looking for out-of-order elements of the vector \( A \). Second, assuming the pattern is satisfiable there is an algorithm which can find values to satisfy the pattern. In this case the satisfying values are a by-product of the algorithm that tests satisfiability. There is thus no question that the SORT "algorithm" can be made effective.
2.3.1

The issue of errors occurring in the pattern match is a little more subtle. We will take the position here (in Chapter 2) only that a vector variable, such as A in the SORT program, represents a two-way infinitely long row of elements. Consequently the issue of array-bounds violation is avoided: there is no such thing as array bounds violation. It is therefore possible to perform the pattern match operation without risking any run-time errors. Furthermore, no run-time errors will occur during the action-part of the production execution either.

Now that we know the algorithm can be made effective we can proceed with the proof that it is correct. We will use the following formula as loop invariant for SORT.

\[ n = n_0 \land n > 0 \land \text{perm}(A[1..n], A_0[1..n]) \]

For this choice of Invariant our proof rule generates the following verification conditions:

1. \[ n = n_0 \land n > 0 \land A[1..n] = A_0[1..n] \Rightarrow n = n_0 \land n > 0 \land \text{perm}(A[1..n], A_0[1..n]) \]

2. \[ n = n_0 \land n > 0 \land \text{perm}(A[1..n], A_0[1..n]) \land 1 \leq i \land i < j \land j \leq n \land A[j] > A[i] \]
   \[ \begin{align*}
   \text{begin temp} & \leftarrow A[i]; \ A[i] \leftarrow A[j]; \ A[j] \leftarrow \text{temp end}
   
   n & = n_0 \land n > 0 \land \text{perm}(A[1..n], A_0[1..n])
   
   \end{align*} \]

3. \[ n = n_0 \land n > 0 \land \text{perm}(A[1..n], A_0[1..n]) \land \forall i, j. 1 \leq i < j \land j \leq n \land A[j] > A[i] \]
   \[ n = n_0 \land n > 0 \land \text{perm}(A[1..n], A_0[1..n]) \land \text{ord}(A[1..n]) \]

The truth of the first VC is obvious. The second VC could, of course, be expanded into a formula in the logical language without the embedded program, and its truth would follow from the fact that the permutation property is preserved across the exchange of two elements.

The third VC is the interesting one. Notice that some essential quantifiers have been introduced, quantifiers that cannot be removed by the device of transforming to prenex form and dropping left-most V-quantifiers. This is a characteristic property of pattern match operators in programming languages, and a property which, to my mind, permits languages with pattern matching facilities such as I have defined here to be legitimately called "higher level" than languages without them. Notice also that these are not "bounded" quantifiers,
even though $i$ and $j$ appear to be bounded by 1 and $n$, for the simple reason that the value of $n$ is unbounded.

The truth of the third VC is not hard to see once it is realized that the formula

$$n > 0 \land \neg (\exists i \exists j: 1 \leq i < j < n \land A[i] > A[j])$$

is equivalent to

$$n > 0 \land \operatorname{ord}(A[1..n])$$

Proof of this equivalence from first principles requires an induction; however, we are concerned here with illustrating the soundness of the proof rule for BPS's, and not with the difficulty of finding proofs of the VC's generated by the rule.

We have now completed the proof of weak correctness of the SORT program. We will prove termination later.

### 2.3.2 Weak Correctness of the DNF algorithm

For the DNF algorithm, expressed as a BTRS, our task is to prove its correctness with respect to the following specifications.

- $w = w_0$ and $w \in L(\land, \lor, \neg)$ and $w$ unquantified
- $\{\text{DNF-program}\}$

$\operatorname{eval} w = w_0$ and $w \in L(\land, \lor, \neg)$ and $w$ unquantified and $w$ is in DNF

Recall that a BTRS operates on the variable $w$. Here we require that (1) $w$ have as its initial value some tree (wff) $w_0$, (2) that it be a tree (wff) in some first order language whose only propositional connectives are $\land$, $\lor$ and $\neg$, and (3) that it contain no quantifiers.

In the postcondition the first conjunct, $\exists w \equiv w_0$, means that the wff which is the result of concatenating the final value of $w$ (surrounded by parentheses) with an $\equiv$-sign followed by the initial value of $w$ (in parentheses) is universally valid. Notice carefully the implied quoting and concatenation conventions used here, as they will consistently be used scores of times in the rest of this thesis.
The remainder of the postcondition is self-explanatory. It requires that \( w \) remain within the language \( L \), that it remain unquantified, and that upon termination it be in Disjunctive Normal Form. To be concrete we will define a formula to be in DNF if it is of the form:

\[
\forall_{i=1..m} \left( \land_{j=1..n(m)} (p_{ij}) \right)
\]

where association is to the left, and the \( p_{ij} \) are all either atomic formulae or the negations of atomic formulae, and where the usual conventions for empty disjunctions (\texttt{true}) and empty conjunctions (\texttt{false}) apply. However, we will not try to be formal about what it means for a wff to be "of the form . . .", but rather assume that the reader already understands such notions.

Before we prove weak correctness we should check that the DNF algorithm can be made effective. Is it decidable for all of the patterns and for all wffs \( w \) whether or not the patterns are satisfiable in \( w \)? The answer is clearly yes: \( w \) is a finite-size wff, and an exhaustive search over its nodes is sufficient to decide satisfiability of any of the patterns. Furthermore, if a pattern is satisfiable in \( w \), such a search can identify where in the tree the pattern matches, and what subtrees to bind to the pattern variables, and it can do so without any run-time errors. These remarks will hold for all tree-replacement systems, not just the one exemplified as DNF, so we will never again worry about these issues for TRS's.

The proof we will give will be informally argued and is illustrative of the forms of arguments used consistently in Chapters 6 and 7. We use the following loop invariant:

\[
\exists \ w = w_0 \ and \ w \in L(\land, \lor, \neg) \ and \ w \ unquantified.
\]

We must first show that the loop invariant holds at the beginning of execution of the TRS, which is trivial.

Second, we must show that the loop invariant remains invariant across all of the productions. In the case of the first conjunct this follows from the fact that each of the productions represents an identity replacement. Thus, we use the identity schemas:
\[ F \vdash \neg x = x \]
\[ F \vdash \neg(x \land y) = (\neg x \land \neg y) \]
\[ F \vdash \neg(x \land y) = (x \land y) \lor (x \land z) \]
\[ F \vdash (x \land y) \land z = (x \land z) \lor (y \land z) \]
\[ F \vdash x \land (y \land z) = (x \land y) \land z \]
\[ F \vdash x \lor (y \lor z) = (x \lor y) \lor z \]
\[ F \vdash x \lor x = x \]
\[ F \vdash x \land x = x \]

which are valid for any wffs (trees) \( x, y \) and \( z \). The invariance of the other two conjuncts follows from the trivial observations that no operators outside \( L(\land, \lor, \neg) \) are introduced into \( w \) and no quantifiers are introduced by any production.

To show that the Postcondition holds upon termination we note that the first three conjuncts are present in the invariant, and thus the only question is the fourth conjunct, the assertion that \( w \) is in DNF. This follows from the fact that upon termination there can be no occurrences of any of the production patterns in the formula \( w \). If the formula were not in DNF then one of the first seven patterns \((a)-(g)\) would have to match some part of the formula. And if one of the patterns is satisfiable, the BTRS cannot have halted, contradicting the assumption that it did halt. Hence the formula must be in DNF.

Notice that we did not prove that if none of the first seven patterns is satisfiable that the wff \( w \) is in DNF, we just asserted it. Its proof is below the level at which we want to reason in this thesis, and we may from time to time leave holes in the argument of about this size (or smaller).
2.4 Markov Production Systems

The Basic Production System and Tree-Replacement System described in the last few sections is an attractive formalism but it has some theoretical and practical disadvantages. Although the rule for weak correctness proofs is very simple and elegant proving termination of such systems is extremely difficult in practice. The problem is simply that there is too much nondeterminism. In general, proving termination of a nondeterministic program means proving termination of all computations that may be evoked by it. The more computations that can be evoked, the stronger is the theorem that asserts that all computations terminate, and thus the more difficult is the proof.

In addition to the difficulty in proving termination, there are complexity disadvantages to the kind of unbridled nondeterminism permitted by the BPS and BTRS formalism. Consider, for example, the following BTRS for putting a propositional formula (using ¬, ∧, ∨) into Disjunctive Normal Form:

\[
\text{do} \left( \\
\begin{array}{l}
(a) \quad \neg p :\to p \\
(b) \quad \neg(p \land q) :\to \neg p \lor \neg q \\
(c) \quad \neg(p \lor q) :\to \neg p \land \neg q \\
(d) \quad p \land (q \lor r) :\to p \land q \lor p \land r \\
(e) \quad (q \lor r) \land p :\to q \land p \lor r \land p
\end{array}
\right) \text{od}
\]

Let us consider this program applied to the wff
\[\neg((a \land (b \lor (c \land d))))\].

One possible execution applies production (a) and then production (d), yielding the wff
\[(a \land b) \lor (a \land (c \land d))\]

as its result, which is the best we can expect. But another computation of the same program on the same input is this one:
This computation continues for some time, finally resulting in a ridiculously large formula which is, however, equivalent to
\[(a \land b) \lor (a \land (c \land d))\]
and is in fact in Disjunctive Normal Form.

The former computation is evoked when we interpret the BTRS with the rule that production (a) has priority over productions (b) and (c) which in turn have priority over productions (d) and (e), e.g. whenever productions (a) and (b) are both eligible to execute, production (a) is always chosen. The latter computation is executed when the priorities are reversed.

Obviously we have no use for the latter computation. It takes much more time and space and produces an inferior result compared to the first computation. But there is no simple way to write a BTRS which allows the first computation but not the second.

In this case it is obvious that what is needed is to have some way of controlling the nondeterminism. There are too many execution paths possible, most of which are expensive. To deal with both problems with BPS and BTRSs -- difficulty of convergence proof and unwanted expensive computations -- we introduce two new kinds of production system, the Markov Production System (MPS) and the Markov Tree-Replacement System (MTRS). Many variations on PSs and TRSs are possible, but these are the ones found useful for this thesis.

A Markov Production System (MPS) has the following textual structure:
do

\( \{ \text{pattern}_{1,1} \Rightarrow \text{action}_{1,1} \} \)

\[
\vdots
\]

\( \text{pattern}_{1,n_1} \Rightarrow \text{action}_{1,n_1} \)

\( \{ \text{pattern}_{2,1} \Rightarrow \text{action}_{2,1} \} \)

\[
\vdots
\]

\( \text{pattern}_{2,n_2} \Rightarrow \text{action}_{2,n_2} \)

\( \{ \text{pattern}_{m,1} \Rightarrow \text{action}_{m,1} \} \)

\[
\vdots
\]

\( \text{pattern}_{m,n_m} \Rightarrow \text{action}_{m,n_m} \)

\} \) od

Syntactically, an MPS is constructed by taking groups of productions, in this case \( m \) groups, separating them by semicolons and surrounding them with the delimiters `do[...do`. The groups of productions, containing in this case \( n_1, n_2, ..., n_m \) productions respectively, use `\[` as a separator and are surrounded by the large parens `(` and `)`.

Informally we can define the semantics of an MPS as follows. Let us call the above MPS \( M(m) \), and let us denote by \( M(m-1) \) the following smaller MPS which includes only the first \( m-1 \) groups of productions:
\[
\text{do[}
  (\text{pattern}_{1,1} \Rightarrow \text{action}_{1,1}) \circ \\
  \quad \circ \\
  \quad (\text{pattern}_{m-1,1} \Rightarrow \text{action}_{m-1,1}) \\
  \quad \circ \\
  \quad (\text{pattern}_{m-1,n-1} \Rightarrow \text{action}_{m-1,n-1}) \\
\text{]} \text{od}
\]

Then the action of the interpreter on the program \( M(m) \) follows these steps:

1. Execute \( M(m-1) \) to completion.

2. Find one production (nondeterministically) from among the \( m^{\text{th}} \) group (last group) which is eligible and execute it (once). If none of the productions in the last group are eligible then the procedure terminates.

3. GOTO Step 1.

Since Step 1 refers to \( M(m-1) \) which is itself an MPS, this definition of the interpreter's action is recursive. The recursion depth is terminated by the following boundary condition: the MPS

\[
\text{do[}(p_1 \Rightarrow a_1) \circ \\
  \quad \circ \\
  \quad (p_n \Rightarrow a_n) \\
\text{]} \text{od}
\]

is equivalent to the BPS
\[
\text{do}( p_1 \Rightarrow a_1 ) .
\]

\[
\vdots
\]

\[
p_n \Rightarrow a_n
\] \) do .

More formally, we can define the semantics of \( \mathcal{M}(m) \) by viewing it as an abbreviation for the program
\[
\mathcal{M}(m-1);
\]
\[
\text{do}(\text{pattern}_{m,1} \Rightarrow \text{action}_{m,\text{nm}}; \mathcal{M}(m-1)) \) do
\]
where the recursion at the boundary is treated as before.

These definitions may seem a little abstruse, but we can also give a good intuitive account of the semantics. We treat the program \( \mathcal{M}(m) \) as a collection of \( n_1 + n_2 + \ldots + n_m \) productions which have priorities associated with them. The \( n_1 \) productions of the first group have the highest priority, while the \( n_m \) productions in the last group have the lowest priority. Productions in the same group have equal priority. The interpretation, then, of an MPS is the same as that for the corresponding BPS with all \( n_1 + n_2 + \ldots + n_m \) productions except that in each production cycle the eligible productions of higher priority have preference over the eligible productions whose patterns are satisfied on a given cycle, the one of highest priority is executed. If several productions are eligible which are of equal and maximal priority, then one of them is chosen arbitrarily (nondeterministically) for execution. A little thought will convince the reader that the above formal definition is equivalent to this intuitive one; we omit the proof.

The Markov Tree-Replacement System (MTRS) bears the same relation to a MPS as a BTRS does to a BPS. Thus, the patterns in an MTRS are always tree-schemas (or expression-schemas), and the actions are always subtree substitutions (or subexpression substitutions).
We can illustrate the notions of MPS and MTRS with the following new program for putting a propositional wff into DNF.

```
do
  (a) \( \neg p :\rightarrow p \)
  (b) \( \neg(p \land q) :\rightarrow \neg p \lor \neg q \)
  (c) \( \neg(p \lor q) :\rightarrow \neg p \land \neg q \)
  (d) \( (p \land (q \lor r)) :\rightarrow (p \land q) \lor (p \land r) \)
  (e) \( (q \lor r) \land p :\rightarrow (q \land p) \lor (r \land p) \)
```

Here we have the same five productions as before, but organized so that production (a), which removes double negation, has highest priority; DeMorgan's laws (b) and (c) have equal medium priority; and the distributive laws (d) and (e) have equal lowest priority.

This MTRS for DNF is a much better program than the corresponding BTRS. It uses less space and produces a shorter result than the BTRS, and generally uses less time as well.

We can summarize the theoretical relationships between MPSs and BPSs (MTRSs and BPSs) in the following general theorem.

**Theorem 3:** Let \( M \) be an MPS (MTRS) and let \( B \) be the BPS (BTRS) constructed from \( M \) by simply removing all priority restrictions on the productions of \( M \). Then we have the following results:

1. If \( B \) always converges on some input, then so does \( M \).
2. If \( B \) always converges on all inputs, then so does \( M \).
3. A proof that \( B \) converges can thus be regarded as a proof that \( M \) converges.
4. If \( B \) is weakly correct with respect to predicates \( P \) and \( Q \) then so is \( M \).
5. A proof that \( B \) is weakly correct with respect to \( P \) and \( Q \) can thus be regarded as a proof that \( M \) is weakly correct with respect to \( P \) and \( Q \).
6. The shortest proof of (weak correctness, termination, strong correctness) of \( M \) need be no more than a constant number of steps longer than the shortest proof of (weak correctness, termination, strong correctness) of \( B \).
7. In any complexity measure which is a maximum over all inputs (of size n) of the maximum of some function of the computations evoked by that input (e.g. the usual DSPACE and DTIME measures - but not NTIME) the complexity of M is less than or equal to the complexity of B.

Proof: Parts 1, 2, 4 and 7 are trivial consequences of the fact that the computations evoked by M are always a subset of the computations evoked by B.

Parts 3 and 5 follow respectively from Parts 2 and 4. We could add to any proof system for weak correctness, termination or strong correctness the derived inference rule that if B is weakly correct, terminates, or is strongly correct then we can infer that M is also.

Part 6 is a consequence of Parts 3 and 5. □

Of course, the various converses to this theorem do not, in general, hold.

In the course of reasoning about correctness of our algorithms in later chapters we will make tacit use of this theorem.
3. Three-valued Error Logic for Program Verification

In this chapter we will describe a three-valued propositional logic called $PC_3$ and a three-valued partial function logic called $PFC_3$ and we will illustrate its relation to program verification. Since BITZV is formalized in this logic, it is important for the reader to become familiar with it in order to understand the rest of this thesis. In Section 3.1 we will discuss the reasons why this three-valued logic seems more appropriate for program verification than the usual predicate calculus logic. In Sections 3.2 to 3.4 we describe $PC_3$ and $PFC_3$ in detail. In Section 3.5 we develop a Hoare-style calculus of "firm correctness" of programs.

Three-valued logic is not new. Its application to programming language semantics dates at least as far back as 1963 [Mccarthy 63]. What does seem to be new is its application to the problems of partial functions error, handling and "firm correctness", as well as our wholesale commitment to three-valued logic in preference to two-valued logic as the more appropriate formalism for program verification.

3.1 The problem of partial functions and partial predicates

One of the most annoying problems in program verification is the proper handling of partial functions and predicates, i.e. functions and predicates that are defined for some arguments and undefined for others. We are not referring especially to the partial functions and predicates which arise from procedures with infinite loops or infinite recursion, but more specifically to those partial functions, such as integer division, which are undefined for some arguments and which serve as the interpretation for primitive function (or predicate) symbols in the programming and assertion languages.

Partial functions are quite common. Besides division (undefined when the denominator is 0) there are array access and assignment (undefined when the index is out of bounds), set minimum (undefined when the set is empty), popstack (undefined for the empty stack) and a large number of others, often involving data structures which are either "full" or "empty". Even ordinary addition and multiplication (of integers or reals) may be considered partial functions when overflow or underflow is a concern. Primitive partial predicates apparently
Occur infrequently, but atomic formulae composed from partial functions, such as \( x/y = z \) or \( v[i] = \text{key} \) are quite common, and represent partial predicates on the variables occurring in them.

The problem with partial functions and predicates is that the usual predicate calculus, which is the most frequently used logical formalism for program verification, requires that function and predicate symbols be interpreted by total functions and relations. The predicate calculus simply does not directly address the issue of partial functions and predicates. Furthermore, there is apparently no very simple way to patch things so that it does.

Of course we are not claiming that the expressive power of the predicate calculus is fundamentally inadequate to the task of formalizing the properties of partial functions. We are saying only that if partial functions are to be permitted as the interpretations of function symbols, and if the semantics of logic and that of programs are to remain compatible, then major changes to the semantics of the predicate calculus are necessary.

One might consider formalizing the notion of partial functions and predicates through using the idea of approximation.

**Definition 1:** If \( f: \mathbb{D}^n \to \mathbb{D} \) is a partial function defined on \( C \subseteq \mathbb{D}^n \), and \( f^*: \mathbb{D}^n \to \mathbb{D} \) is a total function such that for all \( <d_1, \ldots, d_n> \in C \), \( f(d_1, \ldots, d_n) = f^*(d_1, \ldots, d_n) \) then we say \( f^* \) is a total approximation to \( f \).

**Definition 2:** If \( p \subseteq \mathbb{D}^n \) is a partial relation defined on \( C \subseteq \mathbb{D}^n \), and \( p^* \subseteq \mathbb{D}^n \) is a total relation such that for all \( <d_1, \ldots, d_n> \in C \), \( p(d_1, \ldots, d_n) = p^*(d_1, \ldots, d_n) \) then we say \( p^* \) is a total approximation to \( p \).

Notice that in these definitions we view the partial function as more "accurate" than the total function, contrary to the usage of some other researchers.

There are at least two ways to define the semantics of partial functions and predicates in terms of approximations. We will illustrate these using the integer division partial function as
our model, but it should be clear that any other partial function or predicate would encounter similar problems.

The first way is to choose a particular total function, which approximates division, say

\[ x \text{ div } y = \begin{cases} \text{ if } y = 0 \text{ then } x \text{ else } x/y \end{cases} \]

and use it directly as the interpretation for the /-symbol. This choice can be captured axiomatically with the following two axioms.

\[ y \neq 0 \implies (x/y = z \equiv x \geq y \cdot z \wedge x < y \cdot (z+1)) \]

\[ x/0 = x \]

This choice, or any other similar choice, however, has unfortunate consequences. It requires that both

\[ x/0 = x/0 \quad \text{and} \]

\[ a=b \implies (a\cdot a - b\cdot b)/(a-b) = 0 \]

be considered valid for the real numbers. Such a choice might not look so bad until you note that the validity of \( x/0 = x/0 \) means that the program

\[
\begin{align*}
\text{if } x/0 &= x/0 \\
&\text{then} \\
&S_1 \\
&\text{else} \\
&S_2
\end{align*}
\]

must execute the then branch of the conditional, whereas that is emphatically not the intended semantics. The program should abort.

An alternative semantics for partial functions and predicates, using the notion of approximation in a different way, is to say that a sentence \( w \) involving / holds in the standard model of arithmetic if and only if for all total approximations div to division, \( w \) holds with / interpreted as div. Such a choice corresponds to the sentences deducible from the axiom

\[ y \neq 0 \implies ((x/y = z) \equiv x \geq y \cdot z \wedge x < y \cdot (z+1)) \]

by itself (i.e. without any axiom describing division by 0) together with a complete (non-RE)
axiomatization of the rest of arithmetic.

Such a semantics is a substantial change from the usual first-order semantics. For example, it entails that neither $1/0 = 3$ nor $1/0 \neq 3$ holds (i.e. that both are false), since neither is true for all approximations to division. It also entails that any property true of all reals is true of the results of division by zero, so that

$$x/0 = x/0,$$

$$x/0 + y/0 = y/0 + x/0$$

still hold.

The implications for programs of this latter kind of approximation semantics are straightforward and contrary to our intentions. The program

```plaintext
if x/0 = x/0
    then
        S_1
    else
        S_2
```

would always execute the then branch, while

```plaintext
if x/0 = y/0
    then
        S_1
    else
        S_2
```

would execute the then branch if $x=y$ and the else branch otherwise. In both cases the intended semantics is abortion.

The point of this discussion is simply this: if we define any semantics for partial functions and predicates in such a way that partial functions take ordinary, albeit arbitrary, elements of the underlying domain as values where they "should" be undefined, and partial predicates take on ordinary truth values where they "should" be undefined, we do not properly capture the intended semantics for partial functions in programs. Consequently we seem to have two choices: either we must give up the desirable close connection between the semantics of
expressions in programs and the semantics of the same expressions in the underlying logic, or we must modify the semantics of the underlying logic in such a way that partial predicates need not always take one of the two truth values \{true, false\} and partial functions need not always take "ordinary" domain elements as a value. We choose to do the latter, and are thus led to an at least three-valued logic. We will begin describing three-valued logic in detail in the next section, but we will insert a brief overview here.

In this logic there are three distinct and coequal truth values: true, false and error. Depending on the application the third truth value might be named "unknown", "don't care", "undefined", or even "loop", but we use the name error to reflect our intention that it help represent zero divide, stack underflow and other such erroneous events. What are usually considered partial predicates are modelled (interpreted) as total three-valued predicates, taking the truth value error where they are informally considered to be undefined.

Similarly, the interpretation of every data type (logical sort) will consist of a non-empty set, together with a distinguished element of the set called the error element. A partial function will be represented as a total function whose value is the error element wherever the partial function is "undefined". The data type integer is thus interpreted as the set \( \mathbb{Z} \cup \{ e_{\bot} \} \), where \( e_{\bot} \) is a value distinct from all members of \( \mathbb{Z} \). Division is interpreted as the function

\[
x \div y = \text{if } y = 0 \lor x = e_{\bot} \lor y = e_{\bot} \text{ then } e_{\bot} \text{ else } x/y.
\]

(The notation in the previous line is informal; we will formally define =, \lor and if-then-else later.)

We define in the next sections three-valued Boolean functions which play a role analogous to those of \( \land, \lor, \neg, \Rightarrow \) and \( = \) of the ordinary two-valued propositional calculus, and we also define several logical operators with no analogy in two valued logic. We also redefine the meanings of the quantifiers \( \forall \) and \( \exists \), and all of the other semantic notions such as interpretation, model, satisfiability and validity.
3.2 Three-valued propositional logic

In this section we will describe a three-valued propositional calculus called PC3. It will be the propositional part of the three-valued functional calculus PFC3 of the next section. We will describe only its semantics, without developing an axiomatization.

The language of three-valued propositional logic, \( L_{PC3} \), is constructed from a denumerable set of propositional variables \( P = \{ p, p_1, p_2, ..., q, q_1, q_2, ..., r, r_1, r_2, ... \} \) and the following truth functional connectives:

- **0-ary**: true, false, error
- **1-ary**: T, F, E, \( \neg \), #
- **2-ary**: \( \land, \lor, \text{wand}, \text{wor}, \text{cand}, \text{cor}, (o \mid o), \neq, \equiv, \supset \)
- **3-ary**: ( \( o \mid o \mid o \) )
- **4-ary**: ( \( o \mid o \mid o \mid o \) )

We will write zero-ary functions without the empty argument list, i.e. true rather than \( \text{true}() \). Unary functions will be prefix; binary functions (except the conditional) will be infix; and the conditional functions will be written with internal arguments, as \( (p \mid q) \), \( (p \mid q_1 \mid q_2) \) or \( (p \mid q_1 \mid q_2 \mid q_3) \). The language \( L_{PC3} \) of three-valued propositional logic is the set of all well-formed expressions constructed from propositional variables and these 20 function symbols. We will, as usual, drop most parentheses for readability.

Before proceeding with the description of the logic we will give intuitive explanation of the truth functional operators we have introduced. During the discussion the reader will want to consult the truth tables for the propositional operators which are given in Figure 3-0.

3.2.1 Constants

The zero-ary functions true, false and error are simply constants representing the three truth values. Although one can formalize propositional logic without them, for mechanical reasoning they are practically indispensable.
### 1-ary:

<table>
<thead>
<tr>
<th>T</th>
<th>F</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>E</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

### 2-ary:

<table>
<thead>
<tr>
<th>∧</th>
<th>t</th>
<th>f</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>E</td>
<td>F</td>
<td>T</td>
<td>E</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>∨</th>
<th>t</th>
<th>f</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>E</td>
<td>T</td>
<td>T</td>
<td>E</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>✷</th>
<th>t</th>
<th>f</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>E</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>≈</th>
<th>t</th>
<th>f</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>E</td>
<td>T</td>
<td>T</td>
<td>E</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>≥</th>
<th>t</th>
<th>f</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>E</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

### 4-ary:

\[
(p \mid q_1 \mid q_2 \mid q_3) \triangleq (T(p) \land q_1) \lor (F(p) \land q_2) \lor (E(p) \land q_3)
\]

### 3-ary:

\[
(p \mid q_1 \mid q_2) \triangleq (p \mid q_1 \mid q_2 \mid \text{error})
\]

### 2-ary:

\[
(p \mid q) \triangleq (p \mid q \mid \text{error} \mid \text{error})
\]

**Figure 3-1: Truth Tables for PC3**
3.2.2 Negation

The negation function, \( \neg \), is the closest three-valued analogue to the usual two-valued negation. It maps \text{true} to \text{false} and \text{false} to \text{true}, as expected, and it assigns \( \neg \text{error} = \text{error} \).

3.2.3 The T-, F- and E-functions

In PC2 one asserts "p is true" by simply writing "p". There is no way to assert directly "p is false", but the equivalent effect can be achieved in PC2 by writing "\(-p\)" which asserts that "the negation of p is true". Because we have three truth values in PC3 the situation is more complicated. We need ways of asserting "p is true", "p is false" and "p is error". An examination of the table for three-valued negation should convince the reader that those three possibilities will not be expressible using only "p" and "\(-\)". We therefore introduce the unary operators T, F and E where T(p) means "p is true", F(p) means "p is false" and E(p) means "p is error". Each of these truth functions returns a value which is always either \text{true} or \text{false}. Since they never take the value \text{error}, they are sometimes used to convert a statement which could be three-valued into one which is actually two-valued.

3.2.4 The \#-function

The last unary operator, \#, should be read "is defined". The expression \#(p) is \text{true} whenever p has one of the two truth values \text{true} or \text{false}, and it is \text{false} when p has the value \text{error}. The reading "is defined" is chosen because in the partial functional calculus of the next section the truth value \text{error} usually arises when some partial function is undefined, i.e. takes an error value.

The \#-operation is definable from other operations, and is thus technically dispensable. The identity

\[ \#p = T(p) \lor F(p) \]

shows that it is easily expressible from T, F and \lor. (We will describe \approx and \lor shortly). However, we keep \# as a primitive operation because it corresponds, for type Boolean, to the
generic "is defined" operators PFC3 that we describe in Section 3.3.

3.2.5 Equivalence and equality

A key operation, which will be defined for all types (not just Boolean), is equivalence, denoted by $\equiv$. The sentence "p $\equiv$ q" is true when p and q have the same truth value, and false otherwise. This operator is important because it plays the same inference role in three-valued logic as ordinary equivalence does in two-valued logic. If

$\vdash A$

and A contains B as a subexpression ($B \alpha A$) and

$\vdash B \equiv C$

then we can conclude

$\vdash A \equiv \text{REPL}(A, B :\rightarrow C)$

where $\text{REPL}(A, B :\rightarrow C)$ denotes the result of replacing the expression C for (an occurrence of) the expression B in A.

The "weak equality" operation, denoted by $\equiv$, can be defined using the one-way conditional function (described below in Section 3.2.9) as

$\vdash (p \equiv q) \equiv (\equiv p \land \equiv q \mid p \equiv q)$.

The $\equiv$ connective takes the value error if either or both of its arguments is error, and otherwise agrees with the $\equiv$ connective. It models the usual Boolean equivalence operator denoted in most programming languages by $=$ or EQV. The choice of $\equiv$ to denote the "strong" form of equivalence and $\equiv$ to denote the "weak" form is for compatibility with the usage in PFC3 described in Section 3.3. The words "weak" and "strong" in this context refer respectively to more and less sensitivity to error values as arguments.

3.2.6 Conjunction and Disjunction

The connectives we have chosen as the three-valued analogues to the usual two-valued
The conjunction and disjunction operations are the $\land$ and $\lor$ ("strong and" and "strong or") functions. An examination of their truth tables shows that they agree with their two-valued analogues when confined to arguments which are true or false.

They do not, in their error behavior, model any of the frequently implemented conjunction or disjunction operations in programming languages - _wand, wor, cand and cor do that. But because they seem appropriate for assertion languages and because of their compatibility with the definitions of validity and unsatisfiability (explained below) we have given them greatest prominence in our logic and reserved the characters "$\land$" and "$\lor$" to denote them.

The $\land$ and $\lor$ connectives are commutative and associative, and satisfy a large number of Boolean algebra identities as we shall see. Because of this we will adopt the usual convention that unary operators such as $\neg$ and $\#$, have highest association precedence, $\land$ has higher precedence than $\lor$, and $\#$, $\&$ and $\div$ have lowest precedence. We will drop as many parentheses as seems reasonable while still avoiding ambiguity.

Here are the principle Boolean algebra identities satisfied by the connectives of PC3:

- $\neg \neg p \equiv p$ \hspace{1cm} Double negation
- $\neg(p \land q) \equiv \neg p \lor \neg q$ \hspace{1cm} DeMorgan
- $\neg(p \lor q) \equiv \neg p \land \neg q$ \hspace{1cm} DeMorgan
- $\neg true \equiv false$ \hspace{1cm} Identity
- $\neg false \equiv true$ \hspace{1cm} Identity
- $p \land q \equiv q \land p$ \hspace{1cm} Commutativity
- $p \lor q \equiv q \lor p$ \hspace{1cm} Commutativity
- $p \land (q \land r) \equiv (p \land q) \land r$ \hspace{1cm} Associativity
- $p \lor (q \lor r) \equiv (p \lor q) \lor r$ \hspace{1cm} Associativity
- $p \land (q \lor r) \equiv (p \land q) \lor (p \land r)$ \hspace{1cm} Distributivity
- $p \lor (q \land r) \equiv (p \lor q) \land (p \lor r)$ \hspace{1cm} Distributivity
- $p \land true \equiv p$ \hspace{1cm} Identity
- $p \lor false \equiv p$ \hspace{1cm} Identity
- $p \land false \equiv false$ \hspace{1cm} Absorption
- $p \lor true \equiv true$ \hspace{1cm} Absorption
- $p \land p \equiv p$ \hspace{1cm} Idempotence
3.2.6

\[ \neg p \lor p = p \]  
Idempotence

Two laws of Boolean algebra which fail for the structure \((B, \text{true, false, } \land, \lor, \neg)\) are the laws of complementation.

\[ \neg \neg_3 (p \lor \neg p) = \text{true} \]
\[ \neg \neg_3 (p \land \neg p) = \text{false} \]

In both cases the assignment \(\varphi(p) = \text{error}\) serves as a refutation.

The Law of the Excluded Middle,
\[ \neg \neg_3 p \lor \neg p \]
fails for three-valued logic, because it does not hold under the assignment \(\varphi(p) = \text{error}\). But an analogous law, the Law of the Excluded Fourth, does hold:
\[ \neg \neg_3 T(p) \lor F(p) \lor E(p). \]

Numerous other tautologies are also expressible with the operators we have already presented. For example
\[ \neg \neg_3 \#(p) \land \#T(p) \land \#F(p) \land \#E(p) \land \#(p = q) \]
expresses all in one line the fact that \#(p), T(p), F(p), E(p) and \(p = q\) always take the truth values \text{true} or \text{false}, and never \text{error}.

The operators \{\text{true, false, error, } T, F, E, \#, \neg, \land, \lor\} together form a (more than) complete set of three-valued Boolean operations (as we shall shortly prove) so there is no technical necessity for defining additional operations. We will do so, however, as a convenience.

3.2.7 Wand, wor, cand, cor

The \text{wand} and \text{wor} operations ("weak and" and "weak or") are defined to model the version of the "and" and "or" operations which is most easily and perhaps most frequently implemented in programming languages. The usual implementation for "and" is 1) Evaluate both arguments, and if an error occurs in either, abort; 2) Return the conjunction of the two
arguments as the result. This implementation is captured by \texttt{wand} and \texttt{wor} since if either argument is \texttt{error} the result is \texttt{error} (modelling abortion).

The \texttt{cand} and \texttt{cor} operations model the "conditional and" and "conditional or" operations which appear in some programming languages. The usual implementation for \texttt{cand} is

1) Evaluate the first (left) argument and if an error occurs, abort; 2) If the first argument is \texttt{false}, return \texttt{false} as the result without evaluating the second argument; 3) Otherwise evaluate the second argument and return its value. A similar implementation is used for \texttt{cor}.

Examination of the truth tables for the \texttt{cand} and \texttt{cor} operators shows that they accurately model the \texttt{cand} and \texttt{cor} operators as usually implemented.

Some languages, for instance BLISS/11, define the \texttt{and} operation in such a way that the operands may be evaluated in either order, and if the first one evaluated is \texttt{false}, the second one may or may not be evaluated. Such a semantics makes \texttt{and} a nondeterministic "function".

When a programmer writes the BLISS equivalent of

\begin{verbatim}
i = 0;
if i \neq 0 \land n/i = j
  then
    S_1
  else
    S_2
\end{verbatim}

it is, so far as he is concerned, nondeterministic whether the Boolean expression evaluates to \texttt{false} (because the left conjunct is evaluated first and the evaluation of the right conjunct is skipped) or evaluates to \texttt{error} (because the right conjunct is evaluated first). Such nondeterministic Boolean operations are not represented directly by any Boolean function, although the effect can be expressed indirectly by treating such nondeterministic operations as abbreviations for nondeterministic control structures.

The \texttt{wand}, \texttt{wor}, \texttt{cand} and \texttt{cor} operations can all be defined from the previously described functions by the following identities:
I$\models$ $p$ and $q$ $\equiv$ $(\neg p \land \neg q \land p \land q \lor (E(p) \lor E(q)) \land \text{error})$

I$\models$ $p$ or $q$ $\equiv$ $(\neg p \land \neg q \land (p \lor q) \lor (E(p) \lor E(q)) \land \text{error})$

I$\models$ $p$ cand $q$ $\equiv$ $(T(p) \land q \lor \neg T(p) \land p)$

I$\models$ $p$ cor $q$ $\equiv$ $(F(p) \land q \lor \neg F(p) \land p)$.

In addition, one can easily verify the following tautologies:

I$\models$ $p$ and $q$ $\equiv$ $q$ and $p$ Commutativity

I$\models$ $p$ or $q$ $\equiv$ $q$ or $p$ Commutativity

I$\models$ $p$ and $(q$ or $r)$ $\equiv$ $(p$ and $q)$ or $(p$ and $r)$ Distributivity

I$\models$ $p$ or $(q$ and $r)$ $\equiv$ $(p$ or $q)$ and $(p$ or $r)$ Distributivity

I$\models$ $p$ and $(q$ and $r)$ $\equiv$ $(p$ and $q)$ and $r$ Associativity

I$\models$ $p$ or $(q$ and $r)$ $\equiv$ $(p$ or $q)$ and $(p$ or $r)$ Associativity

I$\models$ $p$ cand $(q$ cor $r)$ $\equiv$ $(p$ cand $q)$ cor $(p$ cand $r)$ Left-distributivity

I$\models$ $p$ cor $(q$ cand $r)$ $\equiv$ $(p$ cor $q)$ cand $(p$ cor $r)$ Left-distributivity

I$\models$ $\neg(p$ and $q)$ $\equiv$ $(\neg p$ and $\neg q)$ DeMorgan

I$\models$ $\neg(p$ or $q)$ $\equiv$ $(\neg p$ or $\neg q)$ DeMorgan

I$\models$ $\neg(p$ cand $q)$ $\equiv$ $(\neg p$ cand $\neg q)$ DeMorgan

I$\models$ $\neg(p$ cor $q)$ $\equiv$ $(\neg p$ cor $\neg q)$ DeMorgan

Note that the commutative and right-distributive laws fail for cand and cor. For example

I$\models$ $(\text{true cor error})$ cand false $\equiv$ $(\text{true cand false})$ cor $(\text{error cand false})$

3.2.8 Material Implication

The $\triangleright$-connective we have defined for PC3 is a close relative of the $\triangleright$-connective of PC2. (Let us temporarily denote them by $\triangleright_3$ and $\triangleright_2$.) It has three important properties which qualify it to be viewed as an appropriate three-valued analogue of two valued implication.

1. $\triangleright_3$ agrees with $\triangleright_2$ when its arguments are confined to true and false.

2. $\triangleright_3$ is "transitive" in a three-valued sense. If $p \triangleright_3 q$ is true and $q \triangleright_3 r$ is true then $p \triangleright_3 r$ is true.

3. The rule of modus ponens is sound for $\triangleright_3$. If $F_3p$ and $F_3p \triangleright_3 q$ then $F_3q$.

These criteria do not, however, even come close to determining a unique three-valued
truth function. If we further require that

1. \( \supset_3 \) be two-valued (i.e. that it never take the value error), and

2. \( (p \supset_3 q) = \text{true} \) whenever \( \not p \) in PFC3 (see Section 3.3)

then the truth function we have chosen is determined. It can be defined naturally enough by the identity

\[ \equiv (p \supset q) = (\neg T(p) \lor T(q)) \]

Thus, \( p \supset q \) is true whenever \( p \) is not true (i.e. is false or error) or whenever \( q \) is true (or both).

Although we have characterized \( \supset_3 \) as the natural analogue to \( \supset_2 \), the reader should be cautioned that one important property of \( \supset_2 \) is not possessed by \( \supset_3 \). The identity

\[ \equiv (p = q) = (p \supset q) \land (q \supset p) \]

is not valid. It fails for the assignment \( \varphi(p) = \text{false}, \varphi(q) = \text{error} \). Thus, equivalence is not the same as mutual implication. This does not indicate a failure in our selection of the truth function denoted by \( \supset_3 \), nor that for \( \land \) or \( \equiv \). None of the reasonable candidates have this property.

### 3.2.9 Conditionals

Three extremely convenient operators in three-valued logic are the one-way, two-way and three-way conditional functions. They can be defined as follows.

\[ \equiv (p \mid q_1 \mid q_2 \mid q_3) = (T(p) \land q_1 \lor F(p) \land q_2 \lor E(p) \land q_3) \]

\[ \equiv (p \mid q_1 \mid q_2) = (p \mid q_1 \mid q_2 \mid \text{error}) \]

\[ \equiv (p \mid q_1) = (p \mid q_1 \mid \text{error} \mid \text{error}) \]

The three-way conditional \( (p \mid q_1 \mid q_2 \mid q_3) \) can be viewed as a case-expression on the truth value of \( p \). If \( p \) is true, the expression's value is the value of \( q_1 \); if \( p \) is false the expression's value is that of \( q_2 \); and if \( p \) is error the expression's value is that of \( q_3 \). The two-way and
one-way conditionals are frequently useful variations in which the third alternative, or both
the second and third alternatives, is always error.

Part of the importance of the conditional operations is that they properly model the
behavior of programming language conditionals such as if-then-else. For example, consider
the Boolean conditional expression of Algol-60, if \( p \) then \( q \) else \( r \). If evaluation of \( p \)
completes normally and returns true, then the value of the expression is the value of \( q \); but
the program aborts if \( q \) aborts. This is completely independent of what happens with \( r \), and
sensible implementations therefore do not evaluate \( r \). On the other hand, if \( p \) is false, then the
expression's value is similarly determined by \( r \), independent of the value of \( q \), and sensible
implementations thus do not evaluate \( q \). Finally, there is, of course, the third alternative:
evaluation of \( p \) might err, in which case the entire program aborts. This is a fairly complex
behavior, requiring several sentences to describe, but it is precisely modelled by the
two-way conditional expression \( (p|q|r) \), or the three-way conditional \( (p|q|r|\text{error}) \).
We need only interpret true and false in the usual manner, and interpret error as abortion.
(We will not be concerned with termination issues in this thesis.) The nearest approximation
in two-valued logic to these conditional functions of three-valued logic is the if-then-else
truth function, defined for example in [Manna 74].

\[
\text{if } p \text{ then } q \text{ else } r \equiv (p \Rightarrow q) \land (\neg p \Rightarrow r)
\]

The \( \neg \), \( \land \) and \( \Rightarrow \) connectives in that definition are, of course, those of two-valued propositional
logic. This if-then-else function simply fails to capture the semantics properly when \( p \), \( q \)
and/or \( r \) contain occurrences of partial functions or predicates.

Besides these conditional functions of type Boolean, it will prove valuable to define
conditionals for other types, and we shall do so in Section 3.3. In each case they will
accurately model the error behavior of the corresponding conditional expression construct
which is familiar from programming languages.
3.2.10 Complete sets of three-valued propositional connectives

For two-valued propositional logic it is well known that the sets \{\neg, \to\} and \{\neg, \land, \lor\} are each complete sets of connectives in the sense that any two-valued truth function of any finite arity can be represented by a propositional expression involving only the \neg and \to connectives, or only the \neg, \land and \lor connectives. For PC3 to be satisfactory we must be sure that we have a complete set of three-valued connectives.

**Definition 3:** A set of three-valued propositional connectives is complete iff every three-valued Boolean function \(f: \mathcal{B}^n \to \mathcal{B}\) can be represented by a propositional wff \(w\) with free propositional variables \(p_1, \ldots, p_n\) such that \(f = (\lambda p_1 \ldots \lambda p_n w)\).

Probably the simplest, most straightforward complete set of functions is \(\{\text{true}, \text{false}, \text{error}, \langle\circ | \circ | \circ | \circ\}\}\). We prove this now.

**Theorem 4:** \(\{\text{true}, \text{false}, \text{error}, \langle\circ | \circ | \circ | \circ\}\}\) is a complete set of three-valued truth functions.

**Proof:** By induction on the arity of the function to be represented.

**Base:** The three 0-ary functions are representable by the wffs \text{true}, \text{false} and \text{error}, each of which has 0 free propositional variables.

**Induction:** Suppose all \(n\)-ary Boolean functions are representable. Let \(f\) be an \((n+1)\)-ary Boolean function. Define

\[
\begin{align*}
  f_T &= \lambda p_1 \ldots \lambda p_n f(p_1, \ldots, p_n, \text{true}) \\
  f_F &= \lambda p_1 \ldots \lambda p_n f(p_1, \ldots, p_n, \text{false}) \\
  f_E &= \lambda p_1 \ldots \lambda p_n f(p_1, \ldots, p_n, \text{error}).
\end{align*}
\]

Let \(f_T, f_F\) and \(f_E\) be representable by wffs \(w_T, w_F\) and \(w_E\) respectively (which is possible by the induction hypothesis). Then \(f\) can be represented as follows:

\[
f = \lambda p_1 \ldots \lambda p_{n+1} (p_{n+1} | w_T | w_F | w_E)
\]

It is trivial to verify that \((p_{n+1} | w_T | w_F | w_F)\) has \(n+1\) free variables and that it contains only the Boolean functions \text{true}, \text{false}, \text{error} and \(\langle\circ | \circ | \circ | \circ\}\). \(\square\)

We showed earlier that the three-way conditional was definable from other Boolean functions by the identity

\[
E(p | q_1 | q_2 | q_3) = (T(p) \land q_2) \lor (F(p) \land q_2) \lor (E(p) \land q_3).
\]
If we note that true and false are definable by the two identities

\[
\begin{align*}
\text{true} & \equiv T(p) \lor F(p) \lor E(p) \\
\text{false} & \equiv E(E(p))
\end{align*}
\]

we are lead immediately to the following theorem:

**Theorem 5**: \( \{\text{error, T, F, E, }\land, \lor\} \) is a complete set of Boolean functions.  
**Proof**: Trivial. \( \square \)

Also, we note these identities

\[
\begin{align*}
\text{true} & = \#p \land p \\
\text{false} & = \#p \land \neg p \\
\text{E} & = \neg \#p \\
\text{p} \lor q & = \neg(\neg p \land \neg q)
\end{align*}
\]

which lead directly to

**Theorem 6**: \( \{\text{error, }\#, \neg, \land\} \) is a complete set of Boolean functions.  
**Proof**: Trivial. \( \square \)

### 3.3 Semantics of PC3

The semantics of PC3 follows closely that of the usual two-valued propositional calculus, PC2. Let us denote by \( B \) the set of truth values \( \{\text{true, false, error}\} \), and by \( P \) the set of all propositional variables. We will define truth value assignment, valuation, satisfiability and validity as follows.

**Definition 7**: A truth value assignment is a map \( \varphi: P \to B \). \( \square \)

**Definition 8**: The valuation \( V_\varphi \) induced by a truth value assignment \( \varphi \) is a map \( V_\varphi: \text{LPC3} \to B \) such that

1. \( V_\varphi(p) = \varphi(p) \), for all propositional variables \( p \in P \).
2. \( V_\varphi(op(\beta_1, \ldots, \beta_k)) = OP(V_\varphi(\beta_1), \ldots, V_\varphi(\beta_k)) \) where \( op \) is one of the 20 propositional connectives, \( \beta_1, \ldots, \beta_k \) are the appropriate number of arguments for the arity of \( op \), and \( OP \) is the function defined by the truth table for \( op \).

**Definition 9:** A wff \( w \in \text{LPC3} \) is satisfiable iff there is an assignment \( \varphi \) such that \( V_\varphi(w) = \text{true} \). We write "\( \models w \)".

**Definition 10:** A wff \( w \in \text{LPC3} \) is valid iff for all assignments \( \varphi, V_\varphi(w) = \text{true} \). We write "\( \vdash w \)".

These definitions are similar to the usual definitions of satisfiability and validity in PC2, but there are several points worth noting. First, for a wff to be satisfiable there must be an assignment which makes it \text{true}; it is not sufficient for there to be an assignment making the wff non-\text{false}. The distinction between \text{true} and non-\text{false} is at the heart of three-valued logic. A similar remark applies to the definition of validity.

Third, we should notice that although there are three truth values in the semantics of the object language, in the informal semantics of the meta-language there are only two truth values. A wff is either valid or it is invalid; it is satisfiable or it is unsatisfiable. In each case there is no third choice.

We have introduced the symbol "\( \models \)" to mean "is satisfiable". Apparently there is no standard logical notation for this notion, so we defined our own and will make frequent use of it. Occasionally we will have to distinguish between validity (satisfiability) in two-valued logic and validity (satisfiability) in three-valued logic. At such times we will use a subscript to distinguish the two, i.e. \( \vdash_2 w \) for validity in two-valued logic, \( \vdash_3 w \) for validity in three-valued logic, \( \models_2 \) for satisfiability in two valued logic and \( \models_3 \) for satisfiability in three-valued logic. We will also feel free to write \( \not\models_2 \) and \( \not\models_3 \) to indicate unsatisfiability, and \( \not\vdash_2, \not\vdash_3 \) for nonvalidity.
3.3.1 The relationship between Validity and Satisfiability

In two-valued logic the relationship between validity and satisfiability is quite simple:

\[ \log F_2 w \iff \log \neg w. \]

A correspondingly simple relationship holds in three valued logic:

**Theorem 11:** For all PC3 wffs \( w \),

\[ \log F_3 w \iff \log \neg \top(w). \]

**Proof**: If \( w \) is true under all assignments, then \( T(w) \) is also true under all assignments; hence \( \neg T(w) \) is always false and \( \neg \top(w) \).

**Proof**: If \( \neg T(w) \) is never true under any assignment, then it must be false under all assignments, error not being in the range of the \( T \)-function. But if \( \neg T(w) \) is false under all assignments, then \( T(w) \) is true under all assignments, and so is \( w \). Hence \( \log Fw \).

We will repeat this theorem in Section 3.4 because it holds for the partial functional calculus PFC3 as well, once \( \log F \) and \( \log \neg \) have been suitably defined, and the proof works without change.

3.3.2 Conjunction, Disjunction, Validity, Satisfiability

In two-valued logic conjunction is closely related to validity, and disjunction to satisfiability, in the following way.

\[ \log F_2 p \land q \iff \log F_2 p \land \log F_2 q \]
\[ \log \neg_2 p \lor q \iff \log \neg_2 p \lor \log \neg_2 q \]

Similar relationships hold in three-valued logic.

\[ \log F_3 p \land q \iff \log F_3 p \land \log F_3 q \]
\[ \log \neg_3 p \lor q \iff \log \neg_3 p \lor \log \neg_3 q \]

\[ \log \neg_3 p \lor q \iff \log \neg_3 p \lor \log \neg_3 q \]

However, we should notice that the following two relations do not hold.

\[ \log \neg_3 p \lor q \iff \log \neg_3 p \lor \log \neg_3 q \]
\[ \log \neg_3 p \lor q \iff \log \neg_3 p \lor \log \neg_3 q \]
In both cases the properties fail in the $\Leftarrow$ direction (although they hold in the $\Rightarrow$ direction.)

The important relationship between disjunction and satisfiability will be most often used in the negative form as

$$\overline{\overline{A_3p \lor q} \iff \overline{A_3p} \land \overline{A_3q}}.$$ 

The fact that this relation holds for $\lor$, but not for $\text{lor}$ or $\text{cor}$ is the reason that we have given greatest prominence in our logic to the $\land/\lor$ pair instead of the $\text{and}/\text{lor}$ pair or the $\text{and}/\text{cor}$ pair.

3.3.3 Normal Forms

In two-valued propositional calculus it is well-known that every wff $w$ can be put into conjunctive- or disjunctive-normal-form (CNF or DNF). Such normal forms also exist for wffs of the three-valued propositional calculus, and will play a significant role in the algorithms of Chapters 6 and 7.

Definition 12: A wff $w \in LP(3)$ is in (three-valued) disjunctive normal form iff, when $w$ is expressed as an operator-operand tree, any string of nodes on a path from the root node to a leaf node is a member of the language $v^* \land^* (\text{error} \lor p \lor \neg p \lor \#p \lor \neg\#p)$ where $\ast$, $\lor$ and juxtaposition represent the Kleene regular expression operators for repetition, alternation and concatenation respectively, and "p" represents any propositional variable.

If we consider that $\land$ and $\lor$ are both associative, then this definition simply requires that $w$ be of the form

$$V_{i \leq m} \land_{j \leq n_i} (w_{ij})$$

where each $w_{ij}$ is error or, for some variable $p$, of the form $p$, $\neg p$, $\#p$ or $\neg\#p$.

Theorem 13: For any wff $w$ we can find (effectively) a wff $w'$ such that $\mathcal{L}_3 w \equiv w'$ and $w'$ is in DNF.

Proof: In the previous subsection we showed that $\{\text{error}, \neg, \#\}$ is an adequate set of connectives. Clearly then, any wff $w$ is equivalent to some wff $w_1$ such that the only function symbols in $w_1$ are $\text{error}$, $\neg$, $\#$, $\land$ and $\lor$. In fact, although we don't prove it here, such a $w_1$ can be constructed (effectively) from $w$ merely by substituting, for all occurrences of the other 15 operators, equivalent expressions involving only $\text{error}$, $\neg$, $\#$,
\[ \land \text{ and } \lor. \]

From \( w_1 \) the required wff \( w' \) can be constructed using the following tree-replacement system.

\[
\text{do} \]
\[\quad \text{a) } \]
\[
\begin{align*}
\neg \text{true} & \rightarrow \text{false} \\
\neg \text{false} & \rightarrow \text{true} \\
\neg \text{error} & \rightarrow \text{error} \\
\#\text{true} & \rightarrow \text{true} \\
\#\text{false} & \rightarrow \text{false} \\
\#\text{error} & \rightarrow \text{false} \\
p \land \text{true} & \rightarrow p \\
\text{true} \land p & \rightarrow p \\
p \land \text{false} & \rightarrow \text{false} \\
\text{false} \land p & \rightarrow \text{false} \\
p \lor \text{true} & \rightarrow \text{true} \\
\text{true} \lor p & \rightarrow \text{true} \\
p \lor \text{false} & \rightarrow p \\
\text{false} \lor p & \rightarrow p \\
\#\#p & \rightarrow \text{true} \\
\#\neg p & \rightarrow \#p \\
\neg\neg p & \rightarrow p
\end{align*}
\]
\[\text{b) }\]
\[
\begin{align*}
\#(p \lor q) & \rightarrow (#p \land p) \lor (#q \land q) \lor (#p \land #q \land \neg p \land \neg q) \\
\#(p \land q) & \rightarrow (#p \land \neg p) \lor (#q \land \neg q) \lor (#p \land #q \land p \land q)
\end{align*}
\]
\[\text{c) }\]
\[
\begin{align*}
\neg(p \lor q) & \rightarrow \neg p \land \neg q \\
\neg(p \land q) & \rightarrow \neg p \lor \neg q
\end{align*}
\]
\[\text{d) }\]
\[
\begin{align*}
p \land (q \lor r) & \rightarrow (p \land q) \lor (p \land r) \\
(q \lor r) \land p & \rightarrow (q \land p) \lor (r \land p)
\end{align*}
\]
\[\text{)do}\]

We will omit the detailed correctness proof for this algorithm since it is not especially enlightening or difficult. Both the weak correctness proof and the termination proof are similar to those for the two-valued DNF algorithm of Chapter 2. \( \square \)

Like the DNF algorithm of Chapter 2, this algorithm would be correct even if all of the productions were at the top level and unordered. But, even though the correctness result
for such an algorithm would be stronger, the algorithm itself would be inferior to this one in execution time and length of output.

3.4 The Three-valued Partial Functional Calculus: PFC3

In this section we extend the three-valued propositional logic PC3 of the last section to a three-valued, multisorted, first-order, partial functional calculus called PFC3. We will describe the language and its semantics in detail, but we will not construct any axiomatization since it would not be used in the rest of the thesis.

The logic we will develop is a partial-functional logic, since it has individual variables, symbols for partial-functions (and -predicates), and quantifiers similar to those of the usual functional (predicate) calculus. It is first-order since only individual variables (as opposed to partial-function variables) are permitted to be bound by quantifiers. It is multisorted because each term is considered to have a type (or logical sort) associated with it, and terms are required to be constructed in such a way that the arguments to functions (and predicates) are of the correct types. And, finally, the logic is three-valued because the central type, Boolean, contains three truth values.

3.4.1 Types

We will use the word "type" to refer to what logicians mean by "logical sort". Semantically a type refers to a domain (set) of values. The usual predicate calculus is most frequently viewed as having one type, by which is meant that all of the values of terms are drawn from a single set known as the "universe of discourse". However, in the view we espouse here -- a view consistent with the usual practice in programming languages -- the predicate calculus is viewed as having two types: type Boolean, the domain of truth values, and an unnamed type which is the domain of interpretation for individual variables and terms.

A generalization of this view leads to the well-known notion of a multisorted logic. There is a (possibly infinite) set of domains (sets) each of which is denoted by a type symbol: S, T,
T₁, T₂ ... X, Y, Z, etc. One of the domains, denoted by B, is the set of Booleans, either {true, false} in two-valued logic or {true, false, error} in three-valued logic.

Functions from any finite cross-product of domains to any single domain are considered meaningful, but other functions (such as f: X u Y → Z) are excluded from consideration (unless of course some domain T is equal to X u Y). Functions taking values of type B are called relations.

The multisorted logic described here is intended to model accurately the type system of programming languages, wherein each expression has a unique type and Boolean is just another type. The correspondence will be clearer in the next subsections.

3.4.2 Generic functions

Every type comes equipped with seven associated function symbols. Because they are defined for all types (including B) and play similar roles for each type, we will call them generic functions symbols. They have the same status that = has in most treatments of the usual predicate calculus and can thus be viewed as logical symbols.

For a type X the seven functions are as follows:

\( E_X : \rightarrow X \)  
The error constant for Type-X

\( \#_X : X \rightarrow B \)  
The definedness function: its value is true if its argument is not \( E_X \); false otherwise.

\( =_X : X \times X \rightarrow B \)  
The identity function: its value is true if its arguments are the same element (including error elements); false otherwise.

\( \neq_X : X \times X \rightarrow B \)  
The weak equality function: its value is error if either of its arguments is \( E_X \); otherwise true if its arguments are the same elements and false if they are not.

\( (\circ \mid \circ)_X : B \times X \rightarrow X \)  
The one-way conditional function. The value of \( (b \mid x)_X \) is x if b is true; otherwise it is \( E_X \).

\( (\circ \mid \circ \mid \circ)_X : B \times X \times X \rightarrow X \)  
The two-way conditional function. The value of \( (b \mid x_1 \mid x_2)_X \) is \( x_1 \) if b is true; it is \( x_2 \) if b is false; otherwise it is \( E_X \).
\( \langle \circ | \circ | \circ | \circ \rangle_{X} : B \times X \times X \times X \rightarrow X \)

The three-way conditional function. The value of \( \langle b | x_{1} | x_{2} | x_{3} \rangle_{X} \) is \( x_{1} \) if \( b \) is true; \( x_{2} \) if \( b \) is false; \( x_{3} \) if \( b \) is error.

We will usually drop the type subscripts, allowing the types in an expression to be inferred from the types of the arguments.

When \( X \) is Type B, the generic functions are truth functions, and are the same as the propositional functions error \((E_{B})\), \#, \&, \texttt{=} \((\circ, \circ)\), \((\circ | \circ | \circ)\), and \((\circ | \circ | \circ | \circ)\) that were discussed in Section 3.2.

3.4.3 Syntax for PFC3: Three-valued, multisorted first order languages

Following the usual presentation of logic, we must begin by defining the syntax of a first order language, augmented with multiple sorts and three truth-values.

**Definition 14:** A three-valued, multisorted first order partial functional language PFC3 is defined as follows:

1. There is a nonempty and possibly infinite set of **type symbols**. (We will use capital letters for type symbols). One of the type symbols must be B (for Boolean).

2. For every type symbol \( T \) there is an infinite sequence of individual variable symbols of type \( T \). Individual variables will be denoted by (possibly subscripted) lower case letters. The type of a variable will be encoded typographically by the convention that \( t, t_{1}, t_{2}, \ldots \) are always of type \( T \), and \( x, x_{1}, x_{2}, \ldots \) are always of type \( X \), etc. The variables \( b, b_{1}, b_{2}, \ldots \) are called propositional variables.

3. For all finite non-empty sequences of type symbols \( X_{1}, \ldots , X_{n} \), there is a countable supply of nonlogical (uninterpreted) function symbols \( f_{1}^{1}, f_{2}^{2}, f_{3}^{3}, \ldots , f_{n}^{n} \), \( g_{1}^{1}, \ldots \) each representing a (partial) function from \( X_{1} \times \cdots \times X_{n} \) to \( X \). Zero-ary function symbols (for which \( n = 0 \)) are called constant symbols (of Type-X). Function symbols for which \( X \) is \( B \) are called predicate symbols. For a symbol \( f_{X_{1}}^{1} \cdots X_{n} \) \( X_{X} \) we will (loosely) say that \( X_{1} \cdots X_{n} X \) is its type.

4. In addition to the nonlogical function symbols there are, for each type \( X \), the following generic function symbols: \( E_{X}, \#_{X}, \&_{X}, =_{X} \), \((\circ | \circ)_{BXX}, \((\circ | \circ | \circ)_{BXXX}, \text{and } (\circ | \circ | \circ | \circ)_{DXXX} \). They are, respectively, the error constant symbol for \( X \), the definedness predicate for \( X \), the weak and strong equality predicates for \( X \) and the one-way, two-way and three-way
conditional functions for type X. When \( X = B \), these symbols are identified, respectively, with the error, \( \#, =, \# \), \( \langle \circ \mid \circ \rangle \), \( \langle \circ \mid \circ \mid \circ \rangle \) and \( \langle \circ \mid \circ \mid \circ \rangle \) connectives for PC3 of Section 3.2.9.

5. For type B only, there are also the following logical function symbols: \( \land_{BBB} \), \( \lor_{BBB} \), \( \land_{BBB} \), \( \lor_{BBB} \), \( \land_{BBB} \), \( \lor_{BBB} \), \( \top_{BB} \), \( \bot_{BB} \), \( \text{true}_B \), \( \text{false}_B \). These are identified with the corresponding propositional connectives of PC3. Note: we will consider "\( E_B \)" and "error\( _B \)" to be the same symbol.

6. The quantifier symbols \( \forall \) and \( \exists \) are symbols in the language.

7. Finally, the parentheses \( "(\)" and \( "\)" and \( ",.\)" and their typographical variants are symbols in the language.

From these symbols we construct wffs inductively as follows:

**Definition 15:**

1. A variable \( x \) of type \( X \) is an expression of type \( X \).

2. If \( f_{X_1} \ldots X_n^X \) is a function symbol of type \( X_1 \ldots X_n \) and \( \tau_1 \ldots \tau_n \) are expressions of types \( X_1 \), \ldots, \( X_n \) respectively, then \( f_{X_1} \ldots X_n^X(\tau_1 \ldots \tau_n) \) is an expression of type \( X \).

3. If \( \beta \) is an expression of type B (Boolean) and \( x \) is a variable of type \( X \), then \( (\forall x.\beta) \) and \( (\exists x.\beta) \) are expressions of type B.

4. A wff is an expression of type B.

By a term (of type \( X \)) we will mean an expression (of type \( X \)) which contains no occurrences of quantifiers. All of the usual conventions involving free and bound variables and their scopes apply in this logic. Notice that what is usually referred to as an "atomic formula" is here called a "term of type B". We will follow the convention that type symbol subscripts will be dropped except where necessary for emphasis or to avoid ambiguity. In most cases only some constant symbols will have to be embellished with explicit type subscripts.

3.4.4 Semantics of the logic

The semantics for this logic is a variation on the Tarskian semantics for the usual first-order predicate calculus. The major differences (aside from the handling of multiple
types) are the treatment of error values, partial functions and assignments to free and bound variables.

For the language $L_{PFC3}$ let $T_L$ be the set of type symbols of $L_{PFC3}$ and $F_L$ be the set of logical, generic and nonlogical function symbols of $L_{PFC3}$. We first define an interpretation.

Definition 16: An interpretation for a first-order language $L$ for PFC3 contains the following three components:

1. a family $U$ of nonempty sets, one of whose members is $\text{Boolean} = \{\text{true, false, error}\}$,

2. a family $V$ of functions each of which maps some finite cross-product of members of $U$ into another member of $U$, and

3. a mapping $I$ such that $I: T_L \to U$ and $I: F_L \to V$ according to the following restrictions:
   a. $I(B) = \text{Boolean}$ (i.e. the interpretation of the type symbol $B$ is required to be the set of truth values)
   
   b. If $f_{x_1 \ldots x_n} \in F_L$ then $I(f_{x_1 \ldots x_n}): I(x_1) \times \cdots \times I(x_n) \to I(X)$ (i.e. the interpretation respects the type signatures of function symbols).
   
   c. For the Boolean function symbols we require that $I(\text{true}_B) = \text{true}$, $I(\text{false}_B) = \text{false}$, $I(\text{error}_B) = \text{error}$, $I(T_B) = T$, $I(F_B) = F$, $I(E_B) = E$, $I(\neg B) = \neg$, $I(\land B) = \land$, $I(\lor B) = \lor$, $I(\Rightarrow B) = \Rightarrow$, $I(\wedge B) = \wedge$, $I(\vee B) = \vee$, $I(\land B) = \land$, $I(\lor B) = \lor$, $I(\Rightarrow B) = \Rightarrow$, $I(\wedge B) = \wedge$, $I(\vee B) = \vee$, i.e. that the Boolean function symbols be interpreted as they were in the propositional logic PC3.
   
   d. For all types $X$, the generic functions are interpreted as follows:
      
      i. $I(\text{E}_X)(t)$ may be any element in $I(X)$ (except for the requirement above that $I(E_B) = \text{error}$). Let us denote $I(\text{E}_X)$ by $x_X$.
      
      ii. $I(\text{E}_{X}) (t)$ is $\text{true}$ if $t$ is not $x_X$, and $\text{false}$ if $t$ is $x_X$.
      
      iii. $I(\text{E}_{X})(t_1, t_2)$ is $\text{true}$ if $t_1$ and $t_2$ are the same elements of $I(X)$, and $\text{false}$ otherwise.
      
      iv. $I(\text{E}_{X})(t_1, t_2)$ is $\text{error}$ if $t_1$ is $x_X$ or $t_2$ is $x_X$; otherwise it is $\text{true}$ if $t_1$ and $t_2$ are the same element of $I(X)$ and $\text{false}$ if they are not.
      
      v. $I(\text{E}_{X})(t_1, t_2)$ is $\text{true}$ if $t_2$ is $x_X$ or $t_3$ is $x_X$; otherwise it is $\text{true}$ if $t_1$ and $t_2$ are the same element of $I(X)$ and $\text{false}$ if they are not.
      
      vi. $I(\text{E}_{X})(t_1, t_2)$ is $\text{true}$ if $t_1$ is $x_X$ or $t_2$ is $x_X$; otherwise it is $\text{true}$ if $t_2$ is $x_X$ or $t_3$ is $x_X$; otherwise it is $\text{false}$ if $t_1$ and $t_2$ are the same element of $I(X)$ and $\text{false}$ if they are not.
vii. \( I((\ast \ast)_{BXX})(b, t_1) \) is \( t_1 \) if \( b \) is \text{true}, and \( e_X \) if \( b \) is \text{false} or \text{error}. \( \Box \)

This rather long-winded definition is actually a straightforward variation of the usual notion of an interpretation for a multisorted first-order language. We are only adding the requirements that error values and the generic functions be properly interpreted for each type.

The clauses of this definition which apply to the generic and Boolean operators may seem puzzling. In two-valued logic the only "generic" operator usually considered is equality (denoted by \( = \)), and its meaning is defined either semantically (through the notion of a normal interpretation) or syntactically (using an infinite first-order axiomatization for equality). Also, in most treatments the propositional connectives are not viewed as in the domain of the interpretation mapping \( I \). But with as many logical and generic operators as we have here, it seemed natural to include them in the definition of an interpretation. This decision simplifies the definition of a valuation later on.

Let us denote by \( V_L \) the set of all individual variables in language \( L \), and for a particular type \( T \) let \( V_{L,T} \) denote the set of individual variables of type \( T \) in language \( L \). For an interpretation \( I \), let \( U_I \) be the family of sets used as the interpretations of the type symbols of \( L \).

**Definition 17:** An assignment for language \( L \) with interpretation \( I \) is a mapping \( \varphi: V_L \to \bigcup_{S \in T}(s) \) such that if \( x \in V_{L,X} \) then \( \varphi(x) \in I(X) \).

In other words, an assignment maps variable symbols to elements of the appropriate types.

**Definition 18:** Definition:
An ordinary assignment for language \( L \) with interpretation \( I \) is an assignment \( \varphi \) such that for all variables \( x \), \( \varphi(x) \neq e_X \).
Thus, an assignment is ordinary if no variable is assigned an error value. We will use only ordinary assignments in this thesis because our logic is designed for program verification, and the usual semantics for programming languages does not permit variables to hold error values; even when errors occur in a computation, error values are not assignable to variables.

An interpretation and an ordinary assignment for a language $L$ together induce a valuation on the wffs of $L$ in the usual manner.

**Definition 19:** For language $L$ with interpretation $I$ and ordinary assignment $\varphi$ the induced valuation is the function $V_{I,\varphi}: L \rightarrow \mathcal{U} \in \mathcal{U}(S)$ such that

1. For a variable $t$, $V_{I,\varphi}(t) = \varphi(t)$
2. For a wff of the form $f_{T_1, \ldots, T_n}(\alpha_1, \ldots, \alpha_n)$ where $f$ is any function symbol,
   
   $$V_{I,\varphi}(f_{T_1 \ldots T_n}(\alpha_1, \ldots, \alpha_n)) =$$
   
   $\mathbb{I}(f_{T_1 \ldots T_n})(V_{I,\varphi}(\alpha_1), \ldots, V_{I,\varphi}(\alpha_n))$

3. For a wff of the form $(\forall t. \psi)$:
   a. $V_{I,\varphi}((\forall t. \psi))$ is true if for all ordinary assignments $\psi$ such that $\psi(x) \equiv \varphi(x)$ for all $x$ different from $t$, $V_{I,\psi}(\alpha) \equiv \text{true}$;
   b. $V_{I,\varphi}((\forall t. \psi))$ is false if there exists an ordinary assignment $\psi$ such that $\varphi(x) \equiv \psi(x)$ for all $x$ different from $t$, $V_{I,\psi}(\alpha) \equiv \text{false}$;
   c. $V_{I,\varphi}((\forall t. \psi))$ is error otherwise.

4. For a wff of the form $(\exists t. \psi)$:
   a. $V_{I,\varphi}((\exists t. \psi))$ is true if there exists an ordinary assignment $\psi$ such that $\varphi(x) \equiv \psi(x)$ for all $x$ different from $t$, $V_{I,\psi}(\alpha) \equiv \text{true}$;
   b. $V_{I,\varphi}((\exists t. \psi))$ is false if for all ordinary assignments $\psi$ such that $\psi(x) \equiv \varphi(x)$ for all $x$ different from $t$, $V_{I,\psi}(\alpha) \equiv \text{false}$;
   c. $V_{I,\varphi}((\exists t. \psi))$ is error otherwise.

**Definition 20:** A wff $w$ in language $L$ is universally valid if, for all interpretations $I$ and ordinary assignments $\varphi$, $V_{I,\varphi}(w) \equiv \text{true}$. We write $\mathbb{I}w$. 
Definition 21: A wff \( w \) in language \( L \) is \textbf{satisfiable} if, for some interpretation \( I \) and ordinary assignment \( \varphi \), \( V_I,\varphi(w) = \text{true} \). We write \( \models w \).

These definitions of valuation, validity and satisfiability agree with those for the PC3 of Section 3.2, and parallel the definitions usually given in most texts for the usual predicate calculus. We state without proof that the validity problem for PFC3 is undecidable, but that a complete axiomatization exists (and can be constructed by modifying any of the well-known complete axiomatizations of the two-valued predicate calculus.)

One point about the semantics should be made explicit. In the definitions of valuation, validity and satisfiability we confined attention to ordinary assignments, i.e. assignments in which no variable receives an error value. The effect of this is to consider bound variables as ranging over the non-error values of the appropriate type; they do not range over all values. We will adopt this as a general convention.

Convention: All formal bound variables used in this thesis will be understood to range over the non-error values of the appropriate type. This applies not only to variables bound by \( \forall \) and \( \exists \), but also to variables bound by such other operators as \( \lambda \), \( \sum \), \( \mu \) and the "set of" operator, e.g. \( \{ x \mid P(x) \} \).

3.4.5 Some valid wffs and sound inference rules

To get a flavor for PFC3 we should examine some valid wffs and inference rules and compare them to the corresponding wffs and rules of FC2. In Section 3.2 we compared the propositional parts of the two logics, so here we will concentrate on the term-and-quantifier parts.

A number of the rules for manipulating quantifiers in FC2 are valid also in PFC3, for example, the following schemata:

\[ F_{2,3} \forall x.(\alpha \land \beta) = (\forall x.\alpha \land \forall x.\beta) \]
\[ F_{2,3} \forall x \forall y.\alpha \equiv \forall y \forall x.\alpha \]
\[ F_{2,3} (\forall x.\alpha) \lor \beta \equiv \forall x.(\alpha \lor \beta), \text{if } x \text{ does not occur free in } \beta. \]

Furthermore, as in two-valued logic, the \( \exists \)-quantifier is the dual (w.r.t. \( \lnot \)) of the \( \forall \)-quantifier,
\[ F_{2,3} \forall x. \alpha = \exists x. - \alpha \]

and so we can derive the duals of the earlier formulae.

\[ F_{2,3} \exists x. (\alpha \lor \beta) = (\exists x. \alpha \lor \exists x. \beta) \]
\[ F_{2,3} \exists x \exists y. \alpha = \exists y \exists x. \alpha \]
\[ F_{2,3} (\exists x. \alpha) \land \beta = \exists x. (\alpha \land \beta), \text{ if } x \text{ does not occur free in } \beta. \]

We have dropped type symbols from these formulae, first because there is no ambiguity, second because the equivalences hold for variables and quantifiers of all types, and third, because we wish to emphasize the similarity of form between the valid wffs of PFC3 and those of PC2.

There are several important truths about quantifiers in FC2 which do not hold without modification in PFC3. For example, in two valued logic the following schema holds

\[ F_{3} \forall x. \alpha \supset \alpha(\tau/x) \text{ for any term } \tau. \]

However, the corresponding schema in PFC3 is not valid. The reason is that we have defined the bound variable \( x \) to range over the non-error values of type \( X \), which leaves open the possibility that

\[ \forall x. \alpha \supset \alpha (E_x/x) \]

is not true under some assignments. We can, however, assert the slightly weaker schema

\[ F_{3} \forall x. \alpha \supset \alpha (\# \tau \supset \alpha (\tau/x)) \text{ for all terms } \tau \text{ of type } X. \]

An alternative variant is this one:

\[ F_{3} \forall x. \alpha \supset \alpha (x_1/x) \text{ where } x_1 \text{ is a variable of type } X. \]

The reason this schema is valid is that the free variable \( x_1 \) also ranges over only the non-error values of Type \( X \) because of our restriction to ordinary assignments, just as does the bound variable \( x \).

The existential duals to these schemata have similar properties. The following schema are valid in two-valued logic, but not valid in three-valued logic.

\[ F_{3} \alpha(\tau/x) \supset \exists x. \alpha \text{ where } \tau \text{ is any } x \text{-term.} \]
But these variants are valid:

\[ F_3 \alpha(\tau/x) \Rightarrow (\#\tau \Rightarrow \exists x.\alpha) \] where \( \tau \) is any \( X \)-term

\[ F_3 \alpha(x_1/x) \Rightarrow \exists x.\alpha \] where \( x_1 \) is an \( X \)-variable

Similar remarks hold for the normal rules of inference involving quantifiers. The usual Rule of Generalization for two-valued logic

\[ F_2 \alpha \Rightarrow F_2 \forall x.\alpha, \]

does hold for three-valued logic,

\[ F_3 \alpha \Rightarrow F_3 \forall x.\alpha. \]

(Since we are not developing any proof system we will continue to talk in terms of \( F \) instead of \( I \).) In fact, while the details of the semantics of free variables (namely, the restriction to ordinary assignments) were motivated by semantic issues in programming languages, the corresponding details of the semantics of quantifiers are defined solely so that the Rule of Generalization and its converse hold.

Probably the most important inference rule used in this thesis is that of identity (equivalence) replacement. The rule is

\[ F_3 \alpha, F_3 \tau_1 \equiv \tau_2 \Rightarrow F_3 \text{REPL}(\alpha, \tau_1 \mapsto \tau_2). \]

This rule subsumes the substitution rule for propositional equivalents when applied to type \( \mathbb{B} \) terms. It holds for any terms \( \tau_1, \tau_2 \) of any type.

There is a corresponding rule for weak equality, namely

\[ F_3 \alpha, F_3 \tau_1 = \tau_2 \Rightarrow F_3 \text{REPL}(\alpha, \tau_1 \mapsto \tau_2) \]

but this rule is weaker than the strong equality rule because

\[ F_3 \tau_1 = \tau_2 \Rightarrow F_3 \tau_1 \equiv \tau_2 \]

These latter facts are the main reason we give greater prominence to \( = \) than to \( \equiv \) in PFC3.
3.4.6 Relationship of the Propositional Calculi PC2 and PC3 to the Partial Functional Calculus PFC3

In the usual predicate calculus, FC2, the propositional calculus PC2 is subsumed in the following sense: any wff of FC2 which is a substitution instance of a valid wff of PC2 is valid in FC2 as well. However, although the propositional calculus is subsumed, the lack of propositional variables in most treatments of the predicate calculus prevents the propositional calculus from being actually embedded in the predicate calculus.

But PC3 is both subsumed by, and embedded in, PFC3. We state this formally in the following two lemmas, which are obvious enough that we will not bother to prove them.

**Lemma 22:** Any wff in a PFC3-language which is a substitution instance of a valid wff in PC3 is valid in PFC3. □

**Lemma 23:** Formulae of PC3 can be identified with formulae in PFC3 by identifying propositional variables with B-variables and identifying the propositional connectives with the corresponding Type-B function symbols. When this is done, the truth value of any wff w of PC3 under an assignment φ is the same as that of the corresponding wff w' under the corresponding assignment φ'. □

A curious, and at first confusing, fact about PFC3 is that the two-valued propositional calculus PC2 is embedded in PFC3 (in the sense of Lemma 23) but not subsumed, by it (in the sense of Lemma 22). It is not subsumed because not every substitution instance of a valid wff of PC2 is valid in PFC3. For example

1/0=1 v ~(1/0=1)

is a substitution instance of the formula pv-p, which is valid in PC2. But under the usual interpretation of 0,1 and / (where / takes the value E when the denominator is 0) the above formula takes the truth value error instead of true. However, any valid wff formula from PC2, when viewed as a formula of PFC3, is valid in PFC3. This is because under the semantics of PFC3 we are restricted to ordinary assignments, so that B-variables range over only the
truth values \{true, false\}, and because on the values \{true, false\} the propositional connectives $\land, \lor, \neg, \Rightarrow, \equiv$, etc. agree with their two-valued counterparts. The relationships between PC2, PC3 and PFC3 are summarized, then, by the following examples.

$$
\begin{align*}
F_{PC2} & \equiv b \lor \neg b \\
F_{PC3} & \equiv b \lor \neg b \quad \text{(valid as a wff)} \\
F_{PC3} & \equiv A \lor \neg A \quad \text{(not valid as a schema)} \\
F_{PC3} & \equiv T(b) \lor F(b) \lor E(b) \\
F_{PC3} & \equiv T(b) \lor F(b) \lor E(b) \quad \text{(valid as a wff)} \\
F_{PC3} & \equiv T(A) \lor F(A) \lor E(A) \quad \text{(valid as a schema)}
\end{align*}
$$

3.5 Hoare-style Calculus of Firm Correctness

In this section we will develop a Hoare-style calculus of "firm correctness," which is a slightly more stringent correctness criterion than weak correctness. Such a calculus is fairly straightforward if we assume that the program's assertions are written in a language formalized in PFC3 and if we assume certain reasonable constraints on the primitives available in the programming language.

The notion we want to formalize is that of **firm correctness**. We will say informally that a program $S$ is **firmly correct** with respect to the precondition $P$ and postcondition $Q$, which we denote in this section by

$$P[S]Q$$

(using square brackets instead of the traditional curly brackets), if whenever program $S$ is executed in a state in which $P$ is true (not false or error) then $S$ will either infinitely loop or it will halt normally in a state such that $Q$ is true (not false or error.) The key phrase is "halt normally," which means that the program must not encounter any zerodivide, array bounds error or other condition which results from applying a partial function or predicate to arguments outside of its "normal" (i.e. non-error) domain.

Let us concern ourselves only with programs constructed from assignments, semicolon, if-then-else statements and while-do statements. We will assume that the terms permitted in the programs are unquantified expressions in some assertion language formalized in PFC3.
If we examine the informal definition of firm correctness just given we find that there is a slight ambiguity. What does it mean that the program should not "encounter" any runtime error. There are at least two possible definitions worth considering:

A program "encounters" an error if the final value of some Boolean expression used in a control statement is the value error, or if the final value of the expression on the right-hand side of an assignment is an error value.

A program "encounters" an error if an error value occurs as a result at any time during the evaluation of any expression, either a Boolean expression used in a control statement or an expression of any type used in an assignment statement.

These two definitions are not equivalent. The first essentially takes the position that no variable (nor the location counter) can take an error value. The second takes the position that error values are also not permitted on the runtime expression evaluation stack.

The two definitions become equivalent if we make some simple assumptions about the programming language and its implementation. We call a programming language regular if the following conditions hold:

1. Each primitive nonlogical function and predicate symbol of the language represents a naturally extended partial function, i.e. a partial function with the property that its value is an error value (of the appropriate type) at any of its arguments are error values.

2. The Boolean functions and and or, and the generic functions \((o | o)_x\) and \(\langle o | o | o \rangle_x\) for all types \(X\) are implemented in the usual way so that the first argument is evaluated, and then the second or third arguments are evaluated only if necessary.

3. The Boolean functions T, F, E, \&, \& and \&\& and the generic functions \(u_X\), \(z_X\) and \(\langle o | o | o | o \rangle_x\) do not occur at all in the programming language. For these functions there is no implementation which avoids evaluating error-valued arguments at times when the result is non-error. (Although \& and \& and \&\&\& are excluded, \(\text{wand}\) and \(\text{wor}\) are permitted.)

Under the above conditions it can be proved that an expression \(\alpha\) encounters an error during its evaluation by an interpreter if and only if its final value is an error value (of the appropriate type.) We omit the proof, which would be by induction on the structure of \(\alpha\).
know of no widely known programming languages which violate these conditions, so they must not be too unreasonable.

Here, then, are a list of Hoare-style rules for firm correctness of programs of in a regular programming language. Note that the requirements of the regularity assumption apply only to the programming language, not to the assertion language. Thus \( \land, \lor, =, \#, \supset, \) etc. are all permitted in assertions.

**Assignment Rule:**

\[
P \supset (\#e \land Q(e/x))
\]

\[
P[x:=e]Q
\]

This rule for firm correctness of assignment statements is analogous to Hoare’s rule for weak correctness of assignment statements. In order to prove \( P[x:=e]Q \) this rule states that it is sufficient to prove \( P \supset (\#e \land Q(e/x)) \), i.e. that when \( P \) is true, the expression \( e \) is defined and the expression \( Q(e/x) \) (i.e. \( Q \) with \( e \) substituted for all free occurrences of \( x \)) is true.

There is a sense in which this rule is obvious. The program \( x:=e \) cannot be firmly correct with respect to the predicates \( P, Q \) unless both \( P \supset Q(e/x) \) holds to establish weak correctness, and also \( P \supset \#e \) holds to establish that no error occurs during the evaluation of the expression \( e \). However, the rule is only “obvious” because we have taken care to define the truth tables for the three-valued connectives \( \supset \) and \( \land \), and also to define the meaning of the generic operator \( \# \) in such a way that a rule resembling Hoare’s this closely actually is sound.

Without some interpretational system of semantics we cannot prove soundness and relative completeness of this rule for assignment, nor any of the remaining rules. We will simply present the rules with a little explanatory prose and let that suffice.

**Composition Rule:**
This rule is entirely analogous to Hoare's Composition Rule, and no further comment seems necessary.

**If-then-else Rule:**

\[
P \supseteq \#\beta, \quad P \land \beta[S_1]Q, \quad P \land \neg\beta[S_2]Q
\]

\[
P \quad [\text{if } \beta \text{ then } S_1 \text{ else } S_2]Q
\]

Here the modification to Hoare's rule for if-then-else consists of requiring that no error occur during evaluation of the Boolean expression \(\beta\) (i.e. that \(P \supseteq \#\beta\) holds) and in requiring that \(S_1\) and \(S_2\) be firmly correct with respect to their specs rather than just weakly correct.

Notice again that the three-valued definitions of \(\land, \neg, \supset\) and \(\#\) make this rule look like less of a modification of Hoare's rule than it really is.

**While-do Rule:**

\[
P \supseteq \#\beta \land I, \quad \beta \land I[S] \#\beta \land I, \quad \neg\beta \land I \supset Q
\]

\[
P \quad [\text{while } \beta \text{ do } S]Q
\]

Once again, this is a modification of Hoare's rule for while-do loops. The main differences are the requirement that there be no errors in evaluating the Boolean expression \(\beta\) during the execution of the loop, and the requirement that the body of the loop be firmly correct with respect to its specifications, rather than just weakly correct.

**Rules of Consequence:**

\[
P \supseteq R, \quad R[S]Q
\]

\[
P[S]Q
\]

\[
P \supseteq S, \quad R \supset Q
\]

\[
P[S]Q
\]
These rules illustrate the significance of the $\supset$ connective in arguing about firm correctness. The truth table for $\supset$ is important here; if $p \supset q$ were defined to mean $\neg p \lor q$ this rule would be unsound.
4. The Theory BITZV

4.1 Definition of BITZV

In this chapter we define a formal language called BITZV, and its interpretation in terms of Booleans, integers, a totally ordered set, zsets and vectors. BITZV is formalized in the error PFC3 described in Chapter 3, as a first-order system with five types (or logical sorts) and three truth values, and it includes a large variety of function symbols useful for assertions about sorting, searching and merging programs.

When we refer to a formula of BITZV we will usually mean a formula with no quantifiers. BITZV is sufficiently rich in function symbols and data types so that correctness assertions and loop invariants for a large class of programs can be written without quantifiers. In fact this was a goal in the design of BITZV. And if the assertions have no quantifiers, it follows (for Algol-like languages) that the VC's need not contain any quantifiers. The reduction algorithms and decidability results we present will apply only to unquantified formulae of BITZV, and we will only have use for quantifiers in the justification of certain steps of the reduction algorithms in Chapters 6 and 7.

In Section 4.1 we discuss the five types of objects in BITZV. In Section 4.2 we formally define BITZV and discuss what it means for a formula of BITZV to be true. In Section 4.3 we illustrate the expressiveness of BITZV and discuss some of its limitations. While reading these sections it may be desirable to refer to the Appendix for a tabular summary of the language of BITZV. In Section 4.4 we discuss related work by other investigators and compare it to the work done in this thesis. And in Section 4.5 we present a Binary Insertion Sort program annotated with assertions which are expressed entirely in the language BITZV. Appendix IV gives a set of additional illustrative annotated programs, which further illustrate the expressive range of BITZV.

4.1.1 Type B -- Boolean

Type Boolean is the set of truth values \{true, false, error\}, together with a rather large
variety of truth functions, \( \neg, \land, \lor, \text{wand}, \text{wor}, \text{and}, \text{cor}, \#_B, \theta \), and all of the others described in Chapter 3. These functions are complete, in the sense that they are sufficient to define any three-valued Boolean function of any arity. There are also functions involving arguments or values of other types, such as \(<_1\) (integer less-than), \text{ORD} (vector orderedness), the various equalities \(=_P, =_T, =_Z\) and \(=_V\), etc., but these are more easily discussed in later subsections. Variables of Type B are permitted, as they are with any other type, and such variables serve both as Boolean variables in programs and also as ordinary propositional variables in assertions.

Since Type B should be thoroughly familiar from Chapter 3, we will not pursue it further.

4.1.2 Type I -- Integer

Type I is the set of integers \( \mathbb{Z} \) plus an error value \( E_1 \), together with the generic functions and the usual arithmetic functions and predicates (suitably extended to handle error values). We permit zeroary function symbols for the non-negative integers \( 0, 1, 2 \ldots \), together with unary negation \((-\)) , addition \((+\)) , subtraction \((-\)) , multiplication \((\times\)) , division \((/\)) , less than \((<\)) and all of the generic functions (which includes equality \((=\)) . For convenience we will from time to time use the symbols \(\leq, \geq\) and \(\neq\) with their usual meanings. It is not important whether division is interpreted as truncating toward 0 or truncating to the next lower integer, but we adopt the former convention. Division by 0 always produces the error value \(E_1\). All functions except the conditional functions and \(=_1\) are "naturally extended" so that if any argument is an error value, the result is \(E_1\).

An examination of Appendix III.2 will show that there are other integer-valued functions in BITZV, such as \(\text{lb}(v)\) and \(\text{size}(z)\), that take arguments of types other than Type I. These will be discussed in the sections devoted to those other types.

Since we explicitly permit multiplication and division (along with addition and subtraction) in the language BITZV, it is well known that the validity problem is not decidable, even for unquantified sentences. One of the primary results of this thesis, however, is that for those
unquantified sentences of BITZV which do not include any occurrences of *, /, or of the zset scalar multiplication operator *Z (described in Subsection 4.1.4), the validity problem is decidable. This is because the Type Reduction procedures of Chapters 6 and 7 never introduce *, /, or *Z if they are not present in the original wff. Thus the final sentences of BI will not contain * or / if the original sentence of BITZV from which the Type Reduction starts does not contain *, / or *Z.

A similar remark applies to any other function involving only Integers and Booleans. If Type I is extended to include any other functions, for example log2 (with appropriate handling of truncation and negative arguments), all of the Type Reduction algorithms apply unchanged. The only difference is that those sentences of BI to which sentences of BITZV are reduced may contain occurrences of log2, and BITZV will be recursive if and only if BI is. The description of Type I given here is thus a minimal one, not the maximal one that one might expect.

4.1.3 Type T -- a Totally ordered type

The position of Type T in BITZV is slightly different from that of Type B or Type I. In both of the other cases we have a particular model in mind as an interpretation. The intended interpretation for Type B is the three-valued Booleans and Boolean functions, and the intended interpretation for Type I is Z u {e_T} with the usual collection of arithmetic functions (suitably extended for the error value). In the case of Type T we do not have just a single model in mind, but rather a whole class of models: the class of all models which include a total ordering relation. A model for Type T is a parameter to BITZV, and as we will see, the interpretation of Types Z and V depend on the interpretation of Type T.

Type T, then, is interpreted as a set containing at least the value e_T, together with a binary predicate < which is a total ordering on the non-error values in T and which takes the value E_B when either of its arguments is E_T. We also require Type T to be equipped with the definedness predicate _T, the weak and strong equality predicates =_T and =_T, as well as the one-, two- and three-way conditional functions (b | t)_T, (b | t_1 | t_2)_T and (b | t_1 | t_2 | t_3)_T.
We further permit the predicates $\leq$, $>$, $\geq$ and the binary functions max and min, which are definable in the usual way for any total ordering.

There are other functions that produce values of Type T, but since they involve arguments of Type V or Type Z, we defer their description until the next two subsections.

Because T is interpreted as any totally ordered set, the results in this thesis apply to sorting and searching programs regardless of the type of the elements being sorted. They may be Booleans, integers, character strings or elements of the uncountable set of reals. We explicitly make no assumption about the cardinality or discreteness of the order type of Type T, nor do we require or prohibit maximal or minimal elements, etc.

As was the case with Type I, our specification of the functions permitted in Type T is minimal, rather than maximal. Any particular interpretation of Type T may include other functions or predicates involving only Types T and B. The Type Reduction algorithms of Chapters 6 and 7 will apply without change. For example, if Type T is interpreted as Real, then the arithmetic functions on real numbers may occur; if it is interpreted as Character String, then the function "convert-to-upper-case" may occur. In either case, the result of the Type Reduction will be sentences (in the language BT) that include occurrences of the extra function symbol(s), and BITZV will be undecidable if BT is.

4.1.4 Type Z -- Finite-Zsets-of-T

The notion of a zset is a simple generalization of the notion of a multiset. A multiset [Knuth 68] is like an ordinary set except that elements are considered to occur with a multiplicity of 0, 1, 2, or any natural number. Two multisets A and B are equal if all elements occur in A with the same multiplicity that they have in B. It should be obvious that a multiset m of elements of some type T can be represented (ignoring error elements for the moment) as a map $f_m: T \rightarrow \mathbb{N}$, where $f_m(t)$ is the number of occurrences of element t in multiset m, just as an ordinary set-of-T can be represented as a map from T into \{0, 1\}.

A zset is like a multiset except that elements are permitted to occur with any integral
multiplicity, positive, 0 or negative. (The "z" in "zset" is chosen to be reminiscent of the common mathematical symbol \( \mathbb{Z} \) for the integers, although in this thesis we use "I" to represent the integers.) By analogy with multisets, we say that two zsets A and B are equal if all elements have the same multiplicity in A that they have in B (positive, 0 or negative). Continuing the analogy, we can represent a zset \( z \) by a map \( f_z: T=\{E_T\} \rightarrow I=\{E_I\} \), where \( f_z(t) \) gives the signed number of occurrences of element \( t \) in zset \( z \). Instead of \( f_z(t) \) we will from now on use the notation \( z(t) \), chosen to parallel the notation \( v[i] \) for vector indexing.

In this thesis we will confine our attention to finite zsets. A finite zset is a zset in which no element occurs a (positive or negative) infinite number of times -- this has been implicit in the discussion already -- and in which only a finite number of elements occur with nonzero multiplicity. Thus, for any zset \( z \), \( z(t) = 0 \) for all but a finite number of \( t \in T \). Considered as a function of \( t \), the expression \( z(t) \) is "almost everywhere" 0. From now on, unqualified use of the term "zset" will always mean "finite zset". We will identify the notion of a multiset with that of a zset in which every element \( t \in T \) has nonnegative multiplicity.

Since I have chosen to introduce the notion of zset instead of using the well-known notion of multiset, I have some obligation to justify the choice. Basically, zsets have the same advantage over multisets that the integers have over the natural numbers. The presence of negatives allows simple manipulation of equations and inequalities and convenient canonical forms. Just as the integers form an Abelian group under addition, whereas the natural numbers do not, so the zsets form an Abelian group under zset-addition (defined below) whereas the multisets do not. Thus, although zsets do not usually find their way explicitly into searching and sorting programs, they are natural in reasoning about them.

In the system BITZV, then, Type Z is interpreted as the set of all (finite) zsets-of-T, plus an error value \( E_Z \), together with various functions to be described below. Type Z is actually parameterized by Type T, so that if T is Real then Z is interpreted as Zset-of-Real. If T is an uncountable type, then so is Z.

Because of the unfamiliarity of zsets we will carefully describe the subsumed by the
language of BITZV. The decision procedure given in their The assertion lan-
p.,uap_e defined by
Suzuki and Jefferson is mostly functions and predicates involving Type Z in a semi-formal
style. Please recall the convention in the following discussion that all bound variables range
over only the non-error elements of the appropriate type.

#z: Z → B
#z(z) is true if z is not the error value E_Z, and is false otherwise. Thus,
#z = ¬(z ≡ E_Z).

= Z: ZxZ → B
z₁ = z₂ is true if z₁ and z₂ are both defined (i.e. not E_Z) and are equal as
zsets; it is false if both z₁ and z₂ are defined, but are not the same zset;
and it is error if either argument is E_Z. Weak zset equality can be
defined by the identity (z₁ = z₂) ≡ (\forall t. z₁ <t> = z₂ <t>).

≤ Z: ZxZ → B
z₁ ≤ z₂ is defined as (\forall t. z₁ <t> ≤ z₂ <t>). It thus takes the value error iff one
or both of its arguments are E_Z. It is true when for all t ∈ T the
multiplicity in z₁ of t is less than or equal (in signed value) to its
multiplicity in z₂, and it is false otherwise. The ≤_Z relation is obviously a
partial order on Z-{E_Z}.

• <•>: ZxT → I
z <t> is the number of occurrences (positive, negative or 0) of the
element t in zset z. The value is E_I if z ≡ E_Z or t = E_T.

size: Z → I
size(z) is defined to be \(\sum_z z <t>\). It is the total number of elements
(positive, negative or 0) in zset z, unless z ≡ E_Z, in which case
size(z) = E_I. Since we are dealing only with finite zsets, the summation is
always defined, no matter what the cardinality of Type T is.

min_Z: Z → T
min_Z(z) is defined as \(\mu t. z <t> \neq 0\), i.e. the smallest t ∈ T (according to the
relation <_T) with a nonzero multiplicity in z. Of course, if z ≡ E_Z or if
z = \(\phi\) (the empty zset) then min_Z(z) ≡ E_T. Once again, because we are
dealing with finite zsets, min(z) is always defined for nonempty zsets, no
matter what the cardinality or order type of Type T. (It might be
remarked that this definition is somewhat arbitrary. We could have
deﬁned min(z) = \(\mu t. z <t> > 0\) with only insigniﬁcant consequences.)

max_Z: Z → T
Similar to min_Z above, with the obvious changes.

E_Z: Z
The Type Z error value.

(• | • | • | •)_Z: BxZ^3 → Z
The three-way conditional function for Type Z. \((b | z_1 | z_2 | z_3)_Z\) takes
the value z₁ if b is true, z₂ if b is false, or z₃ if b is error.

(• | • | •)_Z: BxZ^2 → Z
The two-way conditional function for Type Z. We define
\[(b \mid z_1 \mid z_2) \equiv (b \mid z_1 \mid z_2 \mid E_Z)_Z.\]

\[(° \mid °)_Z : B \times Z \rightarrow Z\] The one-way conditional function for Type-Z. We define \((b \mid z)_Z \equiv (b \mid E_Z \mid E_Z)_Z.\)

\(\phi_Z : \rightarrow Z\) The empty zset; has the property that \((\forall t. \phi(t) = 0) = \text{true}.\)

\(\text{neg}_Z : Z \rightarrow Z\) Zset negation; \(\text{neg}(z) = E_Z\) iff \(z = E_Z;\) otherwise \(\text{neg}(z)\) is the unique zset \(z'\) satisfying the relation \(\forall t. z'(t) = -z(t).\) Clearly since \(z\) is finite, \(\text{neg}(z)\) is also. The term \(\text{neg}(z)\) will almost always be denoted by \(-z,\) using the unary minus-sign.

\(+_Z : \times Z \rightarrow Z\) Zset sum; \(z_1 + z_2 \equiv F_Z\) iff \(z_1 \equiv F_Z\) or \(z_2 \equiv F_Z;\) otherwise \(z_1 + z_2\) is the unique zset \(z\) satisfying the relation \(\forall t. z(<t> = z_1(<t>) + z_2(<t>).\) Since \(z_1\) and \(z_2\) are finite, \(z_1 + z_2\) is also finite.

\(-_Z : Z \times Z \rightarrow Z\) Zset difference; \(z_1 - z_2\) is defined to be \(z_1 + (-z_2).\)

\(\{°\}_T : T \rightarrow Z\) \(\{°\}\) is the singleton zset containing one occurrence of \(t\) and \(0\) occurrences of every other element of Type T (unless \(t \equiv E_T,\) in which case \(\{t\} \equiv E_Z).\) Thus, \(\{t\}\) is the unique zset \(z\) satisfying the relation

\(z(<t>) = \{s=t \mid 1 \mid 0\})\)

\(\{°\}_V : V \rightarrow Z\) \(\{v\}\) denotes the zset of elements contained in vector \(v,\) disregarding the order of the elements in \(v,\) but respecting their multiplicity. (If \(v \equiv E_V,\) then \(\{v\} \equiv E_Z).\) Since elements cannot occur a negative number of times in a vector \(v,\) it is clear that \(\{v\}\) is actually a multiset, and that \(\phi \leq \{v\}\) (when \(v\) is not \(E_V\)). Also note that, since vectors as we define them are always finite length, the zset \(\{v\}\) is always finite.

\(*_Z : I \times Z \rightarrow Z\) Scalar multiplication. The zset \(i \cdot z\) has the value \(E_Z\) if \(i \equiv E_I\) or \(z \equiv E_Z;\) otherwise it is the unique zset \(z'\) such that \(\forall t. z'(t) = i \cdot (z(<t>).\) Clearly \(i \cdot z\) must be finite, since \(i\) and \(z\) are.

Because we consider only finite zsets it should be clear that all members of \(Z\) are computationally representable (provided that all members of \(T\) are representable). Perhaps the most straightforward representation is to view a zset as an almost-everywhere-zero map from \(T-\{E_T\}\) to \(I-\{E_I\},\) and to store a zset as a list of \(<\text{element}, \text{multiplicity}\>\) pairs. A special value would have to be reserved to represent \(E_Z.\)

We noted earlier that \(Z-\{E_Z\}\) forms an Abelian group under zset addition. When we also consider the scalar multiplication operation \(i \cdot z\) we note that \(Z-\{E_Z\}\) forms a module over the
ring of integers \( \mathbb{Z} \). (A module is like a vector space except that the scalars need not have multiplicative inverses, thus forming a ring, rather than a field.) The set of all unit zsets, 
\[ \{ \{ t \} \mid t \in \mathbb{T} \} \]
forms an orthogonal basis for the module \( \mathbb{Z} \), and \( \mathbb{Z} \) considered as a module has a finite, countable, or uncountable number of dimensions according to whether Type \( \mathbb{T} \) is finite, countable or uncountable respectively. We will use several properties of Type \( \mathbb{Z} \) considered as a module, including the following two.

**Lemma 1:** Every non-error zset can be represented in a canonical way as a finite sum of the form \( i_1 \{ t_1 \} + i_2 \{ t_2 \} + \ldots + i_n \{ t_n \} \) where \( i_1, i_2, \ldots, i_n, \) and \( u_1, u_2, \ldots, u_n \) and all of the \( t_1, t_2, \ldots, t_n \) are distinct.

**Proof:** Trivial. \( \square \)

**Lemma 2:** Let \( t_1, t_2, \ldots, t_n \) be defined and distinct. Then
\[
i_1 \{ t_1 \} + i_2 \{ t_2 \} + \ldots + i_n \{ t_n \} = \phi_Z
\]
if and only if \( i_1 = 0 \) and \( i_2 = 0 \) and \( \ldots \) and \( i_n = 0 \) are all true. Similarly,
\[
i_1 \{ t_1 \} + i_2 \{ t_2 \} + \ldots + i_n \{ t_n \} \leq \phi_Z
\]
if and only if \( i_1 \leq 0 \) and \( i_2 \leq 0 \) and \( \ldots \) and \( i_n \leq 0 \) are all true.

**Proof:** Trivial. \( \square \)

The set of functions and predicates described above for Type \( \mathbb{Z} \) is maximal for the algorithm of Chapter 7. The addition of any new function or predicate involving Type \( \mathbb{Z} \) would require at least some change to the Type Reduction from BITZ to BIT. This is in contrast to Types B, I and T, for which we have great freedom to add new functions without altering the Type Reductions. It is, however, possible to add functions involving Type \( \mathbb{Z} \) but not Type \( \mathbb{V} \), and still apply the Type Reduction of Chapter 6 (BITZV \( \rightarrow \) BITZ) without change.

### 4.1.5 Type \( \mathbb{V} \) -- Vector-of-\( \mathbb{T} \)

Type \( \mathbb{V} \) is basically the set of all vectors of elements of Type \( \mathbb{T} \). Thus, like Type \( \mathbb{Z} \), the interpretation of Type \( \mathbb{V} \) is parameterized by the choice of an interpretation for Type \( \mathbb{T} \). If
Type T is Real, then Type Z is Zset-of-Real, and Type V is Vector-of-Real. Type V is countable or uncountable according to whether T is countable or uncountable.

In this thesis a vector will consist of a pair of integers \((l,u)\) such that \(l \leq u + 1\), together with a function from the closed interval \([l,u]\) into \(T\). For a vector \(v\) the integer \(l\) associated with \(v\) is denoted \(lb(v)\) (lower bound) and the integer \(u\) is denoted \(ub(v)\) (upper bound). Two vectors \(v_1\) and \(v_2\) are equal if and only if \(lb(v_1) = lb(v_2)\) and \(ub(v_1) = ub(v_2)\) and the two functions are identical. In addition there is one special element of Type V, the error vector \(E_Y\).

The use of the word vector for Type-V is in the computer scientists' sense of "a sequence of elements", not in the mathematicians' sense of "element of a vector space." We do not define any notion of "addition" on vectors, nor any notion of "scalar multiplication," and there are no properties of our vectors that are analogous to the linearity properties of vector spaces.

Our definition of vectors has several important properties. First, each vector has finite length. For theorem-proving purposes infinite-length vectors are much easier to handle than finite-length vectors; one never has to worry about array bounds violation for one thing. Also, vectors manipulated by programs are finite in length, so if we wish to verify actual programs we must forego the considerable combinatorial savings in proof algorithms that is possible by considering vectors to be infinite. We should also note that the choice of finite vectors meshes well with our restriction to finite sets in Type Z.

A second important property is that two vectors can be equal only if they have the same lower- and upper-bounds. We do not consider two vectors to be equal if they merely have the same elements in the same order with the indices all offset by some value. Such an equality concept, "equal as strings", may be very useful, but it appears also to be difficult to handle formally. I do not know whether it is possible to perform a Type Reduction \(BITZV \rightarrow \) BITZ if "equal as strings" is included in the language as a vector predicate.

From one point of view it is not surprising that for vectors to be considered equal their
lower- and upper-bounds must be equal. Vectors are often said to represent finite mappings [Dahl 72], and from a set theoretic point of view two mappings cannot be equal if they have different domains. But our definition has curious consequences in the case of empty vectors. The empty vector whose lb is 1 and ub is 0 is not equal to the empty vector whose lb is 3 and ub is 2. In fact, there are an infinite number of distinct empty vectors according to our definition, one for each integer i, having lower bound i and upper bound i-1. If one takes the point of view that vectors represent functions only, rather than a function plus bounds, then two empty vectors should represent the same function on the same domain, namely, the unique empty function on the empty domain, and they should be considered equal. Thus, according to the definition given here, a vector v must not be thought of as representing simply a map from \( \{ i \mid lb(v) \leq i \leq ub(v) \} \) into \( \{E_T\} \). A vector includes the endpoints \( lb(v) \) and \( ub(v) \) as part of its value. It may seem perverse to insist on this definition, but the alternative definition turns out to be more complicated to deal with formally. The reason seems to be related to the notion of a subvector. If we wish to test whether \( v[i..j] \) is a subvector of \( v[m..n] \), then under our definition we would check the relation

\[
\#v \land \#m \land \#n \land \#i \land \#j \land \\
\text{lb}(v) \leq m \land m \leq i \land i \leq j + 1 \land j \leq n \land n \leq \text{ub}(v).
\]

When both vector terms are known to be defined (not \( E_v \)) we can simplify this to

\[
m \leq i \land j \leq n.
\]

But, if all empty vectors are considered equal, then the empty vector is a subvector of any other vector, and the test (assuming again that both terms \( v[i..j] \) and \( v[m..n] \) are known to be defined) would be

\[
(m \leq i \land j \leq n) \lor j = i-1.
\]

The disjunction forces two cases to be considered for one test, and the number of cases to be considered for \( k \) tests is \( 2^k \). This is an exponential explosion which it seems wise to avoid.

Another way of saying the same thing is that when we know that vector \( v[i..j] \) is disjoint from (does not "overlap") vector \( v[m..n] \), we would like to conclude that "no subvector of
4.1.5

$v[i..j]$ is a subvector of $v[m..n]$. But when all empty vectors are identified, we can only conclude that "no subvector of $v[i..j]$ except the empty vector is a subvector of $v[m..n]$". The exception is another combinatorial nuisance.

A final point to notice about Type V is that it includes vectors of all lengths and bounds; we do not consider vectors of length 10 to be a different type from vectors of length 11. There would be no real problem with making this type distinction; we would just have the nuisance of an infinite number of vector types, and an infinite number of function and predicate symbols to relate them.

The view of vectors presented here is essentially the same as the view presented by Dijkstra [Dijkstra 76]. We will show later in this chapter how much of the rest of Dijkstra's vector handling can be embedded in BITZV.

We now define all of the functions and predicates involving Type V that will be used in this thesis.

$\#_V: V \rightarrow B$

$\#_V(v)$ is true if $v$ is not the error value $E_V$; it is false otherwise. Thus,

$\#_V(v) = (\neg v \in V) E_V$

$=_V: V \times V \rightarrow B$

$v_1 = v_2$ is true if $v_1$ and $v_2$ are both defined (i.e. not $E_V$) and if $lb(v_1) = lb(v_2)$ and $ub(v_1) = ub(v_2)$ and $v_1$ and $v_2$ are equal as functions; it is error if $v_1$ or $v_2$ is the error value $E_V$; in all other cases $v_1 = v_2$ is false. Thus, weak vector equality can be defined by the identity

$(v_1 =_V v_2) \equiv (lb(v_1) = lb(v_2) \wedge ub(v_1) = ub(v_2) \wedge (\forall i, lb(v_1) \leq i \wedge i \leq ub(v_1) \Rightarrow v_1[i] = v_2[i]))$

$\equiv_V: V \times V \rightarrow B$

$v_1 \equiv v_2$ is true if $v_1$ and $v_2$ are the same element of $V$, i.e. if $v_1$ and $v_2$ are both $E_V$, or if neither are $E_V$ and $v_1 = v_2$; it is false otherwise. The $\equiv_V$ relation never takes the value error. Strong vector equality (identity) can be defined by the identity

$(v_1 \equiv_V v_2) \equiv (lb(v_1) = lb(v_2) \wedge ub(v_1) = ub(v_2) \wedge (\forall i, lb(v_1) \leq i \wedge i \leq ub(v_1) \Rightarrow v_1[i] = v_2[i]))$

$\text{ord: } V \rightarrow B$

$\text{ord}(v)$ is true if $v$ is defined (not $E_V$) and if all of the elements of $v$, from $v[\text{lb}(v)]$ to $v[\text{ub}(v)]$, are sorted in non-decreasing order according to the relation $\prec_T$; $\text{ord}(v)$ is false if $v$ is defined, but the elements are not in non-decreasing order; it is error if $v$ is not defined. We can define $\text{ord}(v)$ with the identity
ord(v) \equiv (\forall i . (lb(v) \leq i < \text{ub}(v)) \lor v[i] \leq v[i+1]).

Note that if v is an empty vector, then ord(v)=true.

lb: V \to I

lb(v) is the first of the pair of integers defining the interval which is the
domain of vector v. If v is EV, then lb(v) = E_1. If v is not an empty vector
then lb(v) is the least i such that v[i] is defined. In all cases the relation
\forall v \exists lb(v) \leq \text{ub}(v)+1 is true.

ub: V \to I

ub(v) is the second of the pair of integers defining the interval which is the
domain of vector v. If v is EV, then ub(v) = E_1. If v is not an empty vector,
then ub(v) is the largest i such that v[i] is defined. In all cases the relation
\forall v \exists lb(v) \leq \text{ub}(v)+1 is true.

len: V \to I

len(v) is the length of vector v, unless v is EV, in which case len(v) = E_1.
In all cases len(v) = \text{ub}(v) - \text{ib}(v) + 1.

_o[0]: V \times I \to T

This is the vector access function. The term v[i] has the value E_T if v is
EV or if i is E_1, or if lb(v) > i or i > ub(v). Otherwise v[i] is the result of
applying the function-part of v to argument i.

_elem: V \to T

elem(v) has the value E_T except when lb(v) = ub(v) is true, i.e. when
len(v) = 1. When len(v) = 1 then elem(v) = v[lb(v)]. Note that elem is
defined only for vectors of length 1. [This function is not properly part
of BITZV; it is introduced into the language during Chapter 6.]

min_V: V \to T

min(v) is the smallest element, according to the relation \langle_T, occurring in
vector v. Its value is E_T if v is EV or if v is an empty vector. Using zsets,
the min_V function is defined by the identity min_V(v) = \min_Z(\{v\}).

max_V: V \to T

max(v) is the largest element, according to the relation \langle_T, occurring in
vector v. Its value is E_T if v is EV or if v is an empty vector. Using zsets
the max_V function can be defined, in a way similar to the definition of
min_V, by the identity max_V(v) = \max_Z(\{v\}).

\{0\}_V: V \to Z

\{v\} is the z-set of elements occurring in vector v. Thus, any element of
Type-T occurring exactly k times in the range of vector v has a
multiplicity of k in \{v\}. If v is any empty vector, then \{v\} is the empty
zset \emptyset. If v is EV, then \{v\} is E_2.

EV: \to V

EV is the unique error element for Type V.

(\circ \mid \circ \mid \circ \mid \circ)_V: B \times V^3 \to V

The three-way conditional function for Type-V.

(\circ \mid \circ \mid \circ)_V: B \times V \times V \to V

The two-way conditional function for Type-V.

(\circ \mid \circ)_V: B \times V \to V

The one-way conditional function for Type-V.

(\circ [\circ \ldots \circ]): V \times I \times I \to V

v[i_1 \ldots i_2] is the restriction of v to the interval [i_1 \ldots i_2]. When v[i_1 \ldots i_2]
is defined, its lb is i_1, and its ub is i_2, and it represents a mapping from
4.1.5

[i₁...i₂] to T-{E₇} such that for all i ∈ [i₁...i₂] the identity (v[i₁...i₂])'[i] = v[i] holds. The term v[i₁...i₂] is defined when v,i₁ and i₂ are all defined and satisfy the relation lb(v) ≤ i₁ ≤ i₂ + 1 and i₂ ≤ ub(v); otherwise v[i₁...i₂] has the value Eᵥ.

<0,0,o> : VXlxT → V <v,i,t> represents the result of assigning the value t to position i in vector v. It is defined only when v, i and t are all defined and lb(v) ≤ i ∧ i ≤ ub(v) is true; otherwise it is undefined. When defined, <v,i,t> is the unique vector having the property that <v,i,t>[j] = (i=j | t | v[j])ₜ. The assignment statement v ← <v,i,t> is normally written v[i] ← t in most programming languages.

4.2 Validity in BITZV

It may be belaboring the obvious, but it seems wise to define carefully the notion of a "true" or "valid" formula in the language of BITZV. The general form of the definition is simple.

Definition 3: A wff w in the quantified language for BITZV is true or valid in the system BITZV iff for all standard interpretations J and for all ordinary assignments ϕ of values to variables, if Vₖφ, is the valuation induced by J,φ then Vₖφ(w) = true. We write BITZV ⊨ w, or simply ⊨ w.

It now remains to define all of the underlined terms.

The quantified language for BITZV should already be understood. It is the three-valued first order language of five logical sorts formalized in PC3 and including all of the operators described earlier in this chapter.

Definition 4: By a standard interpretation for BITZV we mean an interpretation with these additional properties:

1. The domain of interpretation for Type I is Z u {e₁} where Z is the set of standard integers and where e₁ is distinct from all integers and is the interpretation of E₁. The nonlogical arithmetic function symbols are assigned their usual interpretations on Z, and are extended to have the value E₁ whenever any argument is E₁.

2. The domain of interpretation for Type T is a set T u {e₇} where T is nonempty and e₇ is distinct from all elements of T. The value e₇ must be the interpretation...
of \( E_T \). There must be a relation \(<\succ T\) which, when confined to \( T \), is a total ordering, but which takes the truth value error when either of its arguments is \( e_T \). The symbol \( <_T \) must be interpreted as the relation \(<\succ T\), and the symbols \( \min_T, \max_T, >_T, \leq_T \) and \( \geq_T \) must have the usual compatible interpretations.

3. The domain of interpretation for Type-Z must be the set of all finite zsets of elements of \( T \), together with a distinct element \( e_Z \). The interpretation of \( E_Z \) must be \( e_Z \), and the interpretations of all other function symbols involving Type-Z are fixed by the choice of \( I \) and \( <\succ T\) according to Subsection 4.1.4.

4. The domain of interpretation for Type-V must be the set of all vectors of elements of \( T \), together with a distinct element \( e_V \). The interpretation of \( E_V \) must be \( e_V \), and the interpretations of all other functions involving Type-V are fixed by the choice of \( J \) and \( <\succ T\) according to Subsection 4.1.5.

Clearly the notion of a standard interpretation for BITZV is highly constrained. The only flexibility is in the choice of \( T \) and \( <\succ T\).

Definition 5: An ordinary assignment of values to variables is a function \( \varphi \) from the set of all variables in the language of BITZV into nonerror values in the domains of the respective types.

The restriction to nonerror values reflects the fact that, in the usual semantics, program variables cannot contain error values. Error values may arise in computation, but they are not stored in variables.

The notion of a valuation induced by an interpretation and an assignment was defined in Chapter 3 in a way similar to that of the usual definition for first order logic, but slightly extended for the five-typed, three-valued case.

Referring back to Definition 3, it is important to note that for a formula to be valid it must take the value true under any (standard ordinary) valuation. It is not sufficient for it to be merely non-false.

This definition of validity is entirely semantic, as it should be. We do not develop a proof theory for BITZV, but instead appeal directly to this definition (implicitly) in our algorithms of Chapters 6 and 7.
4.3 Expressiveness of BITZV

In this section we will investigate the expressive boundaries of the language BITZV. There are many useful functions and predicates directly expressible (or at least representable) in BITZV but which are not obviously so at first glance. Conversely, there are also a large number of notions which seem closely related to BITZV, but which are apparently not representable in it. For a proper evaluation of the results in this thesis we must be explicit about both, and in the rest of this chapter we give a variety of examples.

In Appendix IV we present a sample of sorting, merging, searching and miscellaneous programs which are completely annotated with inductive assertions in the language of BITZV. These programs illustrate what is probably the maximum useful scope of BITZV.

4.3.1 Directly expressible functions

In this section we illustrate a variety of functions and predicates that are directly expressible in BITZV.

**Definition 6:** We say that a (partial) function \( f: S_1 \times \ldots \times S_n \rightarrow S_{n+1} \) is directly expressible without quantifiers in BITZV iff all of \( s_1, s_2, \ldots, s_{n+1} \) are among the types \( B, I, T, Z \) and \( V \), and if there is some term \( u \) of Type \( S_{n+1} \) in BITZV containing \( n \) free variables (of the types \( S_1, \ldots, S_n \) respectively) with no quantifiers, such that \( u \) evaluates to \( f(s_1, \ldots, s_n) \) whenever the free variables of \( u \) have the values \( s_1, \ldots, s_n \).

The notion of a function being directly expressible without quantifiers is in contrast to the more general notion of a function being Boolean representable without quantifiers, which is discussed in the next section.

For assertions about vector searching programs, predicates such as the following are necessary. We sometimes need to write "Element \( t \) occurs in vector \( v \) between \( v[i] \) and \( v[j] \)" which can be directly expressed in BITZV as

\[ \{v[i..j]\}<\|angle > 0. \]
Informally, this says that the number of occurrences of $t$ in the zset (multiset) of elements in vector $v[i..j]$ is greater than zero. More precisely, the expression takes the truth value \texttt{error} if any of $v$, $i$, $j$ or $t$ is an error value, or if $i$ and $j$ do not satisfy $\text{lb}(v) \leq i \land i \leq j+1 \land j \leq \text{ub}(v)$. Otherwise the expression is \texttt{true} or \texttt{false} according to whether or not $t$ occurs at least once in the zset $\{v[i..j]\}$. Note that $\{v[i..j]\}$ is a proper multiset, so that it is not possible for $\{v[i..j]\}<t> \to \text{to be negative.}$

When we are searching for the first occurrence of a key, we have to be able to say "Integer $x$ is the least index greater than or equal to $i$ of the element $t$ in the vector $v$". This is directly expressible in BITZV as

$$\{v[i..x-1]\}<t> \equiv 0 \land v[x] \equiv t.$$  

The first conjunct means that there are no occurrences of $t$ in the subvector $v[i..x-1]$ and the second conjunct says (obviously) that $v[x]$ is $t$. This expression is always \texttt{true} or \texttt{false}. It never takes the value \texttt{error} because the strong equality symbols are used instead of weak equality. If it is desirable to have the expression take the value \texttt{error} in some cases, then it can be modified in various ways. Note that as written, the expression is \texttt{false} for the cases when $v$, $i$, $x$ or $t$ are undefined, when $i$ or $x$ are out of the bounds of $v$, or when $x<i$.

For sorting and merging programs, various predicates about order and permutation are needed. To say "Vector $u$ is a permutation of vector $v$", we can write

$$\{u\} = \{v\}.$$  

To assert that "Vector $u[i..i+c]$ is a permutation of $v[j..j+c]$" we write

$$\{u[i..i+c]\} = \{v[j..j+c]\}.$$  

For a merging program we might need to assert "The multiset of elements in $u[i..j]$ combined with those in $v[k..l]$ is the same as the multiset of elements in $w[m..n]$." We can express this in BITZV as

$$\{u[i..j]\} + \{v[k..l]\} = \{w[m..n]\}.$$  

The above three examples show that the purpose of zsets in the language for BITZV is a
thinly disguised way of handling the permutation notion. We could have added to the
language a predicate \( \text{perm}(v_1, v_2) \), meaning \( v_1 \) is a permutation of \( v_2 \), but such a predicate
cannot capture the meaning of the last example unless some kind of vector union or
concatenation operation is permitted. But the concatenation function, or anything like it,
seems to be very difficult to handle in combination with the other operators of BITZV using
the methods of this thesis.

In most programs the sets that actually arise are in fact multisets. It might be thought
that the possibility of negative multiplicities in sets would prevent the notions needed for
program verification from being expressed precisely. This is not the case, however, because
it is always possible to assert that a set \( z \) is in fact a multiset using the formula
\[
\phi \leq z.
\]
The inequality states that the multiplicity of every element in \( z \) is greater than or equal to its
multiplicity in \( \phi \), namely 0, which is exactly the requirement for a set to be a multiset.

In loop invariants for sorting programs it is frequently necessary to assert that all
elements in one part of a vector are less than or equal to a particular element. This can be
expressed, for example, as
\[
\max(v[i..j]) \leq v[j+1].
\]

Sometimes a sort program partitions a vector into two parts, with all elements in the low
end of the vector being less than or equal to all those in the upper end. Such a situation can
be expressed by the formula
\[
\max(v[\text{lb}(v) .. i]) \leq \min(v[i+1 .. \text{ub}(v)]).
\]
Expressions such as these, in combination with others using the ord predicate, give great
flexibility in writing assertions about the intermediate state of sorting and merging programs.

In his book [Dijkstra 76] Dijkstra uses dynamically expanding and contracting vectors as a
data type in his personal programming language. The expansion and contraction occur when
the operations \( \text{loext}, \text{hiext}, \text{lpop} \) and \( \text{hipop} \) are invoked. The latter two functions are
directly expressible in BITZV as
\[ \text{lopop}(v) = v[\lfloor b(v) \rfloor + 1..ub(v)], \quad \text{and} \]
\[ \text{hipop}(v) = v[lb(v)..ub(v)-1]. \]

We will see in the next subsection that the loext and hiext functions, while not directly expressible in BITZV (without quantifiers), are Boolean representable in BITZV (without quantifiers). The presence of expanding and/or contracting vectors in a program, does not prevent its verification from being carried out within BITZV.

For any constant \( k \), the function \( \text{select}(v,k) \), which returns the \( k^{th} \) largest element of vector \( v \), is directly expressible in BITZV in the following way:
\[
\text{select}(v,1) = \max \{ \{ v \} \} \\
\text{select}(v,k) = \max \{ \{ v \} \} - \{ \text{select}(v,k-1) \} - \ldots - \{ \text{select}(v,1) \}.
\]

Unfortunately it is apparently not possible to express \( \text{select}(v,k) \) in BITZV when \( k \) is not a constant (but I have no proof). Note that \( \text{select}(v,k) \) has the value \( E_T \) when the length of \( v \) is less than \( k \) (or when \( v = E_Y \)).

In most of the examples above the expressions can take any of the three truth values. Sometimes it is convenient to lump \( \text{false} \) and \( \text{error} \) together as \( \text{false} \). To do this one need only surround the assertion with the Boolean operation \( T \), since \( T(p) \) is \( \text{true} \) if \( p \) is \( \text{true} \), and \( \text{false} \) if \( p \) is \( \text{false} \) or \( \text{error} \). We should also note the probably obvious fact that the expressions above are really schemas, in the sense that, wherever we have shown a simple variable symbol such as \( v \) or \( i \), it is permissible to substitute any term (of the appropriate type).

### 4.3.2 Boolean Representable Functions

In the previous subsection we gave several examples of predicates and functions which are directly expressible by terms of BITZV. This is an instance of the more general notion of Boolean representability.

**Definition 7**: A partial function \( f:S_1 \times \ldots \times S_n \rightarrow S_{n+1} \) is **Boolean-representable** iff there are
two formulae $R_f(s_1,\ldots,s_n,s_{n+1})$ and $D_f(s_1,\ldots,s_n)$, where the $s_i$ are free variables of type $S_p$ and $R_f$ and $D_f$ contain no other free variables, such that

$$E \left( f(s_1,\ldots,s_n) = s_{n+1} \right) \equiv B R_f(s_1,\ldots,s_n,s_{n+1})$$

and

$$E \equiv f(s_1,\ldots,s_n) \equiv B D_f(s_1,\ldots,s_n).$$

**Definition 8**: A partial function $f$ is **Boolean-representable without quantifiers** iff it is representable by formulae $R_f$ and $D_f$ that contain no quantifiers.

These definitions apply to any system formalized in PFC3, although they are used in this thesis for the special case where the types $S_1,\ldots,S_{n+1}$ are among $B, I, T, Z$ and $V$. Note that the definitions are well-defined because all types include $=$ and $\#$ operators. In this case the $=$ and $\#$ functions on the right-hand sides are for Type $S_{n+1}$.

It would have been nicer to define Boolean representability more cleanly, saying that $f:S_1 \times \ldots \times S_n \to S_{n+1}$ is representable iff there is a formula $R_f(s_1,\ldots,s_{n+1})$ such that $R_f(s_1,\ldots,s_{n+1}) \equiv B \left( f(s_1,\ldots,s_n) = s_{n+1} \right)$. Because of our assumption that variables cannot be assigned error values, however, the variable $s_{n+1}$ cannot be assigned the error value for Type $S_{n+1}$. A second expression to handle the undefined case is therefore necessary in the definition.

The value of the notion of Boolean representability is this: the validity problem for any extension of BITZV by functions which are Boolean representable without quantifiers can be reduced to the validity problem for BITZV itself. Suppose a new function $f:S_1 \times \ldots \times S_n \to S_{n+1}$ is representable in BITZV by $R_f(s_1,\ldots,s_{n+1})$ and $D_f(s_1,\ldots,s_n)$. Suppose further that a wff $w$ contains an occurrence of $f(u_1,\ldots,u_n)$. Then, for a new variable $v'$ of type $S_{n+1}$, $w$ is valid if and only if

$$E \left( R_f(u_1,\ldots,u_n,v') \land \text{REPLALL}(w, f(u_1,\ldots,u_n) \Rightarrow v') \right) \land$$

$$E \left( D_f(u_1,\ldots,u_n) \land \text{REPLALL}(w, f(u_1,\ldots,u_n) \Rightarrow E) \right)$$

where $E$ is the error constant for Type $S_{n+1}$. By repeated application of this substitution any occurrences of Boolean representable functions can be systematically removed.
There do in fact exist useful functions which are apparently not directly expressible in \( \text{BITZV} \) but which are Boolean representable without quantifiers. A good example is Dijkstra's hiext function ([Dijkstra 76]). By hiext(\(v, t\)) (extension of a vector at the high end) Dijkstra meant the unique vector \(v'\) such that \(\text{lb}(v') = \text{lb}(v), \text{ub}(v') = \text{ub}(v)+1, v'[i] = v[i] \) for all \(i\) between \(\text{lb}(v)\) and \(\text{ub}(v)\), and \(v'[\text{ub}(v')] = t\). We extend the meaning here so that hiext(\(v, t\)) ∈ \(E_V\) whenever \(u \in E_V\) or \(t \in E_T\). I claim (but cannot prove) that there is no Type-V expression \(A(v,t)\) in \(\text{BITZV}\) such that
\[
A(v,t) ≡ V \text{ hiext}(v,t).
\]
However, hiext(\(v, t\)) is Boolean representable without quantifiers in \(\text{BITZV}\) by the following two expressions
\[
\text{hiext}(v,t) ≡ B \left( v' ∨ t \wedge \text{hiext}(v,t) \right),
\]
\[
\# \text{hiext}(v,t) ≡ B \left( v \wedge t \right).
\]
Note that \(v'\) is a new variable, while \(v\) and \(t\) are arbitrary terms.

Another function Boolean-representable without quantifiers is the vector copy function. Let \(\text{vcopy}:V \times V \rightarrow V\) be defined so that \(\text{vcopy}(v_1, v_2)\) is the unique vector \(v'\) such that \(\text{lb}(v') = \text{lb}(v_1), \text{ub}(v') = \text{ub}(v_1)\) and such that \(v'\) agrees with \(v_1\) everywhere except on the subinterval \(\text{lb}(v_2), \text{ub}(v_2)\). On that interval we define \(v'[\text{lb}(v_2), \text{ub}(v_2)] = v_2\). Obviously if \(v_1\) or \(v_2\) is undefined, or if the domain of \(v_2\) is not included in the domain of \(v_1\), then \(\text{vcopy}(v_1, v_2)\) is undefined.

Although \(\text{vcopy}\) is not expressible by a V-term, it is Boolean representable without quantifiers by the following pair of predicates:
\[
\text{vcopy}(v_1, v_2) ≡ v' = \left( v_1 ∨ v_2 ∨ \text{lb}(v_1) ≤ \text{lb}(v_2) \wedge \text{ub}(v_2) ≤ \text{ub}(v_1) \wedgeight.
\]
\[
\text{lb}(v') = \text{lb}(v_1) \wedge \text{ub}(v') = \text{ub}(v_1) \wedge
\]
\[
v_1[\text{lb}(v_1), \text{lb}(v_2)-1] ≡ v'[\text{lb}(v'), \text{lb}(v_2)-1] \wedge
\]
\[
v_2 ≡ v'[\text{lb}(v_2), \text{ub}(v_2)] \wedge
\]
\[
v_1[\text{ub}(v_2)+1, \text{ub}(v_1)] ≡ v'[\text{ub}(v_2)+1, \text{ub}(v_1)].
\]

\#vcopy(v_1, v_2) ≡ \left( v_1 ∨ v_2 ∨ \text{lb}(v_1) ≤ \text{lb}(v_2) \wedge \text{ub}(v_2) ≤ \text{ub}(v_1) \right)
I do not know if there are any other interesting examples of Boolean representable functions in BITZV besides vcopy, hiext, and loext. But in judging the expressiveness of a language it is the representable functions which matter, not just those directly expressible.

Since vector access, length and assignment operations are primitives in BITZV, and since we have shown that hipop and hiext operations for deleting and extending at the high end of a vector are representable in BITZV, it might seem that the type Stack-of-T with the usual operations push, pop, top, empty? and s_stack would be representable in BITZV. This is indeed the case. We can express and represent the stack algebra as follows.

\[
\begin{align*}
\text{isstack?}(v) & \equiv \text{lb}(v) = 1 \\
\text{empty?}(v) & \equiv \text{ub}(v) = 0 \\
\text{top}(v) & \equiv v[\text{ub}(v)] \\
\text{pop}(v) & \equiv v[\text{lb}(v), \text{ub}(v) - 1] \\
\text{push}(v, t) \text{ represented by hiext}(v, t) \\
\text{v}_1 \text{s_stack} \text{ v}_2 & \equiv \text{v}_1 \oplus \text{v}_2
\end{align*}
\]

The entire stack algebra involving those operations can be encoded into BITZV, and any Stack-of-T equation can be verified or refuted by the methods in this thesis.

4.3.3 Limitations on expressiveness of BITZV

Although we have spent several pages describing what can be expressed in BITZV, it is important to get a good feeling for what cannot be expressed.

Some features not currently expressible in BITZV could be added without much difficulty, for example record data types with the usual access and assignment functions, or multidimensional Arrays-of-T, again with access and assignment functions. But many notions which seem "close" to notions expressible in BITZV, however, are actually outside of its boundaries. All of the nonexpressibility results claimed here are tentative: they seem apparent, but I have no proofs.

We have casually indicated that the type Multiset-of-T can be represented within Type-Z. The predicate "z is a multiset" is expressed as \( \phi \leq z \), and the =, \leq, +, size, \{t\} and \(<t\>\) operations, when restricted to multisets in z, represent the corresponding multiset operations.
However, the fact that the type Multiset-of-T is representable in BITZV does not imply anything about the type Set-of-T. The type Set-of-T apparently cannot be represented in BITZV without quantifiers. One might think of representing sets as those zsets in which a finite number of Type-T values have multiplicity 1 and all others have multiplicity 0. But when using that representation there is apparently no way to represent the predicate "z is a set" in BITZV without a quantifier. The same is true for the complement, union and intersection operations on sets. Other possible representations have similar problems. Thus, it is possible to verify using the method of this thesis, a program to test whether or not two vectors contain the same multiset of elements, but one cannot verify a program which tests whether or not two vectors contain the same set of elements. The reason is that one cannot even write necessary postconditions in BITZV. One could, of course, propose adding Type S = Set-of-T to BITZV, but it is not certain that the methods of this thesis can be extended to cover this additional type.

Another serious limitation of BITZV is its inability to represent strings and string operations. We might choose to represent strings as vectors whose lb is 1. In such a case the predicate "v is a string" can be expressed as lb(v)=1, and string equality can be represented as vector equality. But such basic operations as string concatenation, or the deletion or insertion of elements in a string (except at the high end) are not representable without quantifiers. If in our formulation of vectors in BITZV we were to include some function such as "equal as strings" or vector concatenation, or even Dijkstra's vector shift operator (in which the vector shift(v,i) is similar to v except that all indices together with the lb and ub are offset by i [Dijkstra 76]), then at least some string operations could be represented. Unfortunately, adding such functions to BITZV seems to complicate severely the Type Reduction of BITZV to BITZ, if it remains possible at all.

We have already noted that it is possible to represent the function select(v,i) to select the i\textsuperscript{th} largest element of vector v for any constant i. Unfortunately it is apparently not possible to represent the select function with an arbitrary term in the second argument. It is in particular not possible to represent
\[
\text{median}(v) = \text{select}(v, \text{len}(v)/2)
\]

Similarly, although it is possible to write the term \( z < t \) to represent the number of occurrences of \( t \) in \( z \), it is apparently not possible to represent the function \( \text{mode}(z) \) which is the minimum \( t \) with maximum multiplicity in \( z \). Nor is it possible to represent the function \( \text{less}(z, t) \) which returns a count of the number of those elements less than or equal to \( t \) having non-zero multiplicity in \( z \).

We remarked in the last subsection that the algebra of stacks is representable in BITZV. This is true, but it is not very interesting, because it is apparently the case that no interesting program involving stacks can be verified within this algebra. We also should note that although we can represent the algebra of stacks in BITZV, I know of no way to represent the algebra of queues. Such a representation seems to require the "equal as strings" operation on vectors to represent the \( \text{queue} \) operation. It seems always to be necessary to assert something about the relation between a stack and some other data structure, such as a tree, or a string, or another stack, and invariably the thing asserted is outside the scope of BITZV.

Since BITZV was defined with comparison-sorting programs in mind, it is perhaps not surprising that it is not expressive enough to express some of the notions we have talked about so far, because they have little direct bearing on comparison-sorting. However, there are plenty of notions which are relevant to comparison sorting but which are apparently not expressible in BITZV. Perhaps the simplest example is the notion of reverse order. The predicate \( \text{ord} \) asserts that a vector is in non-decreasing order, but the predicate asserting non-increasing order apparently cannot be encoded in BITZV (without a quantifier). Thus, a program to sort in reverse order according to whatever interpretation \( < \) is given, cannot be verified in BITZV. (It would be simple to add such a predicate, but at the moment it is not included.)

Algorithms which sort by manipulating a vector of indices pointing into the vector-to-be-sorted are out of bounds for BITZV. This is because the data type
vector-of-integer is not available to represent the index vector.

Many sophisticated comparison sorting methods such as Heapsort [Knuth 73] are also off limits. In the case of Heapsort, the reason is that the property "v[i..j] is a heap" is apparently not representable in BITZV. And, of course, any sorting program making essential use of a tree data structure, or of operations other than comparisons on elements of Type T, cannot be verified wholly within BITZV.

4.4 Related Work

The problem attacked in this thesis is that of finding a reduction procedure or decision procedure for BITZV. At least two closely related studies have been published and we are now in a position to compare them with the results presented here.

4.4.1 Mateti's work

In his thesis [Mateti 76] and in a subsequent report [Mateti 78] Mateti describes a decision procedure for a language that, though restricted, is capable of expressing the verification conditions for the ordering properties of certain sorting programs. Although his method is completely different, his results have the same character as ours.

Mateti describes a simple programming language and a simple assertion language, and gives a procedure for deciding the validity of any formula that can arise as a verification condition for programs with assertions in his languages. His programming language contains the usual sequential control structures, and four data types corresponding to our B, I, T and V. However, he imposes severe restrictions on programs in the language. Each program is presumed to manipulate segments of exactly one vector variable named X. Elements of the vector cannot be assigned to; instead, the only side-effects permitted on X are the exchange of two elements and the circular shifting of a segment of X by one unit (which Mateti calls "insert").

For his data type PTR (which corresponds to our type I and is used for pointing into the
vector X) Maleti permits only the addition and subtraction of constants. Thus, all PTR-terms are of the form $p + k$ (or simply $p$ or $k$) for variable $p$ and integer constant $k$. (Actually, he restricts $k$ to be $+1$ or $-1$, but the restriction is of no consequence.) General addition of two terms, as well as multiplication or other operations, are not permitted.

The assertion language defined by Maleti is also highly restricted. He permits as assertions only formulae that are propositional combinations of atoms (without quantifiers) which are equivalent to the following forms in BITZV notation:

$$i_1 < i_2, i_1 = i_2, \text{etc.}$$
$$t_1 < t_2, t_1 = t_2, \text{etc.}$$
$$\text{ord}(X[i_1..i_2])$$
$$\max(X[i_1..i_2]) \leq t$$
$$t \leq \min(X[i_1..i_2])$$
$$\max(X[i_1..i_2]) \leq \min(X[i_3..i_4])$$

The terms $i_1, i_2, i_3$ and $i_4$ represent PTR-terms in his language, and thus are of the form $p + k$, or simply $k$. The T-terms (elements of type KEY in his terminology) are either variables, or of the form $X[i]$ (for some PTR-term $i$) where $X$ is the unique vector permitted in his language. Note that max and min functions are not part of Maleti's assertion language. His language cannot, for example, express the predicate

$$\min(X[i_1..i_2]) \leq \max(X[i_3..i_4]).$$

The above expressions in BITZV correspond to primitives in his language. Maleti's program and assertion languages are clearly quite limited when compared to the expressibility of BITZV. The major distinction is his inability to express anything in his language about the permutation properties of programs that manipulate arrays. In fact, his programming language cannot express any algorithm which does not conserve the elements of his vector $X$. Other differences include the lack, in his work, of a vector assignment function, the lack of genuine max and min functions for vector segments, and the restriction to a single vector.

With the exception of his "Insert" function, which is apparently not representable in BITZV, Maleti's system is completely representable in BITZV. ("Insert" appears in his thesis, but not in the subsequent report.)
The algorithm devised by Mateti for deciding the validity of verification conditions is quite different from the method we use. Mateti treats a formula as though it were divided into a hypothesis and a conclusion part. He then shows how to construct a graph which represents a "most general model" for the hypothesis part. This "most general model" can then be examined to see whether or not the conclusion holds in it; if so, the formula is valid, and if not, it is not valid. He does not employ any notion resembling Type Reduction in his algorithm.

Although the methods in Mateti's thesis and this one are very different, there is one significant point of agreement. Mateti makes important use of the idea of partitioning his vector X into nonoverlapping subvectors, and doing so in all ways allowed by the constraints in the formula. This is essentially the same idea as the case splitting used to remove what we call "aliasing" in our algorithms of Chapters 6 and 7.

4.4.2 Suzuki and Jefferson

A direct precursor to the work in this thesis is that of Suzuki and Jefferson, who showed how the permutation property of many sorting programs could be automatically verified ([Suzuki 80]. Once again, the approach was similar to, but less comprehensive than, the one taken in this thesis.

Suzuki and Jefferson treat programs involving three data types: Boolean, Integer, and Vector-of-integer. Since they do not distinguish between Type-I and Type-T, as is done here, their problem is somewhat different. On the one hand the problem is made easier in their work because the special properties of the integer total-order can be used, and the issue of having to devise an algorithm capable of handling any total order type (e.g. finite types, or uncountable types, or the order type of the rationals) is circumvented. On the other hand, their problem is more complicated by the fact that terms such as $v_1[v_2[i]]$, involving nested vector access, are permitted. Such terms cannot occur in BITZV because the subterm $v_2[i]$ is of Type-T and cannot be used as an index into vector $v_1$. Suzuki and
Jefferson, unlike Mateti, permit unrestricted array assignments and any number of array variables.

Although Suzuki and Jefferson's work was intended to address the issue of verifying the permutation properties of sorting programs, their assertion language was fairly limited. An inductive assertion was constrained to be of the form

\[ \text{wff} \land \text{perm}(v, v_0). \]

The first conjunct was required to be a propositional combination of integer atomic formulae involving integer variables and constants, the addition and subtraction functions, the vector access function \( v[i] \), and the = and < predicates. In the second conjunct, \( v \) was required to be a vector variable from the program, while \( v_0 \) was a vector variable not occurring in the program. The predicate \( \text{perm}(v, v_0) \) meant that vector \( v \) is a permutation of vector \( v_0 \). As with Mateti's work and this thesis, no quantifiers were permitted.

The severity of the restrictions in their assertion language should be obvious. The \( \text{perm} \) predicate can occur only once in an inductive assertion, and then only as a top-level conjunct; and one of its arguments must be a variable not occurring in the program. Under such restrictions it is possible to transform any verification condition into a form containing at most one occurrence of the \( \text{perm} \) predicate, and that simplification makes the decision algorithm reasonably straightforward. Yet despite the severe restrictions on the form of assertions, their language is sufficiently rich to express the assertions necessary to verify the papers, though it appears somewhat ad hoc now due to the restricted nature of the problem, is in fact related to the Type Reduction procedure given in Chapter 6. In their procedure they systematically removed all occurrences of functions and predicates involving vectors: first vector assignment, then \( \text{perm} \), and finally vector access, leaving formulae that contain only integer variables and integer functions symbols. We might now describe it as a Type Reduction from \( \text{BIV} \) to \( \text{B} \), although the \( \text{B}, \text{I} \) and \( \text{V} \) types of their papers are not identical to the \( \text{B}, \text{I} \) and \( \text{V} \) of this thesis.
4.4.3 Combination of Permutation and Ordering Predicates

In the previous two subsections we have seen how Mateti's work was oriented toward verifying the ordering property of simple sorting programs, while Suzuki and Jefferson's work was directed toward verifying the permutation property. One way to view this thesis is as a combination and extension of those two results. For the most part they are both subsumed (and then some) by BITZV.

The results in this thesis do not simply represent the sum of those previous results, but something more akin to the product. We can now automatically verify both the ordering and the permutation properties of simple sorting and merging programs, and not only separately, but simultaneously. Even formulae in which the permutation and ordering properties interact can now be decided. Furthermore, the results in this thesis are much cleaner than those of Mateti or Suzuki and Jefferson. There are no arbitrary restrictions on the forms that assertions in BITZV may take, nor on the forms of the programs to which the assertions apply. Finally, while the earlier results were ad hoc and fragile, we present in this thesis the beginning of a methodology for organizing theorem provers for program verification in a comprehensible and robust form.

4.4.4 Reynolds' work

In a recent paper Reynolds described an interesting language/notation which is suitable for reasoning about vectors and which is in some ways similar to our language BITZV ([Reynolds 79]). The most original aspect of his work is his use of partitioned box-diagrams to represent intervals of the integers and also to represent order relations among integer terms. His diagrams, which we cannot represent here typographically, depart significantly from the fixed arity function and predicate paradigm so familiar from the predicate calculus. And while the relations expressible by such diagrams are also expressible in BITZV using conjunctions of integer equalities and inequalities, they are much easier to deal with in Reynolds' notation, both intuitively and (probably) in a mechanical deduction.
Besides introducing the partitioned-box notation, however, Reynolds' paper is mainly devoted to cataloging and systematizing a variety of concepts concerning vectors. There is, not surprisingly, considerable overlap between the notions he defines in his language and those of BITZV. In the remainder of this subsection we will compare the two notations, concentrating mainly on the differences.

At first glance Reynolds' model of a vector seems almost identical to ours. (Both were drawn from [Dijkstra 76].) We both view vectors basically as mapping from the integers into the carrier of some (other) type. Both of us have functions returning the lower and upper bounds of that interval. (Reynolds' lower V and upper V correspond to our lb(V) and ub(V).)

But a close examination reveals two important differences in our respective concepts of vectors.

The first difference concerns errors. Reynolds views a vector X declared using the following declarer

\[ T \text{ array X(a:b)} \]

as a total function from the integer interval [a..b] into the carrier of type \( T \) [Reynolds 79p.292]. By contrast, we view the vector X declared the same way as a partial function from all of \( \mathbb{Z} \) to \( T \) which is guaranteed to be "defined" on the interval [a..b] and "undefined" outside that interval. The difference is that in our case the meaning of an array bounds error, e.g. \( X[a-3] \) (or \( X(a-3) \) in Reynolds' notation) is perfectly well-defined, namely \( E_T \), whereas Reynolds gives it no meaning at all. In fact, Reynolds hardly mentions array bounds errors in his paper, although in the definition of array assignment he remarks that "\([x, i, e]\)"-which we denote by \( <x, i, e> \)-"is defined when \( i \in \text{dom}X \)." Presumably this implies that \( [x, i, e] \) is "undefined" when \( i \notin \text{dom}X \), but Reynolds never describes how to deal with expressions containing "undefined" terms, and he recognizes no vector value corresponding to our error vector \( E_{V} \). These differences can all be summarized by the observation that BITZV is formalized in PFC3, a three-valued calculus of partial functions, while Reynolds' is based on the semantics of the two-valued calculus of total functions.
There is another difference between BITZV and Reynolds' language concerning the subject of degenerate or empty vectors. In BITZV we recognize an infinite number of distinct empty vectors, i.e. those with domain intervals \([-1..-2], [0..-1], [1..0], [2..1], \) etc. A term such as \(v[i..j]\) where \(j+1<i\) takes the error value \(E_v\). By contrast, in Reynolds' paper there is exactly one empty vector whose bounds are two "unspecified" integers \(l_0\) and \(u_0\) where \(u_0 < l_0\). Any term \(v[i_1..i_2]\) (our notation, since we unfortunately cannot type his) such that \(i_2 < i_1\) takes the special empty vector as a value. When \(i_2 + 1 < i_1\) Reynolds calls the interval \([i_1..i_2]\) irregular, but he still assigns the term \(v[i_1..i_2]\) the empty vector as its value. I personally do not see the virtue of these notions of a single empty vector and of irregular intervals, but Reynolds' view is certainly a consistent alternative.

In the cases of non-error, nonempty vector values our concept is identical to Reynolds'. But the list of functions and predicates in Reynolds' language is richer in most respects than BITZV. In fact, all but three significant \(V\)-relevant operators of BITZV are directly expressible in Reynolds' language. Those three operators are \(\min_v, \max_v\) and \(\{^0\}_v\). Of these, \(\min\) and \(\max\) are obvious extensions and were perhaps left out of Reynolds' paper for space reasons. (Obviously he couldn't list every concept relevant to vectors.) Only the function \(\{^0\}_v\) seems to represent a notion touching areas covered by Reynolds, and which might offer an improvement. In Reynolds' language the term \(\{v\}\) means the set of elements in vector \(v\), and the relation \(v_1 \sim v_2\) is the equivalence relation meaning the set of elements in \(v_1\) is the same as the set of elements in \(v_2\). Neither of these notions is expressible in BITZV. But curiously Reynolds does not include any operator in his language comparable to our \(\{^0\}_v\)-function which returns the multiset (zset) of elements occurring in the vector. Such an operator, together with a set of multiset (or zset) functions such as + and -, would seem to make some statements in his language a little clearer.

Reynolds' language treats several other concepts, particularly vector concatenation and other orderings besides nonincreasing order, that are outside of BITZV's expressive range. It would be an exciting thing if someone were to extend the work of this thesis and give Type Reduction procedures to help decide the truth of formulae in a language as rich as Reynolds'.
4.4.4

I conjecture that, with the possible exclusion of the concatenation operator and the shift equivalence relation from Reynold’s language, such a procedure exists.

4.4.5 Nelson and Oppen’s work

In a recent important paper [Nelson 79] Greg Nelson and Derek Oppen published an important algorithm for constructing decision procedures (satisfiability procedures) for large theories out of decision procedures (satisfiability procedures) for small ones. The goal and spirit of the work in this thesis is exactly in line with their work, so we should take the time to make a comparison.

What Nelson and Oppen showed was this. Suppose we have two unquantified theories $T_1$ and $T_2$ which are formalized in the disjoint formal languages $L_1 = \{P_1, P_2, \ldots, f_1, f_2, \ldots\}$ and $L_2 = \{Q_1, Q_2, \ldots, g_1, g_2, \ldots\}$ and are generated by sets of unquantified axioms $A_1$ and $A_2$ respectively. Thus, $T_1$ ($T_2$) is the set of all unquantified formulae in $L_1$ ($L_2$) which are consequences of the axioms in $A_1$ ($A_2$). Let us further consider the free product theory $T_3 = T_1 \ast T_2$, which is defined to be the set of all unquantified wffs from the language $L_3 = L_1 \cup L_2$ which are consequences of the axiom set $A_3 = A_1 \cup A_2$. Oppen and Nelson showed that decision procedures for $T_1$ and $T_2$ can be combined (constructively) into a decision procedure for $T_3$.

There is a resemblance between their procedure and the method of Type Reduction used in this thesis, but it is not a very close resemblance. If the language BITZV were decomposable into a free product in the sense used in Nelson and Oppen’s work, then their result would apply; but the fact is that BITZV cannot be factored in that way, and their result does not give much leverage on the decision problem for BITZV. For example, it might appear at first glance that BITZV can be decomposed into the product of BI, BT, BZ and BV, or perhaps simply the product of BITZ and BITV. But such decompositions of BITZV are not factorings into theories which, when composed by free product, yield back BITZV. Either they leave out some function or predicate symbols (e.g. in which theory, BITZ or BITV would the function symbol $\{^o\}$ be placed?) or they require new axioms to be added in addition to
those which specify the small theories. I can see no way to apply Nelson and Oppen's algorithm to simplify the decision problem for BITZV.

4.5 An example: Binary Insertion Sort

We now give an example of a program, together with its specifications and loop invariants, written in terms of the data types and function symbols defined in BITZV. The program will be a Binary Insertion Sort, which has been chosen to illustrate the fairly wide range of expressions possible in BITZV. We will give assertions sufficiently strong to prove the firm correctness of both the ordering and permutation properties. Since this program contains an embedded binary search, it should serve as an indication of how to write and specify the correctness of a Binary Search program in BITZV as well. Here is the program.

(1) \textbf{var} \quad A_0 : V, \quad \text{! Type declarations}
\quad A : V,
\quad i, j, \ lo, \ hi, \ split : I,
\quad temp : T;

(2) \textbf{precondition} \quad A = A_0;
\quad \text{! The main loop cycles from } i=lb(A)+1 \text{ to } i=ub(A),
\quad \text{! each time inserting } A[i] \text{ into its proper place in}
\quad \text{! the already-sorted part } A[lb(A) \ldots i-1]

(3) \quad i \leftarrow lb(A)+1;

(4) \textbf{invariant} \quad \{A\} = \{A_0\} \land lb(A) = lb(A_0) \land lb(A) \prec i \land i \preceq ub(A)+2 \land
\quad \left(\text{ord}(A[lb(A) \ldots i-1]) \lor i=ub(A)+2\right)

(5) \textbf{while } i \preceq \text{ub}(A) \text{ do}
\quad \textbf{begin}
\quad \text{! Binary-search } A[lb(A) \ldots i-1] \text{ to find the right place}
\quad \text{! to insert } A[i]. \text{ At the end we will have}
\quad \text{! low = hi = the final position for } A[i].

(6) \quad \textbf{begin}
\quad \text{low} \leftarrow lb(A);
\quad \text{hi} \leftarrow i;

(8) \textbf{invariant} \quad \{A\} = \{A_0\} \land lb(A) = lb(A_0) \land \text{ord}(A[lb(A) \ldots i-1]) \land
\quad \left(\text{lb}(A) \leq \text{low} \land \text{low} \leq \text{hi} \land \text{hi} \leq i \land i \preceq \text{ub}(A) \land
\quad \left(A[\text{low}-1] \leq A[i] \lor \text{low} = \text{lb}(A)\right) \land (A[i] < A[hi] \lor \text{hi} = i\right)

(9) \textbf{while } \text{low} < \text{hi} \text{ do}
begin
  split ← (low+hi)/2;
  if A[split]≤A[i]
    then
      low ← split+1
    else
      hi ← split
  end;

! Move each element of A[hi..i-1] one location ! to the right so that A[i] can be inserted ! at A[hi]

(14) temp ← A[i];
(15) j ← i;

(16) invariant {A[lb(A)..j-1]} + {temp} + {A[j..ub(A)]} = {A₀} ∧
     lb(A)=lb(A₀) ∧ ord(A[lb(A)..j-1]) ∧ ord(A[j..i]) ∧
     hi ≤ j ∧ j ≤ i ∧ A[j..i] ∧

while j>hi do
  begin
    A[j] ← A[j-1];
    j ← j-1
  end;

! Insert the former A[i] (now temp) into its ! proper place.

(20) A[hi] ← temp;

! Prepare to work with next element of A
(21) i ← i+1
end

(22) postcondition {A}={A₀} ∧ lb(A)=lb(A₀) ∧ ord(A)

To be simple, we will assume that we have a vanilla programming language with only assignment statements, semicolon connectives, if-then-else connectives, while-do connectives, and begin-end statement grouping. Comments are preceded by "!". The data types available, both in the program and in assertions, are B, I, T, Z and V, and the function and predicate symbols in the language are those of BITZV.
The only kind of assignment we permit is from an expression (of any type) to a variable of the same type. We will, however, permit the notation

\[ v[i] \leftarrow t; \]

to be an abbreviation for

\[ v \leftarrow <v, i, t>; \]

according to conventional program notation.

The invariant statements inserted into the program are loop invariants applying to the immediately following while-do loop. It should be understood that the invariant assertion must be true (i.e. neither false nor error) at the top of each loop iteration before the test is evaluated.

All functions are to be interpreted in their three-valued senses. In particular, the \( \land \) and \( \lor \) functions in the assertions are the strong conjunction and strong disjunction functions.

The input to the program is a single vector, \( A \). We do not need other inputs to describe the bounds or length of the vector because these are part of the value of \( A \), and are accessible through the \( \text{lb}, \text{ub} \) and \( \text{len} \) functions.

The precondition \( A = A_0 \), asserts that vector \( A \) has initially the same value as \( A_0 \). The \( V \)-variable \( A_0 \) does not occur in the program at all; we are merely following the usual practice of "naming" the initial value of an input variable such as \( A \), so that in later assertions we can refer to that initial value. Since \( A \) and \( A_0 \) are both variables, they cannot take error values, and the precondition assertion could just as well have been written \( A = A_0 \). Aside from the requirement of the logic that \( A \) not have the error value, there are no other restrictions on \( A \). It may have any length or bounds, and may even be an empty vector. Notice here the essential use of the vector equality predicate; if BITZV were not equipped with equality predicates for all types, then it could not express the specifications for programs where the postcondition must refer to initial values.
The postcondition is
\[ \{A\} = \{A_0\} \land \text{lb}(A) = \text{lb}(A_0) \land \text{ord}(A). \]
The first conjunct asserts that the zset of elements in A at termination is the same as the
zset of elements in A initially. In other words, A at termination is a permutation of its original
value.

The second conjunct asserts that the lower bound of A is the same as its original lower
bound. Together with the first conjunct, this implies that \( \text{ub}(A) = \text{ub}(A_0) \) is true also. The point
of this second conjunct is that not only is the collection of elements of A unchanged, but also
the bounds of A are unchanged. The only aspect of A that may be changed is the order of
the elements.

The third conjunct, of course, asserts that A is ordered at termination.

The main loop of the sort is based on an iteration of i from \( \text{lb}(A)+1 \) to \( \text{ub}(A) \). In each
iteration the program binary-searches the subvector \( A[\text{lb}(A)..i-1] \) (which is already sorted) to
find the proper (sorted) place for \( A[i] \). When it finds the place it marks it by the variable \( hi \),
moves all of the elements of \( A[hi..i-1] \) one place to the right, and stores the former \( A[i] \) into
\( A[hi] \).

We have written in line (4) the following loop invariant for the main while-do loop of the
program
\[ (4) \quad \{A\} = \{A_0\} \land \text{lb}(A) = \text{lb}(A_0) \land \text{lb}(A) \leq i \land i \leq \text{ub}(A)+2 \land \\
(\text{ord}(A[\text{lb}(A)..i-1]) \lor i = \text{ub}(A)+2). \]
This assertion is slightly more complicated than might seem necessary because of a
combination of two circumstances. First, in line (3) we start the loop counter at \( \text{lb}(A)+1 \), a
slight optimization taking advantage of the fact that both empty vectors and unit-length
vectors are already sorted. Second, our loop invariant applies to the point before the loop
test is made, rather than after. If either of these circumstances were changed, the invariant
could be slightly simpler.
For those practiced in reading BITZV and loop invariants most of the content of this assertion should be self-explanatory. One remark deserves to be made, however. In the case that \( A \) is an empty vector, and thus \( \text{lb}(A) = \text{ub}(A) + 1 = i - 1 \), the term \( \text{ord}(A[\text{lb}(A)..i-1]) \) evaluates to `error` in the loop invariant because of an array bounds violation for trying to select a length-one subvector out of the length-zero vector \( A \). However, the other half of the disjunction, \( i = \text{ub}(A) + 2 \), is `true` under those same circumstances. Hence, the truth value of the disjunction is `error` \( \lor \) `true` which is `true`. This is an example of the significance of three-valued disjunction in an assertion.

We will skip discussion of the binary search loop, lines (6)-(13), and proceed to the insertion loop in lines (14)-(20). In this loop we move each element of \( A[\text{hi}..i-1] \) to the right one space, preparing to insert (the old value of) \( A[i] \) into \( A[\text{hi}] \). At the point that the loop invariant is asserted the value of \( A[i] \) has been copied into the variable `temp` before \( A[i] \) is clobbered by \( A[i-1] \).

All during this loop, the only copy of the original value of \( A[i] \) is in the variable `temp`. Hence the simple permutation property \( \{A\} = \{A_0\} \) does not hold during the execution of the loop. However, it is true that the elements in \( A[\text{lb}(A)..j-1] \) and \( A[j+1..\text{ub}(A)] \) together with the value in `temp` form a permutation of \( A_0 \). The first conjunct of the invariant line (16),
\[
\{A[\text{lb}(A)..j-1]\} + \{\text{temp}\} + \{A[j+1..\text{ub}(A)]\} = \{A_0\},
\]
shows how to express in BITZV the permutation property under such circumstances. It illustrates the use of the vector restriction function \( \circ[\circ..\circ] \), the zset-of-a-vector function \( \{\circ\}_v \), the singleton zset function \( \{\circ\}_T \) (applied to `temp`), the zset addition function \( +_2 \), and the zset equality functions \( =_2 \). As usual, we have dropped subscripts for readability. An equivalent conjunct could be written differently, using the vector assignment operator. We could write
\[
\{<A, j, \text{temp}>\} = \{A_0\},
\]
meaning "after assigning \( A[j] \leftarrow \text{temp} \) the zset of elements in \( A \) would be the same as the zset of elements in \( A_0 \)."
The next-to-last conjunct of (16) contains a disjunction for reasons similar to that discussed in line (4). In the case that \( hi = \text{lb}(A) \) (which occurs when temp is smaller than any element in the already-ordered part of the vector), the term \( A[hi-1] \) would evaluate to \( E_T \) as an array bounds error. But the truth value of \( A[hi-1] \leq \text{temp} \lor hi = \text{lb}(A) \) is \( \text{error} \lor \text{true} \) which is \textbf{true}. The assertion, therefore, can be \textbf{true} even if error values occur in evaluating it.

It should be noted that some of the conjuncts in line 16 are redundant. For example, if the entire assertion is \textbf{true}, then the conjunct \( \text{ord}(A[j+1..i]) \) must be \textbf{true}. But this implies that \( i \leq \text{sub}(A) \), because if not, \( \text{ord}(A[j+1..i]) \) would have the value \textbf{error}. Therefore, in the presence of the conjunct \( \text{ord}(A[j+1..i]) \) the conjunct \( i \leq \text{sub}(A) \) is redundant. This fact can be expressed in the identity

\[
\forall \text{ord}(A[j+1..i]) \Rightarrow i \leq \text{sub}(A)
\]

where \( \Rightarrow \) has its 3-valued interpretation. For similar reason \( j < i \) is redundant. We choose to include these redundant conjuncts simply to reduce the chance of human error.

It would be time- and space-consuming to construct and prove the lengthy verification conditions for this annotated program. It is the case, however, that barring some typographical error, the assertions are sufficiently strong to prove firm correctness. Even without the proof, however, the essential point of the expressiveness of BITZV in the domain of searching and sorting should have been made in this example.
5. Type Reduction and Related Concepts

In this short chapter we describe the methods of Type Reduction and the related reduction methods, Type Separation and Truth Value Reduction, and show how they are composed into a decision procedure for the language BITZV. BITZV and its major sublanguages are related according to Figure 5-1.

![Figure 5-1: BITZV and its sublanguages](image)

Here, of course, BITZV is the language defined in Chapter 4. BITZ is the set of all formulae in BITZV which contain no Type-V terms. Similarly, BIT is the set of all formulae in BITZ which contain no Z-terms, and BI and BT are the sets of all formulae in BIT which contain no T-terms and no I-terms respectively.

The general structure of the decision procedure for BITZV is outlined in Figure 5-2. From this diagram we see that there are four stages of reduction in the decision procedure for BITZV. The first two stages are Type Reductions, and they involve by far the largest amount of mathematical work represented in Figure 5-2. In the next section we explain the general
Figure 5-2: Decision Procedure for BITZV

BITZV

Type Reduction

BITZ

Type Reduction

BIT

Type Separation

BI

BT

BI₂

BT₂

Unquantified Theory of (Presburger) Arithmetic

Unquantified Theory of Total Order
concept of Type Reduction, but the details of its application to the case in point requires all of Chapters 6 and 7 to explain. The result of these two stages is a method by which a decision procedure for BIT can be converted to a decision procedure for BITZV.

The third reduction stage is called a Type Separation. The concept Type Separation is explained in Section 5.3, and in the same section we give the details of the process for the language BIT. For BIT the Type Separation is essentially trivial, although it need not be in general. The result of the Type Separation is a construction whereby two decision procedures for BI and BT can be combined into a decision procedure for BIT.

The final parallel reduction stages we are calling Truth Value reductions. Since BI and BT are both defined in the three-valued logic PFC3, another reduction stage is required to make contact with the well-known unquantified theories of arithmetic and total order (in the usual two-valued calculus FC2). The result of the Truth Value reductions is a construction whereby decision procedures for the ordinary two-valued unquantified theories of total order and arithmetic can be transformed into decision procedures for BI and BT. This is discussed in Section 5.4.

After four stages of reduction we are left only with the problem of finding decision procedures for the theories of unquantified total order. These decision problems are discussed in Section 5.5.

5.1 Definition of Type Reduction

Even simple programs, such as vector merging and sorting programs, use variables and terms of several different types. In an ordinary merge program we will see terms of type Integer used as indices into vectors, terms of type T (the type of the elements being merged, used in comparisons), terms of type Vector-of-T used as the source and destination vectors, and terms of type Boolean used in conditionals and loops. In assertions and verification conditions other types may occur as well. For example, in the last chapter we showed that in the specifications for a merge program, values of type zset-of-T may be useful even though
that type does not explicitly occur in the program.

Some of the types will be primitive scalar types and independent of one another: Integer, T, Boolean. Others will be nonprimitive, having domains which are sum, product or function spaces over the primitive types. Thus, the domain of type \texttt{record(a:T, b:Integer)} is (or is isomorphic to) the cross product of the domains of type T and type Integer. In the case of BITZV, both Type-V and Type-Z are basically function spaces. A vector-of-T is basically two integers (the bounds) and a finite partial map from \( \mathbb{Z} \) onto the domain of T. A zset-of-T is basically a finite partial map from the domain of T onto \( \mathbb{Z} \).

For mixed systems of types, especially those containing both nonprimitive and primitive types, decision procedures (or relative decision procedures) can often be organized by a method called Type Reduction. In general, Type Reduction is any method of reducing the decision problem for a system of types, functions and predicates, to the decision problem for some proper subsystem thereof. However, for purposes of this thesis we will use a narrower definition.

\textbf{Definition 1}: Let \( L(B, T_2, \ldots, T_n) \) be a formal language having types Boolean and \( T_2 \ldots T_n \). Let I be a class of interpretations for the function symbols of the language (including the predicate symbols, which represent Boolean-valued functions). A Type Reduction procedure is a procedure which takes an arbitrary formula \( w \in L(B, T_2, \ldots, T_n) \) as input and constructs a finite set of wffs \( \Omega = \{ w_1, \ldots, w_m \} \subseteq L(B, T_2, \ldots, T_n) \) as output such that

1. No \( w_i \in \Omega \) contains any terms of type \( T_n \), i.e. \( \Omega \subseteq L(B, T_2, \ldots, T_{n-1}) \), and

2. The universal closure of \( w \) is true in all interpretations of class I if and only if the universal closures of all \( w_i \in \Omega \) are true in all interpretations of class I. \( \square \)

A Type Reduction procedure for language \( L(B, T_2, \ldots, T_n) \) over interpretation class I serves as a decision procedure relative to a decision procedure (or oracle) for truth in the sublanguage \( L(B, T_2, \ldots, T_{n-1}) \) over the same interpretation class I. Conditions 1 and 2 of the definition guarantee that a decision oracle for \( L(B, T_2, \ldots, T_{n-1}) \) could be applied to each of the finite number of wffs of \( \Omega \) to produce an algorithm for deciding truth in \( L(B, T_2, \ldots, T_n) \).
We do not specify in the definition whether or not language \( L(B, T_2, \ldots, T_n) \) permits quantification, or which types of variables are quantifiable. Whatever conditions of this nature apply to the input language also apply, we presume, to the output language, \( L(B, T_2, \ldots, T_{n-1}) \).

The definition of Type Reduction also does not specify whether the semantics of the language is to be formalized in FC2 or PFC3. Our intention is that either case may be permitted, with the meaning of the word "interpretation" adjusted accordingly.

In the definition we consider truth for a class of interpretations rather than for a single interpretation because, as is often the case in mathematics, a result may hold for a wide class of models (e.g. all groups, all partial orders) rather than just for a particular model.

In a language with \( n \) types, Type Reduction can in principle be used to eliminate any of the types except Boolean. However, it seems to be most fruitful to apply it in such a way that the eliminated type, \( T_n \), is one of the nonprimitive types. When applied this way, the phrase Type Reduction is a triple-entendre. Its primary meaning, of course, is in the sense of problem-reduction. But we also notice that the number of types in question is reduced from \( n \) to \( n-1 \), and also that the "level" or "order" of the types is reduced as well when we eliminate a higher-order type such as \( V \) or \( Z \).

The kind of problem reduction represented by Type Reduction is a particularly simple kind, namely \( w \) is valid if and only if all of the \( w_i \) of \( \Omega \) are valid. Thus, the truth value of \( \mathcal{I}Ew \) is a simple conjunction of the truth values of \( \mathcal{I}Ew_1, \mathcal{I}Ew_2, \ldots, \mathcal{I}Ew_n \). This is a particularly simple kind of truth-functional reducibility, and one can easily imagine a more elaborate definition modelled on Turing-reducibility [Rogers 67]. But the current definition has proved satisfactory at least for the theory BITZV, and so we adopt it as is for purposes of this thesis.
5.2 Type Reduction: BITZV → BITZ → BIT

The language of BITZV is a five-typed unquantified language. In our current notation it would be denoted \( L(B, I, T, Z, V) \). As described in Chapter 4, a formula of BITZV is considered true (or valid) if it is true in any interpretation in which the domains \( B \) and \( I \) are the Booleans and the integers respectively, the domain of \( T \) is a totally ordered set, and the domains of \( Z \) and \( V \) are the finite zsets-of-\( T \) and vectors-of-\( T \) respectively. The function symbols of BITZV must be interpreted in accordance with the specifications of Chapter 4. Because type \( T \) is interpreted as any totally ordered domain, BITZV refers to a class of models, not just a single model. This is why truth in a class of interpretations was specified in the definition of Type Reduction, rather than truth in a single interpretation.

The decision procedure for BITZV (or relative decision procedure if multiplication is considered) starts with two Type Reductions. The first stage is a Type Reduction from BITZV to BITZ (i.e., from \( L(B, I, T, Z, V) \) to \( L(B, I, T, Z) \)), in which all terms of type \( V \) are removed. The second stage is another Type Reduction from BITZ to BIT (i.e., from \( L(B, I, T, Z) \) to \( L(B, I, T) \)). These two Type Reduction procedures are given in Chapters 6 and 7, and they ensure that any decision procedure for BIT can be (effectively) converted into one for the full system BITZV.

The Type Reduction procedures in Chapters 6 and 7 have an extremely important robustness property which should be emphasized. Consider first the Type Reduction from BITZV to BITZ given in Chapter 6. Suppose we were to extend the language BITZV with new functions not involving Type \( V \). We might, for example, extend it by adding \( \log_2 \) as an integer function, or perhaps exponentiation. Or we might be more ambitious and add an entire collection of new functions along with a new type, e.g., type Set-of-\( T \) with functions \( 0, u, n, -, \subseteq, = \) and maybe a function \( \text{set} \) which maps a Zset to its underlying set. The surprising fact is that we may extend BITZV by adding any collection of functions and types, so long as the added functions do not include Type \( V \) as arguments or results, and the algorithm given in Chapter 6 will still work without change! 
If the algorithm for reducing BITZV to BITZ is applied to a formula w which contains additional types and/or function symbols, the only effect it will have is that the set \( \Omega \) (which is output by the algorithm) will probably contain wffs which will have occurrences of the added function symbols or types. But it will still be the case that all terms of Type V are removed, and that w is valid if and only if all wffs in \( \Omega \) are valid. Therefore, we can view the algorithm of Chapter 6 as a reduction not just from BITZV to BITZ, but from BITZV* to BITZ*, where BITZV* is any extension of the language BITZV which does not add any new function symbols having Type V as an argument or a result, and where BITZ* is the corresponding extension of the language BITZ.

A similar comment can be made about the Type Reduction of BITZ to BIT given in Chapter 7. If the language BITZ were extended to BITZ* by adding any new function symbols which do not have Type Z as an argument or a result, including function symbols involving new types, then the procedure given in Chapter 7 applies without change as a Type Reduction from BITZ* to BIT*.

As an interesting example, we might extend the language BITZ to include Type V and all of the function symbols removed by the first Type Reduction excepting only the function \( \{\cdot\} : V \to Z \). The Type Reduction in Chapter 7 would then apply perfectly well, but it would output \( \Omega \subseteq \text{BITZ} \) rather than \( \Omega \subseteq \text{BIT} \).

This property of our Type Reductions is not part of the definition of Type Reduction; it is simply a property of the algorithms as we have written them. The list of function symbols of the language BITZV which involve Type V is maximal with respect to the algorithm of Chapter 6, in the sense that any additional functions involving Type V would require at least token changes to the algorithm, and probably substantial changes. On the other hand, the list of function symbols in BITZV which do not involve Type V is minimal with respect to our algorithm in the sense that this list may be arbitrarily extended and the algorithm will apply without change. We therefore call this property the minimax property of our Type Reductions. Because the minimax property is so important and because it seems to fall
naturally out of the methodology, I may in future work require it as part of the definition of Type Reduction.

5.3 Type Separation: BIT → (BI, BT)

After two stages of Type Reduction we are left with the theory BIT. This theory is decidable if and only if the theories BI and BT are decidable, as the following argument shows.

Suppose we take an arbitrary formula w ∈ BIT. First, we remove all occurrences of the conditional functions for types I and T by using equivalence transformations such as

\[(b | i_1 | i_2 | i_3) \prec w \Rightarrow \]
\[w \leftarrow (b | REPL(w, (b | i_1 | i_2 | i_3) \Rightarrow i_1))
\| REPL(w, (b | i_1 | i_2 | i_3) \Rightarrow i_2)
\| REPL(w, (b | i_1 | i_2 | i_3) \Rightarrow i_3)\]

repeatedly. We then put the formula into (three-valued) CNF, which will have the side-effect of removing Boolean conditionals.

At this point the formula w is equivalent to the original w₀ and is of the form

\[\Lambda_i \ V_j \ \text{SIGNED_ATOM}_{ij}\]

where a SIGNED_ATOM is an atomic formula preceded by \(\neg\), \# or \# or by nothing. Because we are deciding validity we can restrict our attention to the separate disjunctive parts of the formula, and because the language BIT contains no function (or predicate) symbols which involve arguments of more than one type, we can group the atomic formulae into three classes: I-atoms, involving arguments of type I only, T-atoms, involving arguments of type T only, and B-atoms, which are simple U-variables. Hence, we can decide the validity of an arbitrary w ∈ BIT if we have a decision procedure for wffs w' of the form

\[V_i (\text{SIGNED-B-ATOM}_i) \ V_j (\text{SIGNED-I-ATOM}_j) \ V_k (\text{SIGNED-T-ATOM}_k)\]

Now B-atoms, I-atoms and T-atoms can have no variables in common, and therefore, in this special circumstance, w' is valid if and only if at least one of the disjuncts
\[ V_i (\text{SIGNED-B-ATOM}_i), \]
\[ V_j (\text{SIGNED-I-ATOM}_j), \] or
\[ V_k (\text{SIGNED-T-ATOM}_k) \]
is valid.

These wffs are in the theories B, BI and BT respectively. Hence, if we have decision procedures for B, BI and BT we can construct one for BIT. But the theory B is simply the three-valued propositional calculus, PC3, and is decidable, e.g., by truth tables. So the issue is finally reduced to deciding BI and BT.

This reduction of BIT to BI and BT is fairly trivial, deriving as it does from the fact that there are no functions in BIT which take mixed-type arguments. But the effect of the procedure, like that of Type Reduction, is to reduce the validity problem for a system with \( n \)-types (in this case 3) to that of systems with only a proper subset of those \( n \)-types. We call any such procedure a Type Separation, according to the following definition.

**Definition 2**: Let \( L(B, T_1, \ldots, T_n) \) be an unquantified formal language with types \( T_1, \ldots, T_n \) and Boolean, and let \( L(B, T_1, \ldots, T_{j-1}, T_j) \) and \( L(B, T_1, \ldots, T_n) \) with \( 2 \leq i \) and \( j \leq n \) be proper sublanguages. Then any procedure which reduces the decision problem for \( L(B, T_1, \ldots, T_n) \) to that for \( L(B, T_1, \ldots, T_{j-1}, T_j) \) and \( L(B, T_1, \ldots, T_n) \) will be called a **Type Separation** procedure.

We use the term Type Separation because types \( T_1, \ldots, T_{\min(i-1,j)} \) are "separated" from types \( T_{\max(i,j+1)}, \ldots, T_n \) by the reduction procedure.

In this thesis only one simple example of Type Separation is given, namely BIT \( \rightarrow \) (BI, BT), so it is slightly pretentious to give the concept a name. But there are, in fact, many decision algorithms in the literature which can be usefully viewed as Type Separations, including many producable using the Nelson-Oppen satisfiability procedure for free-product theories [Nelson 79]. In fact the notions in Nelson and Oppen's paper are closely related to and much better developed than the notion of Type Separation given here.

Like our Type Reduction procedures, the Type Separation procedure outlined here also has
a minimax property. We can extend the language BIT with any functions which involve only
Types B and I (e.g., log₂ or =-mod2) or only Types B and T (e.g., +, if Type T were Real or
Int), and the Type Separation procedure works without any change. One simply views it as a
Type Separation from BIT to BI, and BT. Hence, the list of functions involving only types B
and I, or B and T, is minimal with respect to our algorithm. However, the list of function
symbols in BIT involving both types B and I (namely the empty list) is maximal. If any such
function were added to BIT, our Type Separation procedure would have to change.

5.4 Truth Value Reduction: BI₂ → BI₃, BT₂ → BT₃

We now need decision procedures for the theories BI and BT in order to complete the
decision procedures for BIT, BITZ and BITZV. It is tempting just to say that BT is the
unquantified theory of total order, and BI is the unquantified theory of arithmetic (or, in the
absence of multiplication, Presburger arithmetic), but in fact BI and BT are formalized in PFC3
with error values and partial functions, whereas the usual theories of total order and
arithmetic are formalized in FC2 without error values and with only total functions. To be
precise, we should denote BI and BT by BI₃ and BT₃. One way to produce decision
procedures for BI and BT is to reduce them to the decision problems for BI₂ and BT₂. Such a
reduction we will call a Truth Value Reduction. In the cases of both theories BI and BT, the
truth value reductions are quite simple, and we omit them here.

5.5 Presburger Arithmetic, Arithmetic, and the Theory of Total Order

We have finally reduced the problem of deciding truth in BITZV to deciding truth in BI₂ and
BT₂. Both of these theories are well-known to be decidable.

The theory BT₂ is nothing but the unquantified theory of total order, and a decision
procedure for it is trivial (although a good decision procedure is not trivial). BT₂ contains
variables of type T and of type B. If there are m variables of type B and n of type T, then to
test validity of a formula one can test, by brute force, all of the 2ᵐ truth-value assignments
to the B-variables against all of the 2ⁿ non-isomorphic assignments (up to = and
<) of values to the T-variables. The formula is valid iff all such assignments yield true. Hence BT₂ is decidable.

The theory BL₂ is really nothing more than unquantified Presburger arithmetic augmented by Boolean variables. This theory (ignoring the Boolean variables, which cause no problems) is decidable, and in fact it is decidable even if quantification is permitted (by, e.g. Cooper's method [Cooper 72]). The unquantified fragment of Presburger arithmetic is substantially easier to decide than the quantified theory, using algorithms for integer linear programming.

If the language BITZV were to be extended with functions involving Types I and B or Types T and B, then the theories BL₂ and BT₂ would inherit these function symbols (or total extensions of them in the case of partial functions). In particular, this is true of the integer multiplication and division functions. In such a case the algorithms for Presburger arithmetic would not suffice to complete the decision procedure for BITZV. One would have to build an approximate (heuristic and incomplete) decision procedure for the augmented language BL₂ (with multiplication and division). Since BL₂ is undecidable (by the undecidability of Hilbert's 10th problem) we do not get a decision procedure for BITZV if multiplication (or scalar multiplication, i.e. i•z) is included in the language. But at least we do get a relative decision procedure, i.e. relative to an oracle or theorem prover for BL₂ with multiplication. Hence, the reduction methods we have discussed in this chapter are useful not just for finding decision procedures, but for structuring the decomposition of theorem provers for undecidable theories as well.
6. Type Reduction from BITZV to BITZ

The results of this chapter and the next are the central ones of the thesis. Here in Chapter 6 we perform the Type Reduction of BITZV to BITZ. Distributed over many sections, subsections and paragraphs are the algorithm, its specifications (intermediate assertions) and its correctness proof.

The goal of the Type Reduction algorithm is to take an arbitrary wff $w_0 \in$ BITZV and transform it into a finite set of wffs $\Omega \subseteq$ BITZ such that $\models w_0$ if and only if $\models \Omega$. i.e. every wff in $\Omega$ is valid. What we actually do, however, amounts to the same thing: we work in terms of unsatisfiability, and construct $\Omega$ such that $\not\models \Omega \iff \models w_0$.

The algorithm, of course, is expressed in the Production System notation of Chapter 2, and is a sequence of 15 steps, each of which is treated in a separate Section of the Chapter.

The proof of correctness for the algorithm is Hoare-style. We give a Precondition and a Postcondition for the entire algorithm, and we provide numerous intermediate assertions and loop invariants (Production System invariants) for the 15 steps (and sometimes for stages within steps). We then demonstrate the weak correctness of the algorithm by applying (informally) the inference rules for Production Systems and Tree-Replacement systems given in Chapter 2. Termination of each step and stage is proved separately.

Here, then, are the specifications for the Type-Reduction algorithm in this Chapter.

Precondition: $w_0 \in$BITZV (unquantified)

Postcondition:

1. $\Omega \subseteq$BITZ; $\Omega$ is finite, unquantified
2. $\models w_0 \iff \not\models \Omega$

We now begin the algorithm and its proof.
6.1 Step 1: Simplify formula; canonicalize V-terms

In this first step we will simplify the WFF \( w \) by making systematic substitutions to remove all occurrences of the following functions:

\[
\begin{align*}
\nu_1 &= \nu_2, \text{len}(\nu), \nu[i], \text{min}_\nu(\nu), \text{max}_\nu(\nu), (b \mid \nu)_\nu, (b \mid \nu_1 \mid \nu_2)_\nu, \\
(b \mid \nu_1 \mid \nu_2 \mid \nu_3)_\nu, \text{#}_\nu \text{ and } <\nu, i, t>
\end{align*}
\]

After this step these functions will never be re-introduced (except transiently) into any of the WFFs that occur later in the algorithm, so this represents a permanent reduction in the complexity of the language.

Furthermore, we will remove all nested occurrences of the vector restriction function \( o[\ldots] \). This will have the effect of restricting all V-terms to being either the constant \( E_v \), a V-variable, or a restriction of the form \( \nu \text{var}[i_1..i_2] \) where \( \nu \text{var} \) is a V-variable and \( i_1 \) and \( i_2 \) contain no V-terms.

Finally, we will arrange things so that occurrences of the \( \text{lb} \) and \( \text{ub} \) functions will have only V-variables as arguments, while the remaining functions (aside from \( o[\ldots] \)) which take V-arguments have arguments only of the forms \( E_v \) or \( \nu \text{var}[i_1..i_2] \), and never just a bare V-variable.

The algorithm in this step is one very large production system. We will break it up with assertions at various points to aid the comprehension of the algorithm and the proof of correctness.

We will assume throughout that the WFF whose validity we are testing is denoted by \( w_0 \).

6.1.0 Precondition and Postcondition for Step 1

The specifications for the whole of Step 1 are these:
Precondition:

1. \( w \in \text{BITZV} \) (unquantified)
2. \( \exists w \iff \exists w_0 \)

Postcondition:

1. \( w \in \text{BITZV with elem, (unquantified)} \)
2. \( \exists w \iff \exists w_0 \)
3. The only \( V \)-relevant function symbols occurring in \( w \) are \( \#_v \), ord, \( \#_v \), \( \# \), \( \text{elem} \), \( \{ \circ \} \), \( \circ[\circ..\circ] \) and \( E_v \).
4. All \( V \)-args to \( \#_v \), \( \#_v \) and \( \circ[\circ..\circ] \) are \( V \)-variables.
5. All \( V \)-args to \( \#_v \), \( \#_v \) and \( \{ \circ \} \) are either \( E_v \) or of the form \( v[i_1..i_2] \).
6. All occurrences of \( \text{elem} \) are in terms of the form \( \text{elem}(E_v) \) or \( \text{elem}(\text{vvar}[i..i]) \).
7. The function \( \circ[\circ..\circ] \) does not occur nested.

The algorithm in Step 1 is written as one large production system with 8 stages. However, before giving it in detail, we present here a top-level decomposition of it. Each stage is small nondeterministic PS embedded in a larger “sequential” or “Markov” structure.

\[
\begin{align*}
\omega & \leftarrow \omega_0 \\
do & [ \\
\text{Stage 1: Remove all occurrences of } \text{len}, \text{min}_v, \text{max}_v, \langle \circ | \circ \rangle_v, \\
& \langle \circ | \circ | \circ \rangle_v \text{ and } = \\
\}; \\
\text{Stage 2: Remove all occurrences of } \#_v \\
\}; \\
\text{Stage 3: Remove all occurrences of } \langle \circ | \circ | \circ | \circ \rangle_v \\
\}; \\
\text{Stage 4: Remove all occurrences of } \langle \circ, \circ, \circ \rangle \\
\];
\end{align*}
\]
(Stage 5: Remove all occurrences of \( o^0 \); introduce the \( \text{elem} \) function
)
(Stage 6: Remove nested occurrences of \( o^{0..0} \)
)
(Stage 7: Remove occurrences of \( E_v[i..j], \text{lb}(E_v), \text{ub}(E_v), \text{lb}(v[i..j]) \)
and \( \text{ub}(v[i..j]) \)
)
(Stage 8: Put \( \text{elem} \) terms in the form \( \text{elem}(v[i..i]) \)
)
] od

The following eight subsections describe the stages one by one in detail.
6.1.1 Stage 1.1: Remove occurrences of six function symbols

There are six V-relevant functions which are completely redundant in the sense that their occurrences can always be replaced by equivalent terms using different function symbols. We begin by removing all occurrences of these function symbols from the wff \( w_0 \).

6.1.1.1 Specifications

Invariant:

1. \( w \in \text{BITZV (unquantified)} \).
2. \( \exists w \Leftrightarrow \exists w_0 \).

Postcondition:

1. \( w \in \text{BITZV (unquantified)} \).
2. \( \exists w \Leftrightarrow \exists w_0 \).
3. The only V-relevant functions occurring in \( w \) are
   \( \#_v, \approx_v, \text{ord}, \text{lb}, \text{ub}, o^v, \{o\}_v, o^{[o..o]}, E_v, (o | o | o | o)_v \)
   and \( <^{o, o, o} > \). [Note: \( =, \text{len}, \min_v, \max_v, (o | o)_v \) and
   \( (o | o | o)_v \) have been removed.]

6.1.1.2 The Algorithm

\[
\begin{align*}
(a) \quad & v_1 = v_2 \quad \rightarrow \quad (\#_v v_1 \wedge \#_v v_2 | v_1 \equiv v_2)_{R} \\
(b) \quad & \text{len}(v) \quad :\!\!: \quad \text{ub}(v) - \text{lb}(v) + 1 \\
(c) \quad & \text{min}_v(v) \quad :\!\!: \quad \text{min}_z\{\{v\}\} \\
(d) \quad & \text{max}_v(v) \quad :\!\!: \quad \text{max}_z\{\{v\}\} \\
(e) \quad & (b \mid v) \quad :\!\!: \quad (b \mid v | E_v | E_v)_v \\
(f) \quad & (b \mid v_1 \mid v_2) \quad :\!\!: \quad (b \mid v_1 \mid v_2 | E_v)_v
\end{align*}
\]

6.1.1.3 Proof of Weak Correctness

The Invariant holds at the start because \( w_0 \) is quantifier-free and because \( w \) is \( \equiv w_0 \). [Recall that Step 1 begins with the assignment \( w \leftrightarrow w_0 \).] The Invariant remains invariant across the
execution of any production because no quantifiers are introduced and because each
production (a) - (f) is an identity substitution based on one of the following valid schemata.

(a) $v_1 = v_2 \quad \in B \left( \#v_1 \land \#v_2 \mid v_1 = v_2 \right)_v$

(b) $\text{len}(v) \quad \in_I \text{ub}(v) - \text{lb}(v) + 1$

(c) $\text{min}_v(v) \quad \in_T \text{min}_2(\{v\})$

(d) $\text{max}_v(v) \quad \in_T \text{max}_2(\{v\})$

(e) $(b \mid v)_v \quad \in_v (b \mid v \mid E_v \mid E_v)_v$

(f) $(b \mid v_1 \mid v_2)_v \quad \in_v (b \mid v_1 \mid v_2 \mid E_v)_v$

If and when Stage 1.1 terminates, the first and second conjuncts of its Postcondition must
hold because they occur in the Invariant.

The third conjunct says that any $V$-relevant function in BITZV may occur in the wff after
Stage 1.1 except $\in_v$, $\text{len}$, $\text{min}_v$, $\text{max}_v$, $(o \mid o)_v$ and $(o \mid o \mid o)_v$. This is, of course, true
at termination because if any of the latter six symbols did still occur in the wff, one of the
productions (a)-(f) would apply, contradicting the hypothesis that Stage 1.1 terminated.

6.1.1.4 Termination

[Omitted]
6.1.2 Stage 1.2: Remove all occurrences of $\#_v$

In this stage, we remove all occurrences of the symbol $\#_v$ without reintroducing any symbols eliminated in Step 1.2.

6.1.2.1 Notation

The pattern element "\texttt{vvar}" in production (a) must match a V-variable, not a general V-term.

The $\#$-symbols on the RHS of productions (c) and (e) represent variously the functions $\#_v$, $\#_I$ and $\#_T$.

6.1.2.2 Specifications

Invariant:

1. $w \in \text{BITZV}$ (unquantified).
2. $\mathcal{E}w \iff \mathcal{E}w_0$.
3. The only V-relevant functions occurring in $w$ are $\#_v$, $\varepsilon_v$, ord, $\text{lb}$, $\text{ub}$, $\text{o}[^0]$, $\{\text{o}\}_v$, $\text{o}[0..0]$, $E_v$, $\langle \text{o} | \text{o} | \text{o} | \text{o} \rangle_v$ and $\langle \text{o}, \text{o}, \text{o} \rangle$.

Postcondition:

1. $w \in \text{BITZV}$ (unquantified).
2. $\mathcal{E}w \iff \mathcal{E}w_0$.
3. The only V-relevant functions occurring in $w$ are $\varepsilon_v$, ord, $\text{lb}$, $\text{ub}$, $\text{o}[^0]$, $\{\text{o}\}_v$, $\text{o}[0..0]$, $E_v$, $\langle \text{o} | \text{o} | \text{o} | \text{o} \rangle_v$ and $\langle \text{o}, \text{o}, \text{o} \rangle$. [Note: $\varepsilon_v$ has been removed.]

6.1.2.3 The Algorithm

\[
\begin{align*}
\text{(a) } \#\text{vvar} & \rightarrow \text{true} \\
\end{align*}
\]
6.1.2.4 Proof of Weak Correctness

The invariant holds at the beginning of this stage because it is identical to the Postcondition of the previous stage.

Its first conjunct remains invariant because no quantifiers are introduced by (a)-(e), or any productions of Stage 1.

The second conjunct remains invariant because each production (a)-(e) is an identity substitution, as are those of the previous stage. (Note: (a) is an identity because variables range over non-error values only).

The third conjunct remains invariant because no functions not on the list are introduced by any production in this stage or the previous stage.

Showing that the Postcondition holds upon termination boils down to showing that all occurrences of the $\#_v$-symbol are removed. The $\#_v$-symbol must take as its argument a $V$-term, and the only $V$-terms left in $w$ after Stage 1.1 are either simple variables (production (a)) or are based on one of the four function symbols $E_v, <o, o, o>, (o | o | o | o)_v$ or $o[o..o]$ (productions (b)-(e)). If there were any occurrences of $\#_v$ left after Stage 1.2 halts, one of the productions (a)-(e) would apply, contradicting the termination assumption.

6.1.2.5 Termination

[Omitted]
6.1.3 Stage 1.3: Remove occurrences of the Three-way-Conditional function

Here we remove all occurrences of the three-way conditional function for Type-V, without reintroducing any of the functions removed so far.

6.1.3.1 Specifications

Invariant:

1. \( w \in \text{BITZV (unquantified)} \).
2. \( Ew \iff Ew_0 \).
3. The only V-relevant functions occurring in \( w \) are \( \equiv_v \), \( \text{ord} \), \( \text{lb} \), \( \text{ub} \), \( \o[v] \), \( \{ \o[v] \} \), \( \{ \o[v] \} \), \( E_v \), \( (\o[v]) \), \( \o[v] \), \( <^o, o, o> \).

Postcondition:

1. \( w \in \text{BITZV (unquantified)} \).
2. \( Ew \iff Ew_0 \).
3. The only V-relevant functions occurring in \( w \) are \( \equiv_v \), \( \text{ord} \), \( \text{lb} \), \( \text{ub} \), \( \o[v] \), \( \{ \o[v] \} \), \( \{ \o[v] \} \), \( E_v \) and \( <^o, o, o> \).  
   [ Note: \( (\o[v]) \) has been removed. ]

6.1.3.2 The Algorithm

\[
\begin{align*}
(a) & \quad v_1 = (b_1 \mid v_2 \mid v_3 \mid v_4) \quad \Rightarrow \quad (b_1 \mid v_1 = v_2 \mid v_1 = v_3 \mid v_1 = v_4) \\
(b) & \quad (b_1 \mid v_2 \mid v_3 \mid v_4) = v_1 \quad \Rightarrow \quad (b_1 \mid v_2 = v_1 \mid v_3 = v_1 \mid v_4 = v_1) \\
(c) & \quad \text{ord} \left( (b_1 \mid v_1 \mid v_2 \mid v_3) \right) \quad \Rightarrow \quad (b_1 \mid \text{ord}(v_1) \mid \text{ord}(v_2) \mid \text{ord}(v_3))_B \\
(d) & \quad \text{lb}(b_1 \mid v_1 \mid v_2 \mid v_3) \quad \Rightarrow \quad (b_1 \mid \text{lb}(v_1) \mid \text{lb}(v_2) \mid \text{lb}(v_3))_1 \\
(e) & \quad \text{ub}(b_1 \mid v_1 \mid v_2 \mid v_3) \quad \Rightarrow \quad (b_1 \mid \text{ub}(v_1) \mid \text{ub}(v_2) \mid \text{ub}(v_3))_1 \\
(f) & \quad (b_1 \mid v_1 \mid v_2 \mid v_3)_i \quad \Rightarrow \quad (b_1 \mid v_1[i] \mid v_2[i] \mid v_3[i])_T \\
(g) & \quad \{ (b_1 \mid v_1 \mid v_2 \mid v_3) \}_v \quad \Rightarrow \quad (b_1 \mid \{v_1\}_v \mid \{v_2\}_v \mid \{v_3\}_v)_Z
\end{align*}
\]
(h) \((b \mid v_1 \mid v_2 \mid v_3)[i_1 \ldots i_2] \rightarrow (b \mid v_1[i_1 \ldots i_2] \mid v_2[i_1 \ldots i_2] \mid v_3[i_1 \ldots i_2])_v\)

(i) \(<(b \mid v_1 \mid v_2 \mid v_3), i, t> \rightarrow (b \mid <v_1, i, t> \mid <v_2, i, t> \mid <v_3, i, t>)_v\)

6.1.3.3 Weak Correctness Proof

The Invariant holds at the start because it is identical to the Postcondition of the previous stage.

It remains invariant because 1) no quantifiers are introduced; 2) each of the productions is an identity replacement; and 3) none of the functions previously removed are reintroduced, either in this stage or in the previous two stages.

The first and second conjuncts of the Postcondition hold at termination because they are part of the Invariant. To show that the third conjunct also holds at termination we have to show that there can be no occurrences of the \((o \mid o \mid o \mid o)_v\)-symbol at termination. Examination of the \(V\)-relevant function symbols still possibly occurring in \(w\) after Stage 1.2 shows that \(V\)-terms can only occur in 12 argument positions: \(=\) (two positions), \(\text{ord}, \text{lb}, \text{ub}, \) \(\circ [\circ], \{\circ\}_v, \circ [\circ \circ], \circ, \circ, \circ\) and \((o \mid o \mid o \mid o)_v\) (three positions). If any \((o \mid o \mid o \mid o)_v\)-based term occurs in \(w\) at all, then some outermost occurrence of \((o \mid o \mid o \mid o)_v\) occurs as an argument to one of the other nine function positions. We can thus restrict consideration to outermost occurrences. But if \((o \mid o \mid o \mid o)_v\) occurred in any of those latter nine argument positions, then one of the production (a)-(i) would apply, and the production system could not terminate.

6.1.3.4 Termination

[Omitted]
6.1.4 Stage 1.4: Remove occurrences of the Assignment function

Here we remove all occurrences of the vector assignment function \(<\circ, \circ, \circ>\), once again without introducing any of the functions previously removed. (More precisely, some are reintroduced transiently and re-removed.)

6.1.4.1 Specification

Invariant:

1. \(w \in \text{BITZV} \text{ (unquantified).} \)
2. \(\exists w \iff \exists w_0 . \)
3. The only \(V\)-relevant functions occurring in \(w\) are \(\varepsilon_\nu, \text{ord, lb, ub, } \circ[\circ], \{\circ\}, \circ[\circ..\circ], \circ, \circ, \circ\) and \(E_V\).

Postcondition:

1. \(w \in \text{BITZV} \text{ (unquantified).} \)
2. \(\exists w \iff \exists w_0 . \)
3. The only \(V\)-relevant functions occurring in \(w\) are \(\varepsilon_\nu, \text{ord, lb, ub, } \circ[\circ], \{\circ\}, \circ[\circ..\circ] \text{ and } E_V . \) [Note: \(<\circ, \circ, \circ>\) has been removed.]

6.1.4.2 The Algorithm

(a) \(<v_1, i, t> \equiv v_2 ::>
(\#<v_1, i, t> \wedge \#v_2
| v_1[lb(v_1)..i-1] \equiv v_2[lb(v_2)..i-1] \wedge
| t \equiv v_2[i] \wedge v_1[i+1..ib(v_1)] \equiv v_2[i+1..ub(v_2)]
| \neg \#<v_1, i, t> \wedge \neg \#v_2

B \)

(b) \(v_2 \equiv <v_1, i, t> ::> \text{ Same as (a) above } \)

(c) \text{ord}(<v, i, t>) ::> (\#<v, i, t>
| \text{ord}(v[lb(v)..i-1]) \wedge \text{max}(v[lb(v)..i-1]) \leq t
\wedge t \leq \text{min}(v[i+1..ub(v)]) \wedge \text{ord}(v[i+1..ub(v)])\)
\( \text{B) } \)

(d) \( \text{lb}\left( \langle v, i, t \rangle \right) : \leftarrow \left( \langle \#v, i, t \mid \text{lb}(v) \rangle \right)_1 \)

(e) \( \text{ub}\left( \langle v, i, t \rangle \right) : \leftarrow \left( \langle \#v, i, t \mid \text{ub}(v) \rangle \right)_1 \)

(f) \( \langle v, i, t \rangle[i_2] : \leftarrow \left( \langle \#v, i, t \rangle \land \#i_2 \land \text{lb}(v) \leq i_2 \land i_2 \text{sub}(v) \right) \\
\left( \langle i_1 = i_2 \mid t \rangle_T \\
\left( \langle v[i_2] \rangle \right)_T \right) \)

(g) \( \{ \langle v, i, t \rangle \} : \leftarrow \left( \langle \#v, i, t \rangle \left| \{ v[\text{lb}(v) \ldots i-1] \} + \{ t \} + \{ v[i+1 \ldots \text{ub}(v)] \} \right) Z \)

(h) \( \langle v, i, t \rangle[i_2 \ldots i_3] : \leftarrow \left( \langle \#v, i, t \rangle[i_2 \ldots i_3] \right) \\
\langle i_1 < i_2 \land i_3 < i_1 \land t \rangle \left( \langle v[i_2 \ldots i_3] \rangle \right)_v \left( \langle v[i_2 \ldots i_3], i_1, t \rangle \right)_v \)

\)

6.1.4.3 Weak Correctness Proof

The Invariant holds at the start because it is the Postcondition of the previous Stage.

Its first conjunct remains invariant because no quantifier is introduced by any production, here or in any of the previous stages.

Its second conjunct remains invariant because each production (a)-(h) is an identity replacement, as are those of the previous stages.

It might appear that the third conjunct would not remain invariant since we reintroduce occurrences of the \( \langle o \mid o \rangle_v, \langle o \mid o \mid o \rangle_v, \#_v, \max_v \) and \( \min_v \) symbols on the RHS of some of the productions. All of those symbols were removed in earlier stages. However, the reintroduction is only momentary since between any two production executions in this Stage 1.4, all of Stages 1.1 - 1.3 run to completion, thus re-removing the offending symbols in time for the Invariant to be reestablished.
The Postcondition differs from the Invariant only in the requirement that all occurrences of $<\circ, \circ, \circ>$ be removed. The proof that this occurs in similar to that for the $(\circ | \circ | \circ | \circ)_y$-symbol in the previous step. If at termination there were any occurrences of the $<\circ, \circ, \circ>$-function, then there must be at least one outermost occurrence. At this point in the algorithm an outermost occurrence must be as an argument to one of the functions $=, \text{ord, lb, ub, } \circ[\circ], \{\circ\}$ or $\circ[\circ, \circ]$. However, each of those possibilities is eliminated by one of the productions (a) - (h). The production system cannot terminate with an outermost occurrence of $<\circ, \circ, \circ>$, and thus cannot terminate with any occurrence of $<\circ, \circ, \circ>$.

6.1.4.4 Termination

[Omitted]
6.1.5 Stage 1.5: Replace occurrences of $v[i]$ with occurrences of the elem function

In this stage we replace all terms of the form $v[i]$ with terms of the form $\text{elem}(v[i..i])$. We thus introduce a function symbol, elem, which was not present before in the language BITZV.

The replacement of $v[i]$ by $\text{elem}(v[i..i])$ does not really constitute progress toward the goal of eliminating Type-$V$, but it does make the exposition and proof somewhat easier later on. The reason is that it allows us to think exclusively in terms of general subintervals of a vector, without treating subintervals of length one as a special case, as we would have to do if the $v[i]$ notation remained.

6.1.5.1 Specifications

Invariant:

1. $w \in \text{BITZV}$ with elem (unquantified).
2. $\mathcal{F}w \iff \mathcal{F}w_0$.
3. The only $V$-relevant functions occurring in $w$ are $\mathcal{E}_v$, $\text{ord}$, $\text{lb}$, $\text{ub}$, $\circ[\circ]$, $\text{elem}$, $\{\circ\}$, $\circ[\circ..\circ]$ and $\mathcal{E}_v$. [Note: elem has been added.]
4. All occurrences of elem are in terms of the form $\text{elem}(v[i_1..i_2])$, where $i_1 = i_2$.

Postcondition:

1. $w \in \text{BITZV}$ with elem (unquantified).
2. $\mathcal{F}w \iff \mathcal{F}w_0$.
3. The only $V$-relevant functions occurring in $w$ are $\mathcal{E}_v$, $\text{ord}$, $\text{lb}$, $\text{ub}$, elem, $\{\circ\}$, $\mathcal{E}_v$ and $\circ[\circ..\circ]$. [Note: $\circ[\circ]$ has been removed; elem has been added.]
4. All occurrences of elem are in terms of the form $\text{elem}(v[i_1..i_2])$ where $i_1 = i_2$. 

6.1.5.2 The Algorithm

\[
\begin{align*}
\forall[i]: \Rightarrow \text{elem}(v[i..i])
\end{align*}
\]

6.1.5.3 Proof of Weak Correctness

The Invariant holds at the beginning because it is implied by the Postcondition of the previous stage. (Conjuncts 1 and 3 are, in fact, strictly weaker.)

The invariance of Conjuncts 1 and 3 is trivial. The invariance of Conjunct 2 follows from the fact that the production is an identity replacement.

The Postcondition differs from the Invariant only in the requirement that there be no occurrences of \( \theta[\theta] \) at termination. It is trivial to note that if there were any such occurrences, the production would be eligible to execute, contradicting the termination assumption.

6.1.5.4 Termination

[Omitted]
6.1.6 Stage 1.6: Remove nested occurrences of the Restriction function

At this point in the algorithm the only V-valued functions left in the wft $w$ are $E_v$ and $o[\cdot \ldots \cdot]$, and thus the only V-terms are $E_v$, simple V-variables, and terms formed by composition of the $o[\cdot \ldots \cdot]$ function. In the step we will remove all nested occurrences of $o[\cdot \ldots \cdot]$. In other words, after this step there will be no terms of the form $v_1[i_1..i_2]$ such that another term $v_2[i_3..i_4]$ occurs as a subterm of $v_1[i_1..i_2]$. As a result, all V-terms will have one of four forms: $E_v$, $vvar$, $E_v[i_1..i_2]$ and $vvar[i_1..i_2]$.

6.1.6.1 Specifications

**Invariant:**

1. $w \in \text{BITZV with elem, (unquantified)}$.
2. $\mathcal{F}w \iff \mathcal{F}w_0$.
3. The only V-relevant functions occurring in $w$ are $\varepsilon$, ord, lb, ub, elem, $\{o\}$, $E_v$ and $o[\cdot \ldots \cdot]$.
4. All occurrences of elem are in terms of the form $\text{elem}(v[i_1..i_2])$ where $i_1..i_2$.

**Postcondition:**

1. $w \in \text{BITZV with elem, (unquantified)}$.
2. $\mathcal{F}w \iff \mathcal{F}w_0$.
3. The only V-relevant functions occurring in $w$ are $\varepsilon$, ord, lb, ub, elem, $\{o\}$, $E_v$ and $o[\cdot \ldots \cdot]$.
4. All V-terms match one of the forms: $E_v$, $vvar$, $E_v[i_1..i_2]$ or $vvar[i_1..i_2]$.
5. All occurrences of elem are in terms of the form $\text{elem}(v[i_1..i_2])$ where $i_1..i_2$.
6. The function $o[\cdot \ldots \cdot]$ does not occur nested.
6.1.6.2 The Algorithm

\(<v_2[i_3..i_4] = v_1[i_1..i_2] \land v_2 \text{ different from } v_1 \land v_1[i_1..i_2] = w \Rightarrow w \leftarrow (v_2[i_3..i_4] = E_v \Rightarrow \text{REPLALL}(w, v_2[i_3..i_4] :\rightarrow E_v) \land v_2[i_3..i_4] = v' \Rightarrow \text{REPLALL}(w, v_2[i_3..i_4] :\rightarrow v'))\>

Notes:

1. The requirement that \(v_2\) be different from \(v_1\) simply insures that \(v_2[i_3..i_4]\) is a proper subterm of \(v_1[i_1..i_2]\), so that we are actually removing nested occurrences of \(v[0..0]\). Without this requirement, the loop might not terminate.

2. The variable \(v'\) is a new \(V\)-variable, distinct from all others occurring in \(\Omega\). A new \(v'\) is generated with each production iteration.

6.1.6.3 Proof of Weak Correctness

The Invariant holds at the beginning because it is the Postcondition of the previous stage. Furthermore, conjuncts 1, 3 and 4 of the Invariant are obviously invariant. The only question is with Conject 2.

The easiest way to see that Conject 2 is invariant is to argue informally in the following manner. Suppose, for the moment, that in PFC3 both bound and free variables were allowed to range over the error value as well as non-error values. Then it would be obvious that

\(\exists w = (\forall v'. v_2[i_3..i_4] = v' \Rightarrow \text{REPLALL}(w, v_2[i_3..i_4] :\rightarrow v'))\)

and thus that

\(\exists w \iff \forall v_2[i_3..i_4] = v' \Rightarrow \text{REPLALL}(w, v_2[i_3..i_4] :\rightarrow v').\)

However, because we do not permit variables to range over the error value, we must consider two cases:

1. The case where \(v_2[i_3..i_4]\) has the same value as \(E_v\), and
2. The case where $v_2[i_3..i_4]$ has a non-error value.

Splitting into those two cases yields

$$I = w \cdot (v_{i_3..i_4} = E \Rightarrow REPLALL(w, v_{i_3..i_4} \Rightarrow E_v)) \land$$
$$\forall v'. (v_{i_3..i_4} = v' \Rightarrow REPLALL(w, v_{i_3..i_4} \Rightarrow v'))$$

This leads trivially to

$$I = I = w \cdot (v_{i_3..i_4} = E \Rightarrow REPLALL(w, v_{i_3..i_4} \Rightarrow E_v)) \land$$
$$\forall v'. (v_{i_3..i_4} = v' \Rightarrow REPLALL(w, v_{i_3..i_4} \Rightarrow v'))$$

which is what is needed for the invariance of Conjunct 2. This completes the proof that the Invariant is invariant over the productions here in Stage 6. We omit the easy proof that the Invariant is also invariant across Stages 1 to 5 of Step 1.

To prove that the Postcondition holds at termination, it suffices to remark that the Invariant accounts for Conjuncts 1, 2, 3 and 5 of the Postcondition. Conjunct 6 is established because if any nesting of $o[0..0]$ occurred in $w$, the pattern of the production would be satisfiable, contradicting the termination assumption. Conjunct 4 follows from Conjuncts 3 and 6.

6.1.6.4 Termination

[Omitted]
6.1.7 Step 1.7: Rewrite certain miscellaneous expressions

Expressions of the form $E_v[i_1..i_2]$, $lb(E_v)$, $ub(E_v)$, $lb(v[i..j])$, $ub(v[i..j])$, $\text{elem}(v\text{var})$, $\{v\text{var}\}$, $\text{ord}(v\text{var})$, $v\text{var}=v$ and $v=v\text{var}$ can all be replaced by more useful forms. In particular, we change most occurrences of a bare $V$-variable $v\text{var}$ to the form $v\text{var}[lb(v\text{var}).ub(v\text{var})]$, so that $V$-terms are more uniform.

6.1.7.1 Specifications

**Invariant:**

1. $w \in \text{BITZV}$ with $\text{elem}$, (unquantified).
2. $\forall w \iff \exists \forall w_0$.
3. The only $V$-relevant function symbols occurring in $w$ are $\exists$, $\text{ord}$, $\text{lb}$, $\text{ub}$, $\text{elem}$, $\{o\}$, $E_v$ and $o[o]..o]$.
4. All $V$-terms match one of the forms: $E_v$, $v\text{var}$, $E_v[i_1..i_2]$ or $v\text{var}[i_1..i_2]$.
5. All occurrences of $\text{elem}$ are in terms of the form $\text{elem}(v[i_1..i_2])$ where $I::i=I::i2$.
6. The function $o[o]..o]$ does not occur nested.

**Postcondition:**

1. $w \in \text{BITZV}$ with $\text{elem}$, (unquantified).
2. $\forall w \iff \exists \forall w_0$.
3. The only $V$-relevant function symbols occurring in $w$ are $\exists$, $\text{ord}$, $\text{lb}$, $\text{ub}$, $\text{elem}$, $\{o\}$, $E_v$ and $o[o]..o]$.
4. All $V$-args to $\text{lb}$, $\text{ub}$, $o[o]..o]$ are $V$-variables.
5. All $V$-args to $E_v$, $\text{ord}$, and $\{o\}$ are either $E_v$ or of the form $v\text{var}[i_1..i_2]$.
6. All occurrences of $\text{elem}$ are in terms of the form $\text{elem}(E_v)$ or $\text{elem}(v\text{var}[i_1..i_2])$ where $I::i=I::i2$.
7. The function $o[o]..o]$ does not occur nested.
6.1.7.2 The Algorithm

\[ \begin{align*}
(a) & \quad \text{E}_v[i_1\ldots i_2] : \rightarrow \text{E}_v \\
(b) & \quad \text{lb(} \text{E}_v) : \rightarrow \text{E}_1 \\
(c) & \quad \text{ub(} \text{E}_v) : \rightarrow \text{E}_1 \\
(d) & \quad \text{lb(v[i..j])} : \rightarrow \langle \#v[i..j] \mid i \rangle \\
(e) & \quad \text{ub(v[i..j])} : \rightarrow \langle \#v[i..j] \mid j \rangle \\
(f) & \quad \{\text{var}\} : \rightarrow \{\text{var}\ [\text{lb(vvar)}..\text{ub(vvar)}]\} \\
(g) & \quad \text{ord(vvar)} : \rightarrow \text{ord(} \text{var}\ [\text{lb(vvar)}..\text{ub(vvar)}]\) \\
(h) & \quad \text{var} = v : \rightarrow \text{var}\ [\text{lb(vvar)}..\text{ub(vvar)}] = v \\
(i) & \quad \text{v} = \text{var} : \rightarrow \text{v} = \text{var}\ [\text{lb(vvar)}..\text{ub(vvar)}]
\end{align*} \]

6.1.7.3 Proof of Weak Correctness

The Invariant holds at the start of this stage because it is the Postcondition of the previous stage.

The invariance proofs for Conjuncts 1 and 5 are trivial. Inspection reveals that they are invariant over all of the productions of this stage and of all previous stages.

Conjunct 2 is invariant over the productions of Stage 7 because each production is an identity replacement. Since we have already established that Conjunct 2 is invariant over all productions in the previous stages, it is also invariant over Stage 7.

To prove the invariance of Conjuncts 3, 4 and 6 we will have to use more complicated arguments. These conjuncts are either not invariant over all productions in Stage 7 (e.g. Conjunct 3 is not invariant across production (d)) or they are not invariant across all of the productions in previous stages (e.g. Conjunct 6 is not invariant across the production of Stage 5).

Let us refer to the six conjuncts of the Invariant as \(C_1 \ldots C_6\), and let \(P_0\) denote the precondition for all of Step 1. Then the weak correctness proofs for the previous stages have established the following result:

\[ P_0\{\text{do [Stages 1-6] od} \} C_1 \text{ and } \ldots \text{ and } C_6. \]
Note that $C_1$ and ... and $C_6$ is both the Postcondition for Stage 6 and the Precondition/Invariant for Stage 7.

By modifying the proofs in Stages 1-6 we could establish that

$$C_1 \text{ and } C_2 \text{ and } C_5 \{ \text{do Stages 1-6} \} \text{ and } C_1 \text{ and } ... \text{ and } C_6.$$ 

This result is slightly stronger because the precondition, $C_1$ and $C_2$ and $C_5$ is slightly weaker than $P_0$ in that it permits some occurrences of the elem function in the wff $w$. Using this result, for which we omit the proof, we can complete the invariance proof if we can show, for each production $\alpha \rightarrow \beta$ of Stage 7, that

$$C_1 \text{ and } ... \text{ and } C_6 \text{ and } \alpha \rightarrow \beta \{ w \rightarrow \text{REPL}(w, \alpha \rightarrow \beta) \} \text{ C_1 and C_2 and C_5}.$$ 

In the case of each of the nine productions of this result is simple, so we omit details.

The Postcondition for this step follows from the Invariant and the fact that all patterns on the LHS of productions are unmatchable when the system halts. From the Invariant we know that all V-terms are of the form: $E_v$, vvar, $E_v[i_1..i_2]$ or vvar$[i_1..i_2]$. Production (a) removes terms of the form $E_v[i_1..i_2]$. Productions (b)-(e) guarantee that only V-variables remain as arguments to lb and ub. Productions (f)-(i) guarantee that simple V-variables will not appear as arguments to $\{ o \}$, $\equiv$ or ord. V-variables cannot occur as arguments to elem because of the fifth conjunct of the invariant.

6.1.7.4 Termination

[Omitted]
6.1.8 Put elem terms into canonical form

Up to now we have guaranteed that all elem terms are either of the form \( \text{elem}(E_v) \) or \( \text{elem}(v\text{var}[i_1..i_2]) \) where \( E_1 \equiv i_2 \). We now strengthen this condition so that \( i_1 \) and \( i_2 \) actually are the same term (rather than merely equivalent). This will simplify later steps by reducing the number of distinct terms occurring in index positions in the wff \( w \).

6.1.8.1 Specifications

**Invariant:**

1. \( w \in \text{BITZV with elem, (unquantified)} \).
2. \( \#w \iff \#w_0 \).
3. The only \( V \)-relevant function symbols occurring in \( w \) are \( =, \) ord, lb, ub, elem, \( \{o\} \), \( E_v \) and \( o[0..o] \).
4. All \( V \)-args to lb, ub, \( o[0..o] \) are \( V \)-variables.
5. All \( V \)-args to \( \equiv_v, \) ord, and \( \{o\} \) are either \( E_v \) or of the form \( v\text{var}[i_1..i_2] \).
6. All occurrences of elem are in terms of the form \( \text{elem}(E_v) \) or \( \text{elem}(v\text{var}[i_1..i_2]) \), where \( E_1 \equiv i_2 \).

**Postcondition:**

1. \( w \in \text{BITZV with elem, (unquantified)} \).
2. \( \#w \iff \#w_0 \).
3. The only \( V \)-relevant function symbols occurring in \( w \) are \( =, \) ord, lb, ub, elem, \( \{o\} \), \( E_v \) and \( o[0..o] \).
4. All \( V \)-args to lb, ub, \( o[0..o] \) are \( V \)-variables.
5. All \( V \)-args to \( \equiv_v, \) ord, and \( \{o\} \) are either \( E_v \) or of the form \( v\text{var}[i_1..i_2] \).
6. All occurrences of elem are in terms of the form \( \text{elem}(E_v) \) or \( \text{elem}(v[i..i]) \).
7. The function \( o[0..o] \) does not occur nested.
6.1.8.2 The Algorithm

\[
\begin{align*}
\text{elem}(v[i_1..i_2]) = \& \text{ and } i_1 \text{ different from } i_2 \\
\text{Rightarrow} \\
\& \text{ REPL}(u, \text{elem}(v[i_1..i_2]) : \text{elem}(v[i_1..i_1]))
\end{align*}
\]

Note: The requirement that \(i_1\) be different from \(i_2\) means that \(i_1\) and \(i_2\) must be distinct terms.

6.1.8.3 Proof of Weak Correctness

Except for Conjunct 2, the fact that Invariant is preserved across the production in Stage 8 is obvious.

Conjunct 2 is preserved because of Conjunct 6, i.e., the substitution of \(i_1\) for \(i_2\) is performed only under the condition that \(E_{i_1}\). The production, then, is an identity replacement under the conditions which apply when it is executed.

To complete the proof of invariance, we note that none of the productions of the previous seven stages ever applies between executions of the production of Stage 8. Therefore, the Invariant is preserved not just across the productions of Stage 8, but across the null executions of Stages 1-7 as well.

The Postcondition differs from the Invariant only in the requirement that arguments to elem be of the form \(E_V\) or \(v[i..i]\), instead of the weaker requirement that they be of the form \(E_V\) or \(v[i_1..i_2]\) with \(E_{i_1}\). The pattern-part of the production is clearly satisfiable, and hence the PS cannot halt, as long as there is any occurrence of elem whose argument is of the form \(v[i_1..i_2]\) where \(i_1\) and \(i_2\) are distinct terms. Hence, the postcondition is established.

6.1.8.4 Termination

Because, as we have pointed out, none of the productions of Stages 1-7 are eligible to execute between executions of the production in this Stage 8, we need only show that Stage
8 by itself terminates. This is trivial, because each production reduces the number of occurrences of $\text{elem}(v_{i_1...i_2})$ in $w$ in which $i_2$ is different from $i_1$. 
6.2 Step 2: Put formula into DNF; separate the disjuncts

So far what we have done is to take the \( w \), which was originally identical to \( w_0 \), and reduce the number of linguistic constructs occurring in it while preserving the relationship \( Fw \iff Fw_0 \). In the remainder of the Type Reduction of BITZV to BITZ we will continue to reduce the number of linguistic constructs permitted. However, we will be working not with a single wff \( w \), but a finite set of wffs \( \Omega \). And the relation that will be kept invariant will not be \( Fw \iff Fw_0 \), but \( \not\forall \Omega \iff \exists w_0 \).

It is important throughout the rest of this chapter to clearly understand the notation \( \not\forall \Omega \). Usual mathematical practice is to say that a (finite) set of wffs is unsatisfiable (satisfiable) whenever the conjunction of its elements is unsatisfiable (satisfiable). Written symbolically this would be

\[ \not\forall \Omega \iff \not\exists (\land_{w\in\Omega} w) \]

But we mean something much stronger, namely, that each individual member of \( \Omega \) is unsatisfiable:

\[ \not\forall \Omega \iff \land_{w\in\Omega} (\not\exists w) \]

which is the same as saying that the disjunction of the elements is unsatisfiable, i.e.

\[ \not\forall \Omega \iff \not\exists (\lor_{w\in\Omega} w) \]

Here in Step 2 we put the wff \( \neg T(w) \) into disjunctive normal form (3-valued) and then initialize \( \Omega \) to be the set of disjuncts resulting. This establishes the relation \( \not\forall \Omega \iff \exists w_0 \) while retaining all of the other linguistic properties we have established so far.

6.2.1 Specifications

Precondition: Same as Postcondition for Step 1.

Postcondition:
1. \( \Omega \subseteq \text{BITZV with elem; } \Omega \text{ quantifier-free and finite} \)
2. $Ew_0 \iff \not\exists \Omega$.

3. The only $V$-relevant functions appearing in any $w \in \Omega$ are $\oplus$, $\lor$, $\land$, $\exists$, $\forall$, $\{ \}$, $E_v$ and $\alpha[\alpha] \$.

4. All $V$-args to $\exists$, $\land$, $\alpha$, $\{ \}$ are simple $V$-variables.

5. All $V$-args to $\forall$, $\lor$, $\exists$, $\{ \}$ are $E_v$ or $v\alpha[i_{1..i_2}]$.

6. All occurrences of $\alpha$ are in terms of the form $\alpha(E_v)$ or $\alpha(v\alpha[i..i])$.

7. The function $\alpha[\alpha]$ does not occur nested.

8. All $w \in \Omega$ can be rearranged to match the pattern

$$U \land \alpha \land \alpha^\prime \land \alpha^\prime \land \alpha^\prime \land \alpha^\prime \land \alpha^\prime \land \alpha^\prime$$

where

- $U$ contains no occurrences of $\alpha$ or $\lor$
- $\alpha$ is a conjunction of literals of the form $v_1 \equiv v_2$
- $\alpha^\prime$ is a conjunction of literals of the form $\neg v_1 \equiv v_2$
- $\alpha^\prime$ is a conjunction of literals of the form $\#(v_1 \equiv v_2)$
- $\alpha^\prime$ is a conjunction of literals of the form $\neg \#(v_1 \equiv v_2)$
- $\alpha^\prime$ is a conjunction of literals of the form $\lor v$ or $\lor \alpha$
- $\alpha^\prime$ is a conjunction of literals of the form $\neg \lor v$
- $\alpha^\prime$ is a conjunction of literals of the form $\# \lor v$
- $\alpha^\prime$ is a conjunction of literals of the form $\neg \# \lor v$

6.2.2 The Algorithm

$w \leftarrow \neg T(w)$;

$do$

Productions for 3-valued DNF; (see Chapter 3.)

$od$;

$\Omega \leftarrow \{ w \}$;

$od$
\[(\mu_1 \lor \mu_2) \in \Omega \Rightarrow \Omega \leftarrow (\Omega - \{\mu_1 \lor \mu_2\}) \cup \{\mu_1, \mu_2\}\]

6.2.3 Proof of Correctness

We will confine the proof to a few observations. The Precondition of this stop holds because it is the Postcondition for Step 1. After the third statement in the Algorithm the Invariant \( \mathfrak{A} \Omega \iff \mathcal{F}w_0 \) is established because of the relationship between unsatisfiability and validity that was described in Chapter 3, namely

\[\mathfrak{A} \neg T(w) \iff \mathcal{F}w,\]

and also because the transformation to DNF is equivalence (\( \equiv_0 \)) preserving.

The last part of the algorithm, the production system that breaks the wff \( w \) into its component disjuncts, preserves the truth of \( \mathfrak{A} \Omega \iff \mathcal{F}w_0 \) because of the following fact, noted in Chapter 3:

\[\mathfrak{A} (w_1 \lor w_2) \iff (\mathfrak{A} w_1 \text{ and } \mathfrak{A} w_2).\]

The last specification in the Postcondition, that all wffs in \( \Omega \) match the "pattern"

\[U \land \text{EQ} \land \text{EQ}^\ast \land \text{EQ}^+ \land \text{ORD} \land \text{ORD}^+ \land \text{ORD}^+ \land \text{ORD}^+\]

follows from the fact that each wff is in three-valued DNF, and contains no \( \nu \)-symbols. We simply rearrange the conjuncts to fit this pattern, lumping all atoms not based on the \( \equiv_\nu \) or ord symbols into the part labelled "U". Actually, the DNF-algorithm does much more than required for this Postcondition. It is not necessary that the U-part of the formula be in DNF; it is only necessary that it not contain \( \equiv_\nu \) or ord. In fact, the algorithm would perform better if the U-part were not reduced completely to DNF; however, we ignore this possibility.

6.2.4 Termination

The fourth statement of the algorithm terminates because each iteration reduces the total number of occurrences of the \( \nu \)-symbol in \( \Omega \). The fact that the DNF productions terminate was established in Chapter 3.
6.3 Step 3: Remove EQ\#, EQ\^\#, ORD\#, ORD\^\#

The EQ\#, EQ\^\#, ORD\# and ORD\^\# parts of the wffs in \( \Omega \) can be removed by simple substitutions.

6.3.1 Specifications

**Precondition:** Same as Postcondition for Step 2.

**Invariant:** Same as Postcondition for Step 2.

**Postcondition:**

1. \( \Omega \subseteq \text{BITZV} \) with elem; \( \Omega \) quantifier-free and finite.
2. \( Ew_0 \iff \not \exists \Omega \).
3. The only \( V \)-relevant function symbol occurring in \( \Omega \) are \( \#_V \), ord, lb, ub, elem, \( \{ ^o \}, E_v \) and \( o^{[o..o]} \).
4. All \( V \)-args to lb, ub and \( o^{[o..o]} \) are \( V \)-variables.
5. All \( V \)-args to \( \#_V \), ord, and \( \{ ^o \} \) are \( E_v \) or \( \text{vvar}^{[o..o]} \).
6. All occurrences of elem are in terms of the form \( \text{elem}(E_v) \) or \( \text{elem}(\text{vvar}[i..i]) \).
7. The function \( o^{[o..o]} \) does not occur nested.
8. All wffs \( w \in \Omega \) can be rearranged to be of the form

\[
U \land \text{EQ} \land \text{EQ}^\# \land \text{ORD} \land \text{ORD}^\#
\]

where \( U, \text{EQ}, \text{EQ}^\#, \text{ORD} \) and \( \text{ORD}^\# \) have the same definitions as in Step 2.

6.3.2 The Algorithm

Apply the following transformation to each \( w \in \Omega \)

\[
\text{do}
\]

- (a) \( \#_B(\#_1 = \#_2) :\to \text{true} \)
- (b) \( \#_B(\text{ord}(E_v)) :\to \text{false} \)
- (c) \( \#_B(\text{ord}(\#_1 \ldots \#_2)) :\to \text{lb}(v) \leq i_1 \land \#_1 \land \#_2 \land i_1 \leq i_2 + 1 \land i_2 + 1 \leq \text{ub}(v) + 1 \)
6.3.3 Proof of Weak Correctness

The Invariant holds at the start because it is the same as the Postcondition of the previous step. By inspection it is trivial that conjuncts 1, 3-8 remain invariant over all three productions. The second conjunct, $\not\forall \Omega \iff \forall w_0$, remains invariant because each of the productions (a)-(c) is an identity substitution.

The Postcondition differs from the Invariant only in the requirement that there be no conjuncts of the form $\#(v_1 \equiv v_2)$, $\#(v_1 \equiv v_2)$, $\#\text{ord}(v)$ or $\#\text{ord}(v)$ occurring in any wffs $w \in \Omega$. Conjuncts of the forms $\#(v_1 \equiv v_2)$ and $\#(v_1 \equiv v_2)$ are eliminated by production (a); the production system cannot halt with either kind of subexpression still present. Similarly, productions (b) and (c) prevent the production system from halting with any subexpression of the form $\#\text{ord}(v)$. (Recall that the Invariant requires all arguments to ord to be either $E_v$ or of the form $v[i_1..i_2]$ where $v$ is a variable; the two cases are treated separately by productions (b) and (c). Separate cases were not necessary for production (a): the value of $\#(v_1 \equiv v_2)$ is always true, regardless of the forms of $v_1$ and $v_2$.)

6.3.4 Termination

For each wff $w$, the production system halts because each iteration decreases the size (measured in conjuncts, say) of $\text{EQ}^* \land \text{EQ}^{**} \land \text{ORD}^* \land \text{ORD}^{**}$. Since $\Omega$ has a finite number of wffs, the whole process must terminate.
6.4 Step 4: Remove EQ$^*$ and ORD$^*$

In this step we will remove negated occurrences of $\exists_v$-atoms and ord-atoms. The removal will be based on the following facts:

\[ \models \neg(v_1 \equiv v_2) \equiv \left( \neg lb(v_1) \equiv lb(v_2) \lor \neg ub(v_1) \equiv ub(v_2) \lor \right. \\
\left. \exists i.\ lb(v_1) \leq i \land i \leq ub(v_1) \land \neg v_1[i] \equiv v_2[i] \right) \]

\[ \models \neg \text{ord}(v) \equiv \left( \exists i.\ lb(v) \leq i \land i < ub(v) \land v[i] > v[i+1] \right) \]

Since the quantifiers are existential and we are concerned with preserving satisfiability/unsatisfiability, we will be able to remove the quantifiers by pulling them to the front of the formula and then dropping them (as is done with universal quantifiers when validity is at issue).

We will describe this more carefully later in the section on the proof of weak correctness.

6.4.1 Specifications

**Precondition:**  
Same as Postcondition for Step 3.

**Invariant part (b):**  
Same as Postcondition for Step 3 (except for the quantifier-free requirement in Conjunct 1.)

**Postcondition:**

1. $\Omega \subseteq \text{BITZV}$ with elem; $\Omega$ quantifier-free and finite.
2. $\forall w_0 \iff \# \Omega$.
3. The only V-relevant function symbols occurring in $\Omega$ are $\exists_v$, ord, lb, ub, elem, $\{o\}$, $E_v$ and $o[\ldots o]$.
4. All V-args to lb, ub and $o[\ldots o]$ are V-vars.
5. All V-args to $\exists$, ord, elem and $\{o\}$ are $E_v$ or vvar$[i_1..i_2]$.
6. All occurrences of elem are in terms of the form elem$(E_v)$ or elem$(\text{vvar}[i..i])$.
7. The function $o[\ldots o]$ does not occur nested.
8. All wffs \( w \in \Omega \) are of the form

\[ U \land EQ \land ORD \]

where \( U, EQ \) and \( ORD \) have the same definitions as before in Step 2, i.e.

- \( U \) contains no occurrences of \( =_v \) or \( ord \)
- \( EQ \) is a conjunction of literals of the form \( v_1 = v_2 \)
- \( ORD \) is a conjunction of literals of the form \( \text{ord}(v) \).

6.4.2 Notation

In part (b) of the next algorithm the symbol \( j' \) represents a new integer variable, distinct from all other integer variables occurring in the formula in question. This is to prevent name clashes in part (c) of the algorithm.

Each time the second or seventh production in part (b) is executed a new variable name is chosen, but the same variable is substituted for the several times that \( j' \) appears on the RHS of those productions.

6.4.3 The Algorithm

For all \( w \in \Omega \) apply the following transformation:

\[
\text{do}\]

(a) \[
(b_1 | b_2) \rightarrow (T(b_1) \land b_2) \lor (\neg T(b_1) \land \text{error}) \]

\[
(b_1 | b_2 | b_3) \rightarrow (T(b_1) \land b_2) \lor (F(b_1) \land b_3) \lor (E(b_1) \land \text{error}) \]

\( \text{vvar}[i] \rightarrow \text{elem}(\text{vvar}[i..i]) \)

\( \#\text{vvar}[i_1..i_2] \rightarrow \#i_1 \land \#i_2 \land 1b(\text{vvar}) \leq i_1 \land i_1 \leq i_2 + 1 \land i_2 \leq \text{sub}(\text{vvar}) \)

(b) \[
\neg \text{ord}(E_v) \rightarrow E_B \]


6.4.4 Proof of Weak Correctness

Part (a) of the algorithm is really for readability only. It merely removes occurrences of \((\circ \mid \circ)_B, (\circ \mid \circ \mid \circ)_B, \mu_v\) and \(\circ[v]\) which are introduced in part (b) of the algorithm. If we were willing to expand the expressions in part (b), part (a) could be removed, but the algorithm would no longer be as readable. We will not bother to prove correctness of part
(a), except to note that each production is an identity replacement.

Part (b) is the key part of this step, and the Invariant applies to it. The Invariant holds at the start because it is implied by the Precondition.

Conjuncts 4-8 of the Invariant can be seen to be invariant over the six productions of part (b) by inspection.

Conjunct 3 is Invariant despite the temporary introduction of the two- and three-way conditionals, the $\#_V$-function, and the $\circ[r]$-function, because these function symbols are removed by part (a) between executions of productions from part (b).

Conjunct 2 is Invariant because each production is an identity replacement.

Conjunct 1 is trivially invariant. Note that Conjunct 1 does not require quantifier free-ness. We do introduce quantifiers here, but remove them in Steps (c) and (d) to reestablish the quantifier-free property.

All clauses of the Postcondition except the first and the last are part of the Invariant. After part (b) the last conjunct holds because any occurrence of a negated ord-atom or negated $\#_V$-atom would match one of the patterns of part (b). (Recall that all arguments to $\equiv$ or ord are of the forms $E_\vee$ or $v\text{var}[i_1..i_2]$)

Thus, all but the first conjunct of the Postcondition holds after part (b). Parts (c) and (d) remove the quantifiers introduced in part (b). Part (c) brings the quantifiers to the front of the formula (prenex form) in the usual fashion. Since in each case j cannot occur free in $b_2$ (since j was a new variable when introduced in part c), each production is an identity transformation. Upon termination of part (c) the quantifiers must all be at the front (root) of the wff because they started (after part (b)) with nothing but $\wedge$'s and $\vee$'s between them and the root (i.e. no $\neg$'s or $\equiv$'s, etc.)

Part (d), which removes all of the quantifiers, is justified because we are concerned with satisfiability (instead of validity): The relationship
\( \forall \exists j.w \iff \exists w. \)
demonstrates that Conjunct 2 of the Invariant/Postcondition continues to hold.

After parts (c) and (d), all of the Postcondition has been established.

6.4.5 Termination

We will not bother with part (a), but assume it has been incorporated into part (b).

Part (b) terminates because each production reduces the size of the \( \text{EQ}^\wedge \land \text{ORD}^\wedge \) part of the formula, i.e. the number of negated \( \exists \lor \) ord atoms is reduced each iteration.

Part (c) terminates because the sum of the tree-depths of all quantifiers is reduced each iteration.

Part (d) terminates because the number of quantifiers is reduced each iteration.

The above argument shows that parts (a), (b), (c) and (d) terminate for any wff \( w \in \Omega \). Since \( \Omega \) is finite, the whole outer loop terminates.
6.5 Step 5: Hypothesize all possible index order configurations

In this step we perform the most critical part of the Type Reduction of BITZV. We lay the groundwork so that by the end of Step 10 all V-terms will be simple V-variables; $E_v$ and $o[0..9]$ will be removed.

At this point in the algorithm all V-terms occurring in $\Omega$ are either simple V-variables, $E_v$, or of the form $v[i_1..i_2]$ where $v$ is a V-variable and $i_1$ and $i_2$ are I-terms not containing $o[0..9]$. In Step 5 we will be concerned only with the latter, specifically with questions about the definedness, equality and order of terms which, like $i_1$ and $i_2$, define the endpoints of vector segments.

Consider two terms $v[i_1..i_2]$ and $v[i_3..i_4]$. Under some assignments of values these terms may represent defined vector values, while under other assignments they represent the error vector $E_v$. Likewise, under some assignments these two terms may represent segments of the vector $v$ which are disjoint; sometimes they may represent abutting segments (where the right end of one segment meets the left end of another); sometimes they will represent overlapping or even identical vector segments. Such relations between vector terms are our central concern between now and Step 10. Let us define these relations, and others, formally.

Definition 1: Let $v$ be a V-variable, $i_1..i_4$ be I-terms and $\phi$ be an assignment of values. Then

a. $v[i_1..i_2]$ is defined under $\phi$ (i.e. $V_{\phi}(v[i_1..i_2])=true$) if

$$\phi \models i_1 \land (i_2+1) \land lb(v) \leq i_1 \land i_1 \leq i_2 + 1 \land i_2 + 1 \leq \text{sub}(v) + 1$$

b. $v[i_1..i_2]$ is empty under $\phi$ if $v[i_1..i_2]$ is defined under $\phi$ and

$$\phi \models i_1 = i_2 + 1$$

c. $v[i_1..i_2]$ and $v[i_3..i_4]$ are disjoint under $\phi$ if $v[i_1..i_2]$ and $v[i_3..i_4]$ are both defined under $\phi$, and

$$\phi \models i_2 + 1 \leq i_3 \lor i_4 + 1 \leq i_1$$
d. \( v_{[i_1..i_2]} \) abuts \( v_{[i_3..i_4]} \) under \( \varphi \) if \( v_{[i_1..i_2]} \) and \( v_{[i_3..i_4]} \) are both defined under \( \varphi \) and

\[ \varphi \equiv i_2+1=i_3 \]

e. \( v_{[i_1..i_2]} \) and \( v_{[i_3..i_4]} \) overlap under \( \varphi \) if \( v_{[i_1..i_2]} \) and \( v_{[i_3..i_4]} \) are defined under \( \varphi \) and not empty under \( \varphi \) and

\[ \varphi \equiv (i_1=i_3 \land i_3<i_2+1) \lor (i_1=i_4+1 \land i_4+1>i_2+1) \lor (i_3<i_1 \land i_1<i_4+1) \lor (i_3<i_2+1 \land i_2+1>i_4+1) \]

f. \( v_{[i_1..i_2]} \) and \( v_{[i_3..i_4]} \) are equal under \( \varphi \) if they are defined under \( \varphi \) and

\[ \varphi \equiv i_1=i_3 \land i_2+1=i_4+1 \]

Examining these definitions we find that for the terms \( v_{[i_1..i_2]} \) and \( v_{[i_3..i_4]} \) we can decide all six of the properties (defined, empty, equality, etc.) provided we know certain definedness, equality and order relations between the terms \( lb(v), ub(v)+1, i_1, i_2+1, i_3 \) and \( i_4+1 \).

[ A note to the reader: I wrote the definitions as I did, with certain obtusenesses such as \( i_2+1 \leq ub(v)+1 \) instead of \( i_2 \leq ub(v) \), in order to reduce the total number of different 1-terms involved in the definitions (a)-(f). Since the number of cases we will generate in this step is a fast-growing function of the number of distinct 1-terms in these definitions, it was important to try to avoid having both \( i_2 \) and \( i_2+1 \) appear, or both \( lb(v) \) and \( ub(v)+1 \). It was a major discovery (for me at least) to notice that the left endpoint terms \( (lb(v), i_1, i_2) \) together with the successors of the right end point terms \( (i_2+1, i_4+1 \text{ and } ub(v)+1) \) are sufficient to define the notions (a)-(f). It is not necessary to include, as I did in an earlier version, the right endpoints \( (i_2, i_4 \text{ and } ub(v)) \) or the predecessors of the left endpoints \( (lb(v)-1, i_1-1, \text{ and } i_3-1) \). Using half as many terms cuts the exponent in half in the function describing the number of cases we have to consider below, and is thus a major improvement.]

Unfortunately, a satisfiable wff \( w \) in which terms such as \( v_{[i_1..i_2]} \) and \( v_{[i_3..i_4]} \) occur usually does not contain enough constraints on \( lb(v), ub(v)+1, i_1, i_2+1, i_3 \) and \( i_4+1 \) to decide the properties (a)-(f). More precisely, it is usually the case that for some \( \varphi \) satisfying \( w \), \( v_{[i_1..i_2]} \) will overlap \( v_{[i_3..i_4]} \), and for another assignment \( \psi \) also satisfying \( w \), \( v_{[i_1..i_2]} \) and \( v_{[i_3..i_4]} \) will
be disjoint.

In this step we take a brute force approach to this problem by breaking a wff $w$ into a fairly large number of cases, one case for each way that the terms $I_b(v), u_b(v)+1, i_1, i_2+1, i_3$ and $i_4+1$ can be defined (under $=_1$) and ordered (under $=_1$ and $<_1$). For example, if the only two V-terms in a wff $w$ were $v[i_1..i_2]$ and $v[i_3..i_4]$, one of the many cases to which $w$ is reduced is the case where $=_1, _2(i_2+1), _3$ and $=_4+1$ are true, and where $lb(v) < i_1 < i_3 < i_2+1 = i_4+1$. Thus, the effect of this step will be to remove $w$ from $\Omega$ and replace it with a large number of variants, one of which is

$$lb(v) < i_1 \land i_2 < i_3 \land i_3 < i_2+1 \land i_2+1 = i_4+1 \land i_4+1 < ub(v) + 1 \land w = 1.$$  

We say "similar to" because the exact details of the algorithm we present may not always produce exactly the wff above, but it would produce one with substantially the same content.

By the end of Step 5 every wff $w \in \Omega$ will have what we have dubbed a "Y-part", that is, a conjunction of atoms containing enough information to decide (effectively) each of the relations (a)-(f) for the V-terms occurring in it. For example, in Formula 1 above, all of the wff except "w" is the Y-part. For the two V-terms which we assume occur in it, namely $v[i_1..i_2]$ and $v[i_3..i_4]$, the Y-part requires that for any $\varphi$ satisfying formula 1 $v[i_1..i_2]$ and $v[i_3..i_4]$ are both defined under $\varphi$, neither is empty under $\varphi$, they are not disjoint under $\varphi$, neither abuts the other under $\varphi$, they do overlap under $\varphi$, and they are not equal under $\varphi$.

Before we describe the Algorithm and the Specifications for this step we will have to take the time to define more precisely the concepts discussed above, to introduce some notation and to prove a few lemmas. This we do in the next few subsections before resuming the cycle of Specifications, Algorithm, Proof of Weak Correctness and Termination.

### 6.5.1 Simplification of I-terms; Index Sets

For any I-term $i$ we define the simplification of $i$, denoted $i^?$, as follows.

**Definition 2**: Let $c_1$ and $c_2$ be pattern variables which match either a non-negative integer constant symbol (0, 1, 2, 77, etc.) or the negation of such a constant (-0, -1, etc.)
Let \([c_1+c_2]\) denote the nonnegative constant, or negation of a positive constant, representing the sum of \(c_1\) and \(c_2\). Thus \([3+5]\) denotes the term 8 and \([(-3)+(-5)]\) denotes the term -8.

Let \(x\), \(x_1\), and \(x_2\) be pattern variables denoting any 1-term.

Then \(i^o\) is the result of applying the following TRS to the term \(i\).

\[
\begin{align*}
&\text{do } \begin{cases}
-0 &\Rightarrow 0 \\
-x+x &\Rightarrow x \\
-x_1-x_2 &\Rightarrow x_1 + (-x_2) \\
-(x_1+x_2) &\Rightarrow (-x_1) + (-x_2) \\
[x_1+x_2] &\Rightarrow [c_1 + c_2] \\
(x + c_1) + c_2 &\Rightarrow x + [c_1 + c_2] \\
(c_1 + x) + c_2 &\Rightarrow x + [c_1 + c_2] \\
(c_2 + x + c_1) &\Rightarrow x + [c_1 + c_2] \\
(c_2 + (c_1 + x) &\Rightarrow x + [c_1 + c_2]
\end{cases}
\end{align*}
\]

\(\text{od} \)

We will not bother to prove any formal properties of \(i^o\). It will be sufficient to note informally that \(i^o\) is the result of removing additive zeros from the term \(i\), removing double negation, replacing subtraction with negation, and, most importantly, performing some additive constant arithmetic on the term \(i\).

The simplification \(i^o\) puts the term \(i\) in a kind of standard form, but there is very little that is standard about it. For example, certain opportunities for combining constants are ignored. If \(i\) is the term \((1+k) + (j+2)\) then \(i^o\) is simply \(i\). The TRS given above will not transform \(i\) to \((k+j)+3\). The TRS for \(i^o\) can be expanded with more rules to make \(i^o\) more nearly canonical than it is with this definition, and the result would be improved. However, for our purposes the following properties are crucial for the simplification operation, and any simplification operation substituted for the one above must have at least these properties.

**Lemma 3:**
1. $i^o = i$, for any $I$-term $i$.

2. $(i^o + 1)^o$ is the same term as $(i + 1)^o$ and $(i^o - 1)^o$ is the same term as $(i-1)^o$, for any $I$-term $i$.

3. $((i + 1) - 1)^o$ is the same term as $i^o$, and $((i - 1) + 1)^o$ is the same term as $i^o$, for any $I$-term $i$. □

Lemma 4:

1. $lb(v)^o$ is the same term as $lb(v)$;

2. $(lb(v) - 1)^o$ is the same term as $lb(v) - 1$;

3. $ub(v)^o$ is the same term as $ub(v)$;

4. $(ub(v) + 1)^o$ is the same term as $ub(v) + 1$. □

We leave these lemmas without proof.

With these properties of the simplification operation established we can now define the Index Set associated with a wff $w$ and a variable $v$.

Let $w$ be a wff satisfying the Precondition for this Step 5, in particular the requirement that all $V$-terms are either $E_v$ or a simple variable or of the form $v[i_1..i_2]$ where $v$ is a simple variable, and also satisfying the requirement that $^o[0..0]$ does not occur nested. Then we define the set of terms $IX(w,v)$ as follows.

**Definition 5:** If $v[i_1..i_2]^o w$ (for a variable $v$) then the terms $(lb(v))^o$, $i_1^o$, $i_2^o + 1)^o$ and $(ub(v) + 1)^o$ are all members of $IX(w,v)$. No other terms are members of $IX(w,v)$. □

The set $IX(w,v)$ contains (the simplifications of) all $I$-terms occurring in a left-endpoint position in the wff $w$, such as $i_1^o$, and (the simplifications of) the successors of all of the $I$-terms occurring in a right-endpoint position, such as $(i_2 + 1)^o$. We include the terms $lb(v)$ (which is the same as $lb(v)^o$) and $ub(v) + 1$ (which is the same as $(ub(v) + 1)^o$) as well; they are "implicit" endpoints. Notice that $IX(w,v)$ is empty if no term of the form $v[i_1..i_2]$ occurs in $w$. 
Sometimes we will have occasion to refer to the set

\[ IX(w) \triangleq IX(w,v_1) \cup \ldots \cup IX(w,v_n) \]

where \( v_1 \ldots v_n \) is a list of all \( V \)-variables occurring in \( w \).

The significance of the definition of \( IX(w,v) \) is that for any two vector terms \( v[i_1..i_2] \) and \( v[i_3..i_4] \) occurring in \( w \), if we know how the elements of \( IX(w,v) \) behave under the predicates \( \#_1, =_1 \) and \( <_1 \), then we can decide how the terms \( v[i_1..i_2] \) and \( v[i_3..i_4] \) behave under the definedness, emptiness, disjointness, abutting, overlapping and equality relations. The reason we have included only simplified terms (simplified in the sense of the \( \circ \)-operator) in \( IX(w,v) \) is to make the set closed under the operation of subtracting 1 and then adding 1 (or adding 1 and then subtracting 1). The terms

\[ i, (i+1)-1, (i-1)+1, (((i+1)-1)+1)-1, ((((i+1)-1)+1)+1) \ldots \]

are all distinct terms, and there are an infinite number of them. But the terms

\[ i^0, ((i+1)-1)^0, ((i-1)+1)^0, (((i+1)-1)+1)^0 \ldots \]

are all the same term, namely \( i^0 \). This fact will be crucial in Step 9.

One further fact about \( IX(w,v) \) is essential. Because at this stage in the algorithm the function \( o^0[0..0] \) does not occur nested in \( w \), it cannot occur at all in any of the terms in \( IX(w,v) \).

6.5.2 Index Constraint Sets and Index Constraint Formulae

We are concerned with their behavior of the set \( IX(w,v) \) and the functions \( \#_1, =_1 \) and \( <_1 \). In order to describe their behavior we define the notion of an Index Constraint Set.

**Definition 6**: An Index Constraint Set (ICS) for wff \( w \) and variable \( v \), which will be denoted \( Y \), is a set of formulae over the terms of \( IX(w,v) \) with the following properties:

1. All elements of \( Y \) are of the form \( \neg i = j \) or \( i < j \) for some \( i,j \in IX(w,v) \).

2. If \( \neg i \not\in Y \) and \( \neg j \not\in Y \) then at least one of \( i < j \) or \( i = j \) or \( j < i \) holds.

3. If \( i \not\in Y \) or \( i < j \) then \( \neg j \not\in Y \) and \( \neg i \not\in Y \).
4. At most one of \( i < j \in I \) or \( i = j \in I \) or \( j < i \in I \) holds.

5. If \( \text{IX}(w,v) \neq \phi \) then \(-\#\text{lb}(v) \notin I \) and \(-\#(\text{ub}(v)+1) \notin I \).

6. If \( \text{IX}(w,v) \neq \phi \) then \( \text{lb}(v) < \text{ub}(v) + 1 \in I \) or \( \text{lb}(v) = \text{ub}(v) + 1 \in I \).

7. If \(-\#i \notin I \) where \( i \in \text{IX}(w,v) \), then \( i = i \in I \).

8. If \( i = j \in I \) then \( j = i \in I \).

9. If \( i = j \in I \) and \( j = k \in I \) then \( i = k \in I \).

10. If \( i < j \in I \) and \( j = k \in I \) then \( i < k \in I \).

11. If \( i = j \in I \) and \( j < k \in I \) then \( i < k \in I \).

12. If \( i < j \in I \) and \( j < k \in I \) then \( i < k \in I \).

The style of this definition is patterned (loosely) after the definition of a Truth Set or of a Hintikka Set as given in [Smullyan 71]. The motivation for the definition is this: if we assume that all of the elements of some ICS (which we will call \( Y \)) are true under some valuation \( V_{\phi} \), then we can compute very simply the truth value under \( V_{\phi} \) of any propositional combination of elements of \( Y \). We will prove this later.

One detail to notice is that if \( \text{IX}(w,v) \) is empty, then there is only one ICS for \( w,v \): the empty set.

**Definition 7:** Let \( \text{ICS}(w,v) \) be the set of all Index Constraint Sets for wff \( w \) and variable \( v \).

Individual ICS's (members of \( \text{ICS}(w,v) \)) will be denoted by \( Y \), or by \( Y(w,v) \) when the dependence on \( w \) and \( v \) needs emphasis.

Here is an example. Consider the wff \( w \)

\[ \{ v[\text{lb}(v),i] \} = \phi \text{.} \]

This wff satisfies the Precondition to Step 5, and is therefore eligible for our discussion. The Index set for \( w,v \) is
\[ IX(w, v) = \{ lb(v), i+1, ub(v)+1 \} \]

The terms in \( IX(w, v) \) are in simplified form since \((i+1)^{\circ} \) is the same term as \( i+1 \). (Note that in this example \( i \) is an I-variable, not a metavariable standing for some generic I-term.)

There are several ICS’s for \( w \) and \( v \). Here are three examples.

\[
Y_1 = \{ lb(v)=lb(v), i+1=i+1, ub(v)+1=ub(v)+1, lb(v)=i+1, \\
i+1=ib(v), i+1<ub(v)+1, lb(v)<ub(v)+1 \}
\]

\[
Y_2 = \{ lb(v)=ib(v), i+1=i+1, ub(v)+1=ub(v)+1, lb(v)<i+1, \\
i+1<ub(v)+1, lb(v)<ub(v)+1 \}
\]

\[
Y_3 = \{ -a(i+1), lb(v)=lb(v), ub(v)+1=ub(v)+1, lb(v)<ub(v)+1 \}
\]

Informally, \( Y_1 \) corresponds to the possibility that \( i+1 \) takes a defined value and that \( lb(v) = i+1 < ub(v)+1 \). Since \( i \) is out of bounds of the vector \( v \), the term \( v[ib(v).i] \) must be undefined (under any \( \varphi \) satisfying \( Y_1 \)).

\( Y_2 \) corresponds to the possibility that \( i+1 \) is defined and within the bounds of \( v \). Since \( lb(v)<i \), the term \( v[ib(v).i] \) is defined and nonempty under any \( \varphi \) satisfying \( Y_2 \).

\( Y_3 \) corresponds to the "possibility" that vector \( v \) is itself non-empty, but the term \( i+1 \) is undefined and hence the term \( v[ib(v).i] \) is undefined for any \( \varphi \) satisfying \( Y_3 \). (Note: \( Y_3 \) is unsatisfiable. The term \( i+1 \) cannot be undefined under any \( \varphi \) since variables are never assigned undefined values. Therefore, the generalization that \( v[ib(v).i] \) is undefined under all \( \varphi \) satisfying \( Y_3 \) is true, but vacuous.)

We will now prove some lemmas and theorems about ICS’s and related notions.

Lemma 8: \( IX(w, v) \) is a finite set.

Proof: Trivial. Wff \( w \) is of finite-length, and thus the set of all terms of the form \( v[i_1..i_2] \) occurring in \( w \) is finite. The set including \( lb(v), ub(v)+1 \) and all terms such as \( i_1^{\circ} \) and \( i_2^{\circ} \) is clearly finite, and simplifying to \( i_1^{\circ} \) and \( (i_2+1)^{\circ} \) can only make the set smaller. \( \square \)

Lemma 9: Any \( Y \in ICS(w, v) \) is finite.
Proof: Trivial. IX(w,v) is finite, and Y is a set of terms of the forms ¬#i, i=j and i<j where i,j∈IX(w,v). There can only be a finite number of such terms. □

Lemma 10: The members of Y, where Y∈ICS(w,v), need not be simultaneously satisfiable.
Proof: As pointed out above, Y₃ is not satisfiable. Another example is this. Suppose the only V-term occurring in w is v[1..5]. One ICS for w,v is
\{¬#1, ¬#6, lb(v)=lb(v), ub(v)+1=ub(v)+1, lb(v)<ub(v)+1\}.

Another is
\{lb(v)=lb(v), ub(v)+1=ub(v)+1, lb(v)<1,
   lb(v)<6, lb(v)<ub(v)+1, 6<1, 6<ub(v)+1, 1<ub(v)+1\}.

Neither is satisfiable, the first because of the inclusion of ¬#1 and ¬#6, and the second because of 6<1. Note that 6 is involved because it is (5+1)°. □

We are now ready to establish a strong connection between the concept of an Index Set and the notion of an ICS. The important point is that for any w and v, and any assignment φ, the set of wffs ¬#i, i=j and i<j (for i,j∈IX(w,v)) which are true under Vₜ is an ICS. Therefore, the set ICS(w,v), which we will later show is finite, bounds the set of behaviors that index terms display under the = and < functions.

Definition 11: Let ICU(w,v), the index constraint universe for a wff w, be the set of all terms of the forms ¬#i, i=j, or i<j for i,j∈IX(w,v).

Thus, every member of ICS(w,v) is a subset of ICU(w,v).

Definition 12: For any assignment φ, let Yᵦₜₜ(w,v) be the set of all terms a∈ICU(w,v) such that Vₜ(a)=true.

Theorem 13: For any assignment φ, wff w, and V-variable v, the set Yᵦₜₜ(w,v) is an ICS.
Proof: We examine one by one the clauses in the definition of an ICS by see if they are satisfied. We abbreviate Yᵦₜₜ(w,v) by Y.

1. The first clause holds by the definition of Y and of ICU(w,v).
2. If \(-\#i \notin Y\) and \(-\#j \notin Y\), then the reason must be that \(-\#i\) and \(-\#j\) are not true under \(V_\varphi\). Thus, \#i and \#j are true under \(V_\varphi\), and hence (at least) one of \(i < j\), \(i = j\) and \(j < i\) must be true and must be in \(Y\).

3. If \(i = j \in Y\) or \(i < j \in Y\) then one of \(i = j\), \(i < j\) must be true under \(V_\varphi\), and thus \#i and \#j are true under \(V_\varphi\). We conclude that \(-\#i\) and \(-\#j\) are not true under \(V_\varphi\), and are thus not members of \(Y\).

4. At most one of \(i < j\), \(i = j\), \(j < i\) can be true under \(V_\varphi\), and thus at most one is a member of \(Y\).

5. Under any assignment \(\varphi\), \(V_\varphi(\#lb(v)) = \text{true}\) and \(V_\varphi(\#(ub(v)+1)) = \text{true}\). Hence \(\#lb(v) \in Y\) and \(\#(ub(v)+1) \in Y\). This, and the remaining arguments can be justified merely by citing certain valid schema of BITZV, and we will do so from now on. For the current clause the schema are \(E \#lb(v)\) and \(E \#(ub(v)+1)\), valid for all assignments \(\varphi\) and all \(V\)-variables.

6. \(E \#lb(v) < \#(ub(v)+1) \lor \#lb(v) = \#(ub(v)+1)\), for all \(V\)-variables \(v\)

7. \(E \#i \geq \#i\), for all terms \(i\).

8. \(E \#i = \#j\), for all terms \(i, j\).

9. \(E (i = j \land j = k) \geq \#i = \#k\), for all terms \(i, j, k\).

10. \(E (i < j \land j = k) \geq \#i < \#k\), for all terms \(i, j, k\).

11. \(E (i = j \land j < k) \geq \#i < \#k\), for all terms \(i, j, k\).

12. \(E (i < j \land j < k) \geq \#i < \#k\), for all terms \(i, j, k\).

Clearly then, \(Y_\varphi(w, v)\) satisfies all of the requirements for membership in \(ICS(w, v)\). \(\Box\)

The next theorem is quite important for subsequent development. It says essentially that an Index Constraint Set \(Y(w, v)\) actually contains complete information about the definedness, equality and order of the elements of \(IX(w, v)\). In other words, if some assignment \(\varphi\) satisfies all of the constraints (wfts) in \(Y(w, v)\), then we can compute straightforwardly the (three-valued) truth value of any propositional combination of atoms constructed from \(\#1, =_1, <_1\) and the terms in \(IX(w, v)\). And, since the defining formulae for definedness, emptiness, ..., equality of \(V\)-terms are all in that class, the truth or falsity of such relations is determined by \(Y(w, v)\).
Definition 14: Let $A(w, v)$ be the set of all Boolean terms (atomic formulae) constructable from the I-terms in $I(X(w, v))$ and the symbols $\#_1, =_1$ and $<_1$. □

Definition 15: Let $L_{IX}(w, v)$ be the language of all unquantified formulae constructable from the atoms in $A(w, v)$ and the three-valued propositional connectives (function symbols mapping $B^3 \rightarrow B$). □

Lemma 16: The formulae defining the notions of definedness, emptiness, disjointness, abutting, overlapping and equality (under an assignment $\varphi$, for V-terms based on variable $v$ and occurring in wff $w$) are all members of $L_{IX}(w, v)$.

Proof: Trivial by inspection. □

Theorem 17: (Complete Information Theorem): A satisfiable ICS for a wff $w$ contains complete information about the truth values of all wffs in $L_{IX}(w, v)$. In other words, if $\varphi$ and $\psi$ are two assignments each satisfying all elements of an ICS $Y(w, v)$, then $V_\varphi$ and $V_\psi$ assign the same truth values to all formulae in $L_{IX}(w, v)$.

Proof: We show that if $\varphi$ and $\psi$ satisfy $Y(w, v)$, then $V_\varphi$ and $V_\psi$ must agree on all of the atoms in $L_{IX}(w, v)$. The truth value of all wffs in $L_{IX}(w, v)$ can then be computed by truth table, thereby assuring agreement of $V_\varphi$ and $V_\psi$.

Suppose $a \in A(w, v)$, i.e. an atom in $L_{IX}(w, v)$. Then $a$ must be of one of the forms $\#_i$, $i=j$ or $i<j$ for some $i, j \in I(X(w, v))$. We will show, for all three cases, that $V_\varphi$ and $V_\psi$ agree on the truth value they assign to $a$.

Case 1: $a$ is $\#_i$. Either $-\#_i \in Y$ or $-\#_i \notin Y$. If $-\#_i \notin Y$, then both $V_\varphi$ and $V_\psi$ evaluate $-\#_i$ to false, and hence agree that $\#_i$ is true. On the other hand, if $-\#_i \in Y$, then by Clause 2 of the definition of ICS, either $i=i$ or $i<i \in Y$. Thus $V_\varphi$ and $V_\psi$ both evaluate one of $i=i$ or $i<i$ to true, and thus must also evaluate $\#_i$ to true, hence agreeing again.

Case 2: $a$ is $i=j$. If $i=j \in Y$ then $V_\varphi$ and $V_\psi$ both assign true to it and thus agree. But if $i=j \notin Y$ then there are two possibilities according to Clause 2 of the definition of ICS: either $-\#_i$ and/or $-\#_j$ are in $Y$, or $-\#_i$ and $-\#_j$ are not in $Y$, but $i<j$ or $j<i$ (or both) are. We consider these possibilities in turn.

Case 2a: Suppose, without loss of generality, that $-\#_i \notin Y$. Then $V_\varphi$ and $V_\psi$ assign true to $-\#_i$, hence they must both assign error to $i=j$, thereby agreeing.

Case 2b: Assume, without loss of generality, that $-\#_i \in Y$, $-\#_j \notin Y$ and $i<j \in Y$. Then $V_\varphi$ and $V_\psi$ assign true to $i<j$, and must thus assign false to $i=j$, thereby agreeing.

Case 3: $a$ is $i<j$. This proof is nearly the same as that for case 2, and will not be repeated. □
One thing to notice is that the proof of this theorem uses only clauses 1, 2 and 3 of the definition of ICS, and although previous theorems have "used" or "mentioned" the other clauses, they did not need them. In fact, if all but clauses 1-3 of the definition were deleted, very little of this thesis would have to be changed. We include the other clauses in order to reduce the size of ICS(w,v), since that size determines the number of cases into which we must break each wff of Ω here in Step 5. We will describe this further below, but we must first introduce a little more theory and notation.

**Lemma 18:** For each w,v, ICS(w,v) is finite.

*Proof:* The number of sets of wffs satisfying Clause 1 alone is finite for each w,v. The remaining clauses only reduce the number of sets satisfying the definition.

We now know that IX(w,v) is finite, that each Y(w,v)∈ICS(w,v) is finite, and that ICS(w,v) itself is finite. We can in fact bound the size of these sets, and the next theorem gives a bound on the size of ICS(w,v) in terms of the size of IX(w,v). The bounding function is written in terms of binomial coefficients (denoted \(\binom{n}{k}\)), Stirling numbers of the second kind (denoted \(\{n\}_k\)), and factorials (denoted \(n!\)).

**Theorem 19:** Let \(n\) be the size of IX(w,v). Then the size of ICS(w,v) is less than or equal to \(ψ(n)\), where

\[
ψ(n) = \sum_{0≤k≤m≤n} \binom{n}{m} \{m\}_k k!
\]

*Proof Sketch:* If we ignore Clause 5 of the definition of an ICS, then any subset of IX(w,v) may be that subset of all terms ∈IX(w,v) such that \(\neg i\not\equiv Y\). There are \(\binom{n}{m}\) ways to choose a subset of size exactly \(m\) terms out of the \(n\) terms in IX(w,v).

Of the \(m\) terms such that \(\neg i\not\equiv Y\), clauses 7, 8 and 9 imply that the relation \(i=j\in Y\) (for \(i\in Y\), \(j\in Y\)) is an equivalence relation. There are exactly \(\{m\}_k\) ways to divide \(m\) terms into \(k\) equivalence classes.

By clauses 2, 4, 10, 11 and 12 we see that the relation \(i<j\in Y\) induces a total ordering on the equivalence classes just described. For \(k\) equivalence classes there are exactly \(k!\) total orders that can be imposed.

Putting all of this together we conclude that the number of sets of wffs satisfying clauses 1-4 and 7-13 is

\[
ψ(n) = \sum_{0≤k≤m≤n} \binom{n}{m} \{m\}_k k!
\]
When we then consider clauses 5 and 6 as well, the number of combinations is further reduced, accounting for the phrase "less than or equal" in the statement of the theorem, rather than "equal".

Theorem 17 and its proof suggest an actual algorithm for determining the truth value of any wff \( \varphi \in L_{\chi}(w,v) \) under some assignment \( \varphi \), given that \( \varphi \) satisfies some \( YEICS(w,v) \). We describe that shortly.

**Definition 20:** Let \( YEICS(w,v) \) and \( u \in L_{\chi}(w,v) \). Define \( Y \upharpoonright u \) to mean "for all \( \varphi \), if \( \varphi \) satisfies all elements of \( Y \), then \( \varphi \) satisfies \( u \)."

This is not the usual use of the \( \vdash \) symbol. It is not any syntactic notion like "derivable from"; there is no inference system or proof theory here. The relation \( \upharpoonright \) is semantic. However, the relation has many of the metatheoretic properties of a derivability relation, so little confusion will arise in the notation.

**Theorem 21:** The following is an algorithm for deciding the relation \( Y \upharpoonright u \) for \( YEICS(w,v) \) and \( u \in L_{\chi}(w,v) \).

Define the function \( BB \) (a Boolean valuation) as follows.

\[
\begin{align*}
BB(\#i) &= \text{true} \quad \text{if} \quad \neg i \notin Y \\
BB(i) &= \text{false} \quad \text{if} \quad \neg i \notin Y \\
BB(i) &= \text{error} \quad \text{never}
\end{align*}
\]

\[
\begin{align*}
BB(i=j) &= \text{true} \quad \text{if} \quad i=j \in Y \\
BB(i=j) &= \text{false} \quad \text{if} \quad \neg i \notin Y, \neg j \notin Y \quad \text{and} \quad i \neq j \in Y \\
BB(i=j) &= \text{error} \quad \text{if} \quad \neg i \notin Y \text{ or } \neg j \notin Y
\end{align*}
\]

\[
\begin{align*}
BB(i<j) &= \text{true} \quad \text{if} \quad i<j \in Y \\
BB(i<j) &= \text{false} \quad \text{if} \quad \neg i \notin Y, \neg j \notin Y \quad \text{and} \quad i<j \notin Y \\
BB(i<j) &= \text{error} \quad \text{if} \quad \neg i \notin Y \text{ or } \neg j \notin Y.
\end{align*}
\]

\( BB(u) \) where \( u \) is not an atom, is the result of applying the truth tables for the propositional connectives in \( u \) to the results of \( BB \) applied to the atoms in \( u \).
Then $Y\text{-}u$ iff $BB(u)$ is true.

**Proof:** Derived from that for Theorem 17.

The importance of this theorem is that it makes the relation $Y\text{-}u$ recursive (effectively computable). Because of this we are free to use the $Y\text{-}u$ relation in the pattern parts of productions and still be guaranteed that the production is implementable (i.e. can be made effective). We will make use of this fact often, because our primary motivation for defining $Y\text{-}u$ is to use it (and extensions of it) to make our patterns more compact in the next several steps.

So far we have been dealing with Index Constraint sets. We actually want to deal with wffs expressing of the same information as an ICS.

**Definition 22:** An Index Constraint Formula for wff $w$ and $V$-variable $v$ is a conjunction (using $\land$, rather than $\text{and}$) of the elements of some $Y\in\text{ICS}(w,v)$. If an ICS is empty (as it will be if $v$ does not occur in $w$) then the corresponding ICF is the wff true.

**Definition 23:** $\text{ICF}(w,v)$ is the set of all distinct ICFs (up to commutativity and associativity of $\land$) constructed by conjunction from the sets in $\text{ICS}(w,v)$.

From now on we will hardly distinguish between an ICS and an ICF. We will denote an ICF by $y$ (or $y(w,v)$), in lower case, and we will extend the notation $Y\text{-}u$, where $Y$ is an ICS, to permit $y\text{-}u$ where $y$ is an ICF.

Since the set $\text{ICF}(w,v)$ contains only a finite number of ICF's we can write a wff expressing the disjunction of all the elements. Such a wff must be valid, as expressed in the following theorem.

**Theorem 24:** Let $\text{ICF}(w,v)$ be a collection $\{y_1, \ldots, y_n\}$ of ICFs. Then

$$\models \bigvee_{1 \leq i \leq n} (y_i).$$

**Proof:** By Theorem 13 the set of terms of the form $\neg w_i$, $i=j$ or $i<j$ for $i,j \in X(w,v)$ which is
satisfied by an assignment \( \varphi \) is an ICS. All possible ICS's are represented as ICF's among the wff's \( y_1 \ldots y_n \). Hence, any \( \varphi \) satisfies one (at least) of the \( y_i \) and thus the disjunction is valid. 

We now study some of the formal properties of \( \mathcal{H} \).

**Lemma 25:** If \( y \in \text{ICF}(w,v) \), and \( u_1, u_2 \in \text{IX}(w,v) \) then

\[
y \cdot u_1 \land u_2 \iff y \cdot u_1 \land y \cdot u_2, \quad \text{and} \quad y \cdot u_1 \lor u_2 \iff y \cdot u_1 \lor y \cdot u_2.
\]

**Proof:** Follows from Theorem 17 and Theorem 21. 

Before we begin describing the algorithm of this step there are two other lemmas we must mention. The main effect of Step 5 will be to substitute for each wff \( w \in \Omega \) the collection of all wffs of the form \( y \cdot w \) where \( y \in \text{ICF}(w,v) \). We need to show that this operation of conjoining \( y \) with \( w \) has no effect on the index set \( \text{IX}(w,v) \).

**Lemma 26:** If \( y \in \text{ICF}(w,v) \) then \( \text{IX}(y \cdot w,v) = \text{IX}(w,v) \).

**Proof:** None of the terms in \( \text{IX}(w,v) \) contain any occurrence of \( 0^{[]} \cdot 0^{[]} \). The conjunct \( y \), therefore, neither adds any terms nor deletes any from \( \text{IX}(w,v) \). 

The operation also has no effect, therefore, on the set \( \text{ICS}(w,v) \).

**Lemma 27:** Let \( y_1 \in \text{ICF}(w,v_1) \) for some wff \( w \) and variable \( v_1 \). Then, for any \( V \)-variable \( v_2 \) (including \( v_1 \))

\[
\text{ICS}(y_1 \cdot w,v_2) = \text{ICS}(w,v_2), \quad \text{and} \quad \text{ICF}(y_1 \cdot w,v_2) = \text{ICF}(w,v_2).
\]

**Proof:** Tacking \( y_1 \) onto \( w \) leaves the Index Set for \( w \) and \( v_2 \) invariant (by the preceding lemma). Hence, the ICS and ICF (formed only from the Index Set) is unchanged. 

6.5.3 Summary of Step 5

Before we present Step 5 formally, we will explain it informally. The idea is to replace each wff $w \in \Omega$ with a (possibly large) finite set of wffs $\{w_1 \ldots w_n\}$ having two important properties:

1. Mutual unsatisfiability: $\neg w \iff \neg \{w_1 \ldots w_n\}$
2. Each $w_i$ has a $Y$-part with complete information about the definedness and order of the terms in $I_X(w_i)$.

By now the reader has probably guessed how we do this.

Suppose $w \in \Omega$ has only one $V$-variable occurring in it as an argument to $\circ[\circ \ldots \circ]$ (but occurring possibly several times). We first compute the set of wffs $ICF(w,v) = \{y_1 \ldots y_n\}$, which we know is a computable finite set. Then we form

$$w_1 \triangleq y_1 \land w$$
$$w_2 \triangleq y_2 \land w$$
$$\ldots$$
$$w_n \triangleq y_n \land w$$

The set $\{w_1 \ldots w_n\}$ is unsatisfiable (i.e., for each $1 \leq i \leq n$, $\neg w_i$) if and only if $w$ is unsatisfiable simply because

$$\models w \equiv (w_1 \lor w_2 \lor \ldots \lor w_n).$$

We prove Lemma 27 as follows:

$$\models w \equiv (w \land \text{true})$$
$$\models w \equiv (w \land (y_1 \lor \ldots \lor y_n))$$
$$\models w \equiv ((w \land y_1) \lor \ldots \lor (w \land y_n))$$
$$\models w \equiv (w_1 \lor \ldots \lor w_n).$$

The first step in this derivation is justified by Lemma 24.

Also, by the definition of $ICF(w,v)$ and Theorem 17 and Lemma 27 we know that $y_i \land w$
contains complete information about the definedness and order of the terms in $IX(y_1, w, v)$. We can easily extend this property so that each wff $w_i$ contains complete information about all terms in $IX(w_i)$ (which is the union of $IX(w_i, v_j)$ for all variables $v_j$ occurring in $w_i$).

Thus, at the end of the procedure, a wff $w \in \Omega$ containing occurrences of variables $v_j$, $1 \leq j \leq m$, will be replaced in $\Omega$ by a set of wffs of the form

$$y_{1,1} \wedge y_{1,2} \wedge \ldots \wedge y_{1,m} \wedge w$$

$$\ldots$$

$$y_{n(1),1} \wedge y_{n(2),2} \wedge \ldots \wedge y_{n(\ell),m} \wedge w$$

where $n(j)$ is the size of the set $ICF(w_j)$. If we call these formulae $w_1 \ldots w_n$, then as a collection they have the required two properties.

We now proceed with Step 5 of the algorithm and its correctness proof.

6.5.4 Specifications

**Precondition:** Same as Postcondition for Step 4.

**Postcondition:**

1. $\Omega \in \text{BITZV}$ with $\text{elem} \in \Omega$ quantifier-free and finite.
2. $w_0 \iff \not \emptyset$.
3. The only $V$-relevant functions occurring in $\Omega$ are $\equiv_v$, $\text{ord}$, $\text{lb}$, $\text{ub}$, $\text{elem}$, $\{0\}$, $\text{E}_v$ and $\emptyset$. [0.0]
4. All $V$-args to $\text{lb}$, $\text{ub}$ and $\emptyset$ are $V$-vars
5. All $V$-args to $\emptyset$, $\text{ord}$, $\text{elem}$ and $\{0\}$ are either $\text{E}_v$ or are of the form $\text{vvar}[i_1, i_2]$.
6. All occurrences of $\text{elem}$ are in terms of the form $\text{elem}(E_v)$ or $\text{elem}(\text{vvar}[i_1])$.
7. The function $\emptyset$ does not occur nested.
8. All wffs $w \in \Omega$ can be rearranged to be of the form

$$Y \wedge U \wedge \text{EQ} \wedge \text{ORD}$$
where

- $Y$ is a conjunction of $y_1 \ldots y_n$, where each $y_i \in \text{ICF}(w, v_i)$
  and where $v_1 \ldots v_n$ are all the $V$-variables occurring in $\Omega$.
- $U$ contains no occurrences of $v_1$ or $\text{ord}$.
- $\text{EQ}$ is a conjunction of $v_1$-atoms.
- $\text{ORD}$ is a conjunction of $\text{ord}$-atoms.

6.5.5 The Algorithm

For each $V$-var $v \in \Omega$ (in any order) perform the following
transformation on $\Omega$:

\begin{verbatim}
begin
  \Lambda := \Omega;
  For each $w \in \Omega$ do
    \Lambda := (\Lambda - \{w\}) \cup \{y \land u \mid y \in \text{ICF}(w, v)\};
  \Omega := \Lambda
end
\end{verbatim}

6.5.6 Weak Correctness

If there are $n$ $V$-variables $v_1 \ldots v_n$ occurring in $\Omega$ then there will be $n$
transformations composed on $\Omega$, and afterwards each $w \in \Omega$ will be of the form

$y_n \land \ldots \land y_1 \land w$

where $w$ was in $\Omega$ before this step and the $y_1 \ldots y_n$ were conjoined one at a time in the $n$
transformation stages.

We will establish the Postcondition informally by establishing each of its conjuncts, relying
heavily on the discussion in Sections 6.5.1-6.5.3.

First, $\Omega$ remains quantifier-free and a subset of $\text{BITZV}$ because each $y \in \text{ICF}(w, v)$ is
quantifier-free; no quantifier or operator outside of $\text{BITZV}$ is even transiently introduced in
this stage. $\Omega$ remains finite because it is constructed by a finite number of unions and
differences of finite sets. Hence, Conjunct 1 of the Postcondition is established.

Conjunct 2 of the Postcondition is invariant because each iteration of each stage replaces \( w \in \Omega \) by \( \{ y \land w \mid y \in ICF(w,v) \} \). We need only prove that

\[
\forall w \iff \exists \{ y \land w \mid y \in ICF(w,v) \}
\]

which follows from Lemma 24 and the discussion in Subsection 6.5.3.

Conjuncts 3, 4, 5, 6 and 7 of the Postcondition are part of the Precondition and are clearly invariant throughout each iteration.

Conjunct 8 of the Precondition says that each \( w \in \Omega \) can be rearranged to be of the form

\[
U \land EQ \land ORD.
\]

This fact is obviously invariant over each outer iteration of the algorithm. At termination each \( w \in \Omega \) can be rearranged to be of the form

\[
y_n \land \cdots \land y_1 \land U \land EQ \land ORD
\]

which fits the required pattern for Conjunct 8 of the Postcondition, where the \( Y \)-part is \( y_n \land \cdots \land y_1 \).

6.5.7 Termination

The termination argument is trivial. Only a finite number of \( V \)-vars occur in \( \Omega \), and no new ones are introduced. Hence, the number of iterations of the outer loop is bounded. Also, at each stage there are only a finite number of wffs \( w \in \Omega \), so only a finite number of iterations of the inner loop can occur. Recall that \( ICF(w,v) \) is finite for all \( w \) and \( v \).
6.6 Step 6: Collapse undefined $V$-terms to $E_v$

We are now at a stage where each $wff$ contains explicit information about the definedness of all of the endpoints of vector terms, and also explicit information about the ordering of those endpoints with respect to $\leq_1$ and $<_1$.

It is now the case that for any $V$-term $v[i_1..i_2]$ in $wff$ $w$, either $v[i_1..i_2]$ is defined for all assignments which satisfy $w$, or it is undefined for all such assignments, and we can tell which by examining the $Y$-part of $w$. As a result, we can replace those terms which are always undefined with the constant $E_v$.

6.6.1 Specifications

**Precondition:** Same as Postcondition of previous step.

**Invariant:** Same as Postcondition for previous step.

**Postcondition:**

1. $\Omega \subseteq \text{BITZV}$ with elem; $\Omega$ quantifier-free and finite.
2. $\neg w_0 \iff \not\exists \Omega$.
3. The only $V$-relevant functions occurring in $\Omega$ are =, ord, lb, ub, elem, $\{^0\}$, $E_v$, and $o[.\ldots]$.
4. All $V$-args to lb, ub and $o[.\ldots]$ are simple $V$-variables.
5. All $V$-args to $=$, ord, elem and $\{^0\}$ are either $E_v$ or of the form $v\text{var}[i_1..i_2]$.
6. All occurrences of elem are in terms of the form elem($E_v$) or elem($v\text{var}[i..i]$).
7. The function $o[.\ldots]$ does not occur nested.
8. All $w \in \Omega$ can be rearranged to be of the form

   $Y \land U \land EQ \land ORD$

   where $Y$, $U$, $EQ$ and $ORD$ are as before.
9. For all $w \in \Omega$, $v \in w$, $i \in \text{IX}(w,v)$: $Y_w^i = i^o$. 
10. For all \( w \in \Omega \), \( v[i..j] = w \): 
\[
Y_w^H(\text{lb}(v)i_1^o \land i_2^o \leq (j + 1)^o \land (j + 1)^o \leq \text{ub}(v) + 1).
\]

6.6.2 The Algorithm

For each \( w \in \Omega \) perform the following transformation:

\[
\text{do}
\]

(a) \( v[i_1..i_2] = w \) and \( Y_w^H(-\#i_1^o \lor -\#(i_2 + 1)^o) \)
\[
\Rightarrow w \leftarrow \text{REPL}(w, v[i_1..i_2] : = E_v)
\]

(b) \( v[i_1..i_2] = w \) and \( Y_w^H\text{lb}(v) > i_1^o \lor i_1^o > (i_2 + 1)^o \lor (i_2 + 1)^o > \text{ub}(v) + 1 \)
\[
\Rightarrow w \leftarrow \text{REPL}(w, v[i_1..i_2] : = E_v)
\]

\text{od}

6.6.3 Proof of Weak Correctness

In case the reader has forgotten the notation, the production

\[
v[i_1..i_2] = w \text{ and } Y_w^H(-\#i_1^o \lor -\#(i_2 + 1)^o)
\]
\[
\Rightarrow w \leftarrow \text{REPL}(w, v[i_1..i_2] : = E_v)
\]

can be read "if something of the form \( v[i_1..i_2] \) occurs in \( w \) and \( Y_w^H(-\#i_1^o \lor -\#(i_2 + 1)^o) \) then replace \( v[i_1..i_2] \) by \( E_v \) in \( w \)."

Conjuncts 1, 3, 4, 6 and 7 of the Postcondition are all true at termination because they are conjuncts of the Precondition and Invariant, and are trivially invariant across both productions.

Conjunct 2, \( \not\in \Omega \Leftrightarrow E_w^0 \), is invariant because the two productions are based on the following identities:

(a) \( \not\#i_1^o \land A(v[i_1..i_2]) \equiv \not\#i_1^o \land \text{REPL}(A, v[i_1..i_2] : = E_v) \)

(b) \( \not\#(i_2 + 1)^o \land A(v[i_1..i_2]) \equiv \not\#(i_2 + 1)^o \land \text{REPL}(A, v[i_1..i_2] : = E_v) \)

(b) \( \text{lb}(v) > i_1^o \land A(v[i_1..i_2]) \equiv \text{lb}(v) > i_1^o \land \text{REPL}(A, v[i_1..i_2] : = E_v) \)

(b) \( i_1^o > (i_2 + 1)^o \land A(v[i_1..i_2]) \equiv i_1^o > (i_2 + 1)^o \land \text{REPL}(A, v[i_1..i_2] : = E_v) \)

(b) \( (i_2 + 1)^o > \text{ub}(v) + 1 \land A(v[i_1..i_2]) \equiv (i_2 + 1)^o > \text{ub}(v) + 1 \land \text{REPL}(A, v[i_1..i_2] : = E_v) \)
where \( A(v_{i_1,...,i_2}) \) is any wff containing an occurrence of \( v_{i_1,...,i_2} \).

Conjunct 5 remains invariant, and thus holds at termination, because the algorithm only replaces occurrences of the form \( v \text{var}[i_1,...,i_2] \) with occurrences of \( E_v \).

The invariance of Conjunct 8 follows from this argument: The only side-effect that either production has on the wffs in \( \Omega \) is to substitute \( E_v \) for some terms of the form \( v[i_1,...,i_2] \). Therefore, for each \( v \) and \( w \), the set \( IX(w,v) \) can only be decreased in size by the action of the production; it cannot be increased.

For either production the occurrence of \( v[i_1,...,i_2] \) which is changed to \( E_v \) must be in the U-part, the EQ-part or the ORD-part of \( w \), since \( 0^0 \) does not occur in the Y-part as we have demonstrated before. If the elimination of the term \( v[i_1,...,i_2] \) from \( w \) does not cause \( IX(w,v) \) to shrink (because \( lb(v) \), \( i_1^0 \), \( i_2^0 + 1 \) and \( ub(v) + 1 \) are in \( IX(w,v) \) by virtue of other occurrences of \( v \) in \( w \) besides the one being eliminated) then clearly the form of the wff is still \( Y A U A E Q A O R D \), leaving Conjunct 8 invariant. On the other hand, if the replacement does shrink the set \( IX(w,v) \), then all of the conjuncts of the Y-part before the production executes can be rearranged and considered conjuncts of the U-part after the production executes. The result will again fit the pattern \( Y A U A E Q A O R D \) and will satisfy the conditions on \( Y, U, EQ \) and ORD given in Subsection 6.5.4. This completes the proof that Conjunct 8 of the Invariant is invariant.

To show that the Postcondition holds at termination we note that the first 8 conjuncts are part of the Invariant, so we need only consider Conjuncts 9 and 10.

Conjunct 9 follows from the fact that the pattern of production (a) is not satisfiable at termination. For all members of \( IX(w,v) \) of the form \( lb(v) \) or \( ub(v)+1 \) it is clear (by the definition of ICS) that \( Y \models lb(v) \) and \( Y \models (ub(v)+1) \). Thus, we need only be concerned with the other terms in \( IX(w,v) \). But the unsatisfiability of the pattern-part of production indicates that there is no \( i_1^0 \) or \( (i_2+1)^0 \) in \( IX(w,v) \) such that \( Y \models i_1^0 \) or \( Y \models (i_2+1)^0 \). Hence, for all members \( i^0 \) of \( IX(w,v) \) it must be the case that \( Y \models i^0 \). This proves that Conjunct 9 holds at termination.
Using Conjunct 9 and the unsatisfiability of the pattern for production (b), a similar argument proves that Conjunct 10 holds at termination as well.

6.6.4 Termination

Since $\Omega$ is finite, the outer loop is executed only a finite number of times. For each $w \in \Omega$, the production system in the loop terminates because the number of occurrences of $o[0..0]$ is reduced with each iteration.
6.7 Step 7: Remove all occurrences of $E_v$

We now remove all occurrences of the constant $E_v$. The wff is now structured in such a way that after this step no $V$-term in any wff $w \in \Omega$ will take the value $E_v$ under any assignment $\varphi$ satisfying $w$.

6.7.1 Specifications

**Precondition:** Same as Postcondition of Step 6.

**Invariant:** Same as Precondition.

**Postcondition:**

1. $\Omega \subseteq \text{BITZV}$ with elem; $\Omega$ quantifier-free and finite.
2. $Ew_0 \not\subseteq \not\Omega$.
3. The only $V$-relevant functions occurring in $\Omega$ are $=$, ord, lb, ub, elem, $\{^o\}$, and $^o[^o..]^o$. [Note: $E_v$ has been removed.]
4. All $V$-args to lb, ub and $^o[^o..]^o$ are simple $V$-vars.
5. All $V$-args to $=$, ord, and $\{^o\}$ are $vvar[i_1..i_2]$. [Note: $E_v$ has been removed.]
6. All arguments to elem are of the form $vvar[i..i]$.
7. No nested occurrences of $^o[^o..]^o$.
8. All $w \in \Omega$ are of the form
   \begin{align*}
   Y & \land U \land EQ \land ORD \\
   \text{where } Y, U, EQ \text{ and ORD are as before.}
   \end{align*}
9. For all $w \in \Omega$, $v \equiv w$, $i \in I(w,v)$: $Y_w \vdash i$.
10. For all $w \in \Omega$, $v \equiv w$, $v[i..j] \equiv w$: $Y_w \vdash (\text{lb}(v) \leq i^o \land i \leq (j+1)^o \land (j+1)^o \leq \text{ub}(v)+1)$.

6.7.2 The Algorithm

\[\text{do}
\begin{align*}
(a) & \quad E_v = E_v :\rightarrow \text{true}.
\end{align*}\]
(b) \( E_v = \varphi[i_1 \ldots i_2] \rightarrow \text{false} \)
(c) \( \varphi[i_1 \ldots i_2] = E_v \rightarrow \text{false} \)
(d) \( \text{ord}(E_v) \rightarrow \text{error} \)
(e) \( \text{elem}(E_v) \rightarrow E_T \)
(f) \( \{E_v\} \rightarrow E_Z \)

6.7.3 Proof of Weak Correctness

The invariance of Conjuncts 1 and 3-7 of the Invariant is trivial to verify.

Conjunct 8 is invariant over productions (a), (d)-(f) trivially, because they do not disturb any terms of the form \( \varphi[i_1 \ldots i_2] \), and thus do not affect IX(w,v). It is also invariant over productions (b) and (c) because, as argued in the previous step, these productions can only shrink IX(w,v) (for the w and v in question), and cannot enlarge it. Hence, to preserve the form of w as Y\text{U}EQ\text{ORD} we need only move some of the conjuncts of the Y-part (namely, those referring to \( i_1^0 \) or \( (i_2+1)^0 \) or both, depending on which terms are removed from IX(w,v)) from the Y-part to the U-part. This preserves the form required by Conjunct 8 and completes the proof of its invariance.

Conjuncts 9 and 10 of the Invariant, like Conjunct 8, are invariant over productions (a) and (d)-(f) because those productions have no effect on IX(w,v) since they do not involve terms containing \( 0[0 \ldots 0] \). They are invariant over productions (b) and (c) because, as described in the previous paragraph, those productions can only shrink IX(w,v). The Y-part of w after the execution of production (b) or (c) still contains the same complete information about the definedness and order of the terms remaining in IX(w,v) that it had before the execution of the productions.

We have shown the invariance of all of the Invariant except Conjunct 2. Its invariance over productions (a) and (d)-(f) is trivial because those productions represent identity replacements. Proof of its invariance across productions (b) and (c) requires the use of Conjuncts 9 and 10. We will prove invariance of (b), since the proof for (c) will be identical.
Any \( w \in \Omega \) containing a term of the form \( E_v^{=v[i_1..i_2]} \) contains it as a top-level conjunct (by Conjunct 8 of the Invariant). But by Conjuncts 9 and 10 the Y-part of the same wff \( w \) implies that \( \neg E_v^{=v[i_1..i_2]} \). Hence, at this stage any wff in \( \Omega \) containing the subformula \( E_v^{=v[i_1..i_2]} \) is unsatisfiable. Therefore, substituting \textit{false} for a top-level conjunct, as production (b) does, preserves that unsatisfiability and leaves invariant Conjunct 2. This completes the proof of invariance.

The proof that the Postcondition holds at termination is simple. The only difference between it and the Invariant is the requirement in Conjunct 3 (and 5) that there be no occurrences of \( E_v \) in any \( w \in \Omega \). But by Conjuncts 3-6 of the Invariant we know that any occurrences of \( E_v \) must be as arguments to \( e_v \), ord, elem or \( \{o\}_v \). Since each of those possibilities is covered by one or more patterns in the algorithm, the algorithm cannot terminate with any occurrences of \( E_v \) in any \( w \in \Omega \).

6.7.4 Termination

The termination argument is trivial; each iteration reduces the number of occurrences in \( \Omega \) of the constant \( E_v \).
6.8 Step 8: Remove zero-length vectors; canonicalize expressions based on \( \equiv_v \)

In the previous sections we were able to use our endpoint information to remove \( E_v \) from all wffs in \( \Omega \). Not only was the symbol \( E_v \) removed, but no \( V \)-term \( v \) can now take the value \( E_v \) under any assignment satisfying \( w \). In this section we will perform the same kind of removal for empty vectors (zero-length vectors). After this step we will be able to assume that any \( V \)-term has a positive length, and this will simplify slightly the removal of vector overlapping in Step 9.

6.8.1 Specifications

**Precondition:** Same as Postcondition of previous step.

**Invariant:** Same as Precondition

**Postcondition:**

1. \( \Omega \subseteq \text{BITZV} \) with \( \text{elem}; \) \( \Omega \) is quantifier-free and finite.
2. \( \forall w_0 \iff \not\exists \Omega. \)
3. The only \( V \)-relevant functions occurring in \( \Omega \) are \( \equiv, \text{ord}, \text{lb}, \text{ub}, \text{elem}, \{^0\} \) and \( 0[^0..^0] \).
4. All \( V \)-args to \( \text{lb}, \text{ub} \) and \( 0[^0..^0] \) are simple \( V \)-vars.
5. All \( V \)-args to \( \equiv, \text{ord}, \text{elem} \) and \( \{^0\} \) are of the form \( \text{vvar}[i_1..i_2] \).
6. All arguments to \( \text{elem} \) are of the form \( \text{vvar}[i..i] \).
7. The function \( 0[^0..^0] \) does not occur nested.
8. All \( w \in \Omega \) are of the form
   \[ Y \land U \land \text{EQ} \land \text{ORD} \]
   where \( Y, U, \text{EQ} \) and \( \text{ORD} \) are as before.
9. For all \( w \in \Omega, v \equiv w, i \in \text{IX}(w,v) \): \( Y_w \vdash \equiv^i \).
10. For all \( w \in \Omega, v[i..j] \equiv w \): \( Y_w \vdash (\text{lb}(v) \equiv^i \land i^0 < (j+1)^0 \land (j+1)^0 \equiv \text{ub}(v) + 1) \)
11. For all \( w \in \Omega, v_1[i_1..i_2] \equiv v_2[i_3..i_4] \equiv w \): \( i_1 \) is the same term as \( i_3 \),
    and \( i_2 \) is the same term as \( i_4 \).
6.8.2 The Algorithm

For all \( w \in \Omega \) perform the following transformation

\[
\text{do }
\begin{align*}
(a) & \quad \text{ord}(v[1, \ldots, 2]) = w \wedge Y \vDash (i_2+1)^o i_1^o \Rightarrow \\
& \quad w \leftarrow \text{REPL}(w, \text{ord}(v[1, \ldots, 2]) \Rightarrow \text{true}) \tag{d} \\
(b) & \quad v_1[1, \ldots, 2] = v_2[i_3, \ldots, i_4, i_2] \wedge Y \vDash (i_2+1)^o i_1^o \vee (i_4+1)^o i_3^o \Rightarrow \\
& \quad w \leftarrow \text{REPL}(w, v_1[1, \ldots, 2] = v_2[i_3, \ldots, i_4] \Rightarrow i_3 = i_1 \wedge i_4 = i_2) \tag{b} \\
(c) & \quad \{v[1, \ldots, 2]\} = w \wedge Y \vDash (i_2+1)^o i_1^o \Rightarrow \\
& \quad w \leftarrow \text{REPL}(w, \{v[1, \ldots, 2]\} \Rightarrow \phi_2) \tag{c} \\
(d) & \quad v_1[1, \ldots, 2] = v_2[i_3, \ldots, i_4, i_2] = w \Rightarrow \\
& \quad w \leftarrow i_3 = i_1 \wedge i_4 = i_2 \wedge \text{REPL}(w, v_2[i_3, \ldots, i_4] \Rightarrow v_2[i_1, \ldots, i_2]) \tag{d}
\end{align*}
\]

6.8.3 Proof of Weak Correctness

The Invariant obviously holds at the beginning of this step, and Conjuuncts 1, 3, 4, 5, 6 and 7 are trivially invariant over all four of the productions. Conjuuncts 8, 9 and 10 are also obviously invariant by arguments we have given before once we note that each of the productions only removes vector terms, thereby only shrinking the size of \( I_X(w, v) \) for each \( v \), or at least never increasing it.

As usual, the substance of the invariance proof involves Conjuunct 2. We will give a brief informal sketch of the proof.

The invariance of Conjuunct 2 over production (a) rests on the fact that empty vectors are always (by definition) ordered. The production is only applicable when

\[
\text{ord}(v[i_1, i_2]) = w
\]
and thus, by Conjunctions 9 and 10 of the Invariant,

\[ Y_w \equiv i_1^0 \land #(i_2+1)^0 \land \text{lb}(w) \leq i_1^0 \land i_1^0 = (i_2+1)^0 \land (i_2+1)^0 \text{sub}(w)+1 \]

i.e. when \( Y_w \) requires that \( v[i_1..i_2] \) be both defined and empty. We can view production (a), then, as an identity replacement based on the following identity:

\[ F \equiv i_1 \land #(i_2+1) \land \text{lb}(w) \leq i_1 \land i_1 = i_2 + 1 \land i_2 + 1 \leq \text{sub}(w)+1 \land \text{ord}(v[i_1..i_2]) = i_1 \land #(i_2+1) \land \text{lb}(w) \leq i_1 \land i_1 = i_2 + 1 \land i_2 + 1 \leq \text{sub}(w)+1 \land \text{true}. \]

Alternatively, it can be viewed as being justified by the following "conditional identity":

\[ F \equiv i_1 \land #(i_2+1) \land \text{lb}(w) \leq i_1 \land i_1 = i_2 + 1 \land i_2 + 1 \leq \text{sub}(w)+1 \Rightarrow (\text{ord}(v[i_1..i_2])=\text{true}). \]

In any case, Conjunction 2 is invariant over production (a) because the new value of \( w \) is equivalent to (in the sense of \( \equiv_B \)) the old.

The arguments showing the invariance of Conjunction 2 over the productions (b)-(d) are similar, so we will confine ourselves to informal summaries.

Production (b) is justified by the fact that if either one (or both) of a pair of vectors \( v_1[i_1..i_2] \) and \( v_2[i_3..i_4] \) is empty, then they have identical values if and only if \( i_1 = i_3 \) and \( i_2 = i_4 \). We can put this justification into the form of the following conditional identity:

\[ i_1 \land #(i_2+1) \land \text{lb}(v_1) \leq i_1 \land i_1 = i_2 + 1 \land i_2 + 1 \leq \text{sub}(v_1)+1 \land \]

\[ i_3 \land #(i_4+1) \land \text{lb}(v_2) \leq i_3 \land i_3 = i_4 + 1 \land i_4 + 1 \leq \text{sub}(v_2)+1 \land \]

\((i_1 = i_2 + 1 \lor i_3 = i_4 + 1) \Rightarrow (v_1[i_1..i_2] = v_2[i_3..i_4] = i_1 = i_3 \land i_2 = i_4).\]

The invariance of Conjunction 2 over production (c) depends simply on the fact that the zset of elements in a defined, empty vector is the empty zset, \( \phi_z \). We will omit the display of the corresponding conditional identity.

Finally, the invariance of Conjunction 2 over production (d) rests on the observation that two vectors cannot be identical if their bounds are not equal.
This completes the proof of invariance.

To prove that the Postcondition holds at termination we first point out that the difference between the Postcondition and the Invariant is a change in Conjunct 10 and the addition of Conjunct 11. We treat these separately.

Conjunct 10 of the Postcondition has the expression \( i^o < (j+1)^o \), whereas Conjunct 10 of the Invariant had \( i^o \leq (j+1)^o \). The Postcondition says, therefore, that not only are there no V-terms taking the undefined value, but there are no V-terms taking any empty vector as a value either (under an assignment satisfying \( w \)).

Production (a) removes occurrences of terms appearing as arguments to \( \text{ord} \) which take a length zero value.

Production (b) removes occurrences of length zero V-terms from either side of an \( \varepsilon \)-atom.

Production (c) does the same for arguments to \( \{ 0 \} \).

Since these exhaust all positions where terms of the form \( v[i_1..i_2] \) can occur in \( w \), the PS cannot halt with any such terms which can be zero length under any assignment satisfying \( w \), and the 10th conjunct is established.

Conjunct 11 is trivially established by production (d).

6.8.4 Termination

Each production reduces the number of occurrences of \( o[0..0] \) in \( w \), or in the case of production (d), leaves the number of occurrences of \( o[0..0] \) constant but reduces the number of atoms of the form \( v_1[i_1..i_2] = v_2[i_3..i_4] \) in which \( i_1 \) is different from \( i_3 \) or \( i_2 \) is different from \( i_4 \).
6.9 Step 9: Remove vector overlap

This step, in conjunction with Step 5, is the key in the Type Reduction of BITZV to BITZ. What we are striving for is a situation in which any two distinct vector terms \( v[i_{1-2}] \) and \( v[i_{3-4}] \) occurring in a wff \( w \) can be considered to be independent, as though they were separate variables.

In Chapter 5 we showed how, in a language simpler than BITZV, the terms \( v[i] \) and \( v[j] \) can be considered independent in a wff \( w \) when it is known that \( i \) cannot equal \( j \), that is, when \( w \) cannot be satisfied by any assignment \( \varphi \) such that \( v_{\varphi(i)} = v_{\varphi(j)} \). We also know from Chapter 5 and from common experience with Algol-like programming languages that when \( i \neq j \) the terms \( v[i] \) and \( v[j] \) can be treated as (distinct) variables of type \( T \), resulting in considerable simplification.

We will now generalize these notions. We observe that for the terms \( v[i_{1-2}] \) and \( v[i_{3-4}] \) to be viewed as independent it is necessary and sufficient that the two segments \([i_{1-2}] \) and \([i_{3-4}] \) not overlap; more specifically, there must not be any integer \( i \) such that \( i_{1} \leq i \leq i_{2} \) and \( i_{3} \leq i \leq i_{4} \). (We can ignore error values here because we have already eliminated them. See Conjunct 10 of the Precondition for this step).

The precise condition we will achieve in this step is this: for any \( V \)-terms occurring in a wff \( w \) and based on the same \( V \)-variable \( v \), say the terms \( v[i_{1-2}] \) and \( v[i_{3-4}] \), either \( y_{w} \neq i_{3} \land i_{2} \neq i_{4} \), in which case the two terms refer to exactly the same segment of the vector, or \( y_{w} \neq i_{3} \lor i_{2} < i_{3} \), in which case the two terms refer to nonoverlapping parts of vector \( v \). The effect will be that the two \( V \)-terms will either refer to identical parts of a vector and act like two occurrences of a single \( V \)-variable, or they will refer to disjoint, independent parts of the vector, thereby acting like occurrences of two distinct \( V \)-variables.

The way we achieve this non-overlapping condition will be further explained in the Proof of Weak Correctness below.
6.9.1 Specifications

Precondition: Same as Postcondition of Step 8

Invariant: Same as Precondition

Postcondition:

1. $\Omega \subseteq \text{BITZV}$ with elem; $\Omega$ is quantifier-free and finite.
2. $\# w_0 \iff \not \Omega$.
3. The only V-relevant functions occurring in $\Omega$ are $\ast$, ord, lb, ub, elem, $\{^o\}$ and $^o[\ldots]^o$.
4. All $V$-args to lb, ub and $^o[\ldots]^o$ are simple $V$-vars.
5. All $V$-args to $\ast$, ord, and $\{^o\}$ are of the form $v\var[i_1\ldots i_2]$.
6. All arguments to elem are of the form $v\var[i..i]$.
7. There are no nested occurrences of $^o[\ldots]^o$.
8. All $w \in \Omega$ are of the form
   $Y \land U \land EQ \land ORD$
   where $Y$, $U$, $EQ$ and $ORD$ are as before.
9. For all $v \approx w$, $i \in \text{IX}(w,v)$: $Y_w \vdash v^o$.
10. For all $w \in \Omega$, $v \approx w$, $v[i..j] \approx w$: $Y_w \vdash (lb(v) \leq i^o \land i^o < (j+1)^o \land (j+1)^o \leq ub(v)+1)$.
11. For all $w \in \Omega$, $v_1[i_1..i_2] \approx v_2[i_3..i_4] \approx w$: $i_1$ is the same term as $i_3$ and $i_2$ is the same term as $i_4$.
12. For all $w \in \Omega$, $v[i_1..i_2] \approx w$, $v[i_3..i_4] \approx w$:
   $Y_w \vdash (i_1^o = i_3^o \land (i_2+1)^o = (i_4+1)^o) \lor$
   $Y_w \vdash (i_4^o + 1)^o \leq i_1^o \lor (i_2^o + 1)^o \leq i_3^o)$.

6.9.2 Notation

In the following program let

LEO $\triangleq Y_w \vdash i_3^o < i_2^o \land i_3^o < (i_2 + 1)^o$

and
HEO $\leq Y_{(4+1)} \wedge (i_4+1)^9$.

LEO ("low-end overlap") and HEO ("high-end overlap") are "macros" for patterns useful in abbreviating the production system below.

6.9.3 The Algorithm

For each $w \in \Omega$ perform the following transformation:

do (a) $\text{ord}(v[i_1..i_2])^w \wedge \text{ord}(v[i_3..i_4])^w \wedge \text{LEO} \Rightarrow$

\[ w \leftarrow \text{REPL}(w, \text{ord}(v[i_1..i_2])) \quad \text{ord}(v[i_1..(i_4-1)]) \wedge \max\{(v[i_1..(i_4-1)]) \leq \min\{(v[i_3..i_2]) \wedge \text{ord}(v[i_3..i_2])
\]

(b) $\text{ord}(v[i_1..i_2])^w \wedge \text{ord}(v[i_3..i_4])^w \wedge \text{HEO} \Rightarrow$

\[ w \leftarrow \text{REPL}(w, \text{ord}(v[i_1..i_2])) \quad \text{ord}(v[i_1..i_4]) \wedge \max\{(v[i_1..i_4]) \leq \min\{(v[i_4+1..i_2]) \wedge \text{ord}(v[i_4+1..i_2])
\]

(c) $v_1[i_1..i_2] = v_2[i_1..i_2]^w \wedge v_1[i_3..i_4]^w \wedge \text{LEO} \Rightarrow$

\[ w \leftarrow \text{REPL}(w, v_1[i_1..i_2] \equiv v_2[i_1..i_2]) \rightarrow \text{v}_1[i_1..i_3-1] = v_2[i_1..i_3-1] \wedge \text{v}_1[i_3..i_2] = v_2[i_3..i_2]
\]

(d) $v_1[i_1..i_2] = v_2[i_1..i_2]^w \wedge v_1[i_3..i_4]^w \wedge \text{HEO} \Rightarrow$

\[ w \leftarrow \text{REPL}(w, v_1[i_1..i_2] \equiv v_2[i_1..i_2]) \rightarrow \text{v}_1[i_1..i_4] = v_2[i_1..i_4] \wedge \text{v}_1[i_4+1..i_2] = v_2[i_4+1..i_2]
\]

(e) $v_2[i_1..i_2] = v_1[i_1..i_2]^w \wedge v_1[i_3..i_4]^w \wedge \text{LEO}$

\[ \Rightarrow \text{same as (c) above}
\]

(f) $v_2[i_1..i_2] = v_1[i_1..i_2]^w \wedge v_1[i_3..i_4]^w \wedge \text{HEO}$
6.9.4 Proof of Weak Correctness

This proof of weak correctness and the proof of termination in the next subsection are probably the key proofs of Chapter 6.

The main idea behind the algorithm in this step is this: whenever two nonempty vector terms in \( w \) refer to overlapping but unequal segments of the integers, for example the terms \( v[i_1..i_2] \) and \( v[i_3..i_4] \) where \( i_1 < i_3 < i_2 < i_4 \), then we must find some way to re-express \( w \) in such a way that the meaning of \( w \) is preserved, but the vector terms which occur in the altered formula do not overlap. The algorithm in this step accomplishes this goal by re-expressing the formula \( w \) using only the vector terms

\[
v[i_1..i_3-1], v[i_3..i_2] \text{ and } v[i_2+1..i_4].
\]

In our example where \( i_1 < i_3 < i_2 < i_4 \) these three terms span the same part of the vector’s domain as the terms \( v[i_1..i_2] \) and \( v[i_3..i_4] \), but they are disjoint, and thus their values are independent.

The purpose of these proofs is to show that such term-splitting is always possible regardless of the way that \( v[i_1..i_2] \) and \( v[i_3..i_4] \) overlap, and without leaving the language BITZV (with elem). And the purpose is further to show that the process can be made to converge.

The actual condition to be achieved in this step is given in Conjunct 12 of the Postcondition, which states that if two vector terms occur in \( w \in \Omega \), then either they are equal, or their domains are disjoint. However, the main part of the proof is devoted to showing that
the other 11 conjuncts, which form the Invariant, are in fact invariant. Then we will get to the Postcondition.

6.9.4.1 Invariant

Simple examination shows that Conjuncts 1, 3, 4, 5, 6 and 7 are obviously invariant across all of the productions.

Conjunct 11 is invariant by a simple argument: Expressions based on $v$ are unaffected by productions (a)-(b) and (g)-(h), and a cursory examination reveals that they are invariant over productions (c)-(f) as well.

The invariance of the rest of the conjuncts, namely (2), (8), (9) and (10) hinges on the following tricky observation. Even though each production removes some V-terms from $w$, and introduces others that did not occur before, the set $\mathbb{I}(w,v)$ is not changed. From the invariance of the value of $\mathbb{I}(w,v)$ the other invariances follow, so we prove it first.

Consider first production (a). Whenever it is executed we can see before its execution the contribution made to the index set $\mathbb{I}(w,v)$ by the terms $v[i_1,i_2]$ and $v[i_3,i_4]$ is

$$\{i_1^0, (i_2+1)^0, i_3^0, (i_4+1)^0\}.$$  

After the execution of production (a) the term $v[i_1,i_2]$ has been removed from $w$ (at least one occurrence has) and it has been replaced by occurrences of the terms $v[i_1,i_3-1]$ and $v[i_3,i_2]$. The contribution of these terms, and the unaffected term $v[i_3,i_4]$ to $\mathbb{I}(w,v)$ is

$$\{i_1^0, ((i_3-1)+1)^0, i_3^0, (i_4+1)^0, i_4^0\}.$$  

The key observation is that $(i_3-1+1)^0$ and $i_3^0$ are the same term, according to Lemma 3 of Section 6.5. Hence, the contribution of the terms in question to $\mathbb{I}(w,v)$ before execution of production (a) is the same as that after it. By extension, the set $\mathbb{I}(w,v)$ remains unchanged over production (a).

A similar analysis for productions (b)-(h), using the same Lemma, will easily establish that $\mathbb{I}(w,v)$ remains unchanged over all of the other productions as well. (In fact, the entire
purpose of introducing the simplification operation \( ^{\circ} \) was to simplify this part of the proof.

Now that we have proved that \( IX(w,v) \) is unchanged by any of the productions, we can proceed with the unfinished business of demonstrating the invariance of Conjuncts 2, 8, 9 and 10 of the Invariant.

None of the productions affects the \( Y \)-part of the formulae. Consequently, since \( IX(w,v) \) remains invariant, the unchanged \( Y \)-part still retains complete information about \( IX(w,v) \). Also note that since ord and \( \equiv \)-atoms are replaced by conjunctions of the same kinds of atoms in productions (a)-(f), the structure of EQ and ORD are preserved. Hence, Conjunct 8 is invariant.

The invariance of Conjuncts 9 and 10 follows directly from the fact that \( IX(w,v) \) is unchanged by any of the productions, and the fact that the \( Y \)-part of the formulae is also unchanged. The invariance of Conjunct 10 requires examining each production, and showing that for each \( V \)-term \( v[i..j] \) introduced by that production the condition \( Y_w \vdash \text{lb}(v) \leq i^{\circ} \land i^{\circ} < (j+1)^{\circ} \land (j+1)^{\circ} \leq \text{ub}(v)+1 \) holds after the production's execution (using of course, the fact that the condition held before its execution.) The analysis for production (a) proceeds like this.

Production (a) introduces only two \( V \)-terms which were not (necessarily) present in \( w \) before its execution. These terms are

\[
\begin{align*}
v[i_1..i_3-1] \\
v[i_3..i_2]
\end{align*}
\]

To show the invariance of Conjunct 10 we observe that production (a) does not alter the \( Y \)-part of a wff \( w \), and we therefore need only show that

\[
\begin{align*}
(i) & \quad Y_w \vdash \text{lb}(v) \leq i_1^{\circ} \\
(ii) & \quad Y_w \vdash (i_3-1+1)^{\circ} \\
(iii) & \quad Y_w \vdash (i_3-1+1)^{\circ} \leq \text{ub}(v)+1 \\
(iv) & \quad Y_w \vdash \text{lb}(v) \leq i_3^{\circ} \\
(v) & \quad Y_w \vdash (i_2+1)^{\circ} \\
(vi) & \quad Y_w \vdash (i_2+1)^{\circ} \leq \text{ub}(v)+1
\end{align*}
\]

Propositions (i), (iv) and (vi) follow from Conjunct 10 of the Invariant before production (a) was executed, since \( v[i_1..i_2] \) and \( v[i_3..i_4] \) were terms occurring in \( w \) at that time. Proposition
(v) is required by the LEO specification in the pattern part of production (a) — in other words, production (a) cannot be executed if proposition (v) does not hold — and it is not disturbed by the execution of the production's right hand side. We are left, then, with demonstrating propositions (ii) and (iii).

Both propositions (ii) and (iii) use the fact (established in Subsection 6.5.2, that \(((i_3-1)+1)^o\) is the same term as \(i_3^o\). We can thus rewrite these propositions as

\[
\begin{align*}
(ii) \quad & Y_{w|j_1}^o < i_3^o \\
(iii) \quad & Y_{w|j_3}^o \leq ub(v)+1
\end{align*}
\]

Rewritten this way, proposition (ii) is justified the same way as (v), and relation (iii) is justified the same way as (vi).

This proof of invariance of Conjunct 10 over production (a) is now complete. The proofs for the other seven productions follow the same pattern and are omitted here. We also omit the proof of invariance for Conjunct 9 since it is similar to, and shorter than, the proof for Conjunct 10.

To complete our proof that the Invariant assertion is indeed invariant over the productions of this algorithm we need only dispose of Conjunct 2, which we have saved until last. Fortunately, the proof of its invariance is now fairly straightforward.

We will take advantage of the symmetry between LEO and HEO. LEO means that \(v[i_{1..i_2}]\) overlaps and extends beyond the low end of \(v[i_{3..i_4}]\), so that \(i_1<i_3<i_2+1\). HEO specifies that \(v[i_{1..i_2}]\) overlaps and extends beyond the high end of \(v[i_{3..i_4}]\), so that \(i_1<i_4+1<i_2+1\). Because of Conjunct 10 of the Invariant we know that \(Y_{i_3}^j<i_4+1\), and thus \(v[i_{3..i_4}]\) is not an empty vector and hence the overlap is not empty under any assignment \(\phi\) satisfying \(w\).

Because of the LEO/HEO symmetry we will confine our attention to productions (a), (c), (e) and (g). We will also take advantage of the symmetry between productions (c) and (e), which do the same thing on the left and right sides of \(\approx_v\)-atoms.

We are thus reduced by symmetry to considering only productions (a), (c) and (g). These
productions preserve the truth of Conjunct 2 because they represent replacements justified by the following "conditional identity schemas", holding for any V-variable v and any I-terms i_1, ..., i_4.

(a) \( \exists i_1 \land \#(i_2+1) \land \text{lb}(v) \leq i_1 \land i_2+1 \leq \text{sub}(v)+1 \land i_1 < i_3 \land i_3 < i_2+1 \quad \Rightarrow \quad \text{ord}(v[i_1..i_2]) = B \left( \text{ord}(v[i_1..i_3-1]) \land \max \{\{v[i_1..i_3-1]\}\} \leq \min \{\{v[i_3..i_2]\}\} \right) \land \text{ord}(v[i_3..i_2]) \)

(c) \( \exists i_1 \land \#(i_2+1) \land \text{lb}(v_1) \leq i_1 \land i_2+1 \leq \text{sub}(v_1)+1 \land \text{lb}(v_2) \leq i_1 \land i_2+1 \leq \text{sub}(v_2)+1 \land i_1 < i_3 \land i_3 < i_2+1 \quad \Rightarrow \quad \begin{align*} \langle v_1[i_1..i_2] = v_2[i_1..i_2] \rangle^B & \quad \langle v_1[i_1..i_3-1] = v_2[i_1..i_3-1] \land v_1[i_3..i_2] = v_2[i_3..i_2] \rangle \end{align*} \)

(g) \( \exists i_1 \land \#(i_2+1) \land \text{lb}(v) \leq i_1 \land i_2+1 \leq \text{sub}(v)+1 \land i_1 < i_3 \land i_3 < i_2+1 \quad \Rightarrow \quad \{v[i_1..i_2]\} \neq \emptyset \{v[i_1..i_3-1]\} \cup \{v[i_3..i_2]\} \)

Since for each \( w \in \Omega \) the hypotheses of these equivalences are true by Conjuncts 8, 9, 10 and 11, the replacements performed by productions (a), (c) and (g) leave the resulting wff identical (in the sense of \( \#_B \)) to the original.

This completes the proof that Conjunct 2 is invariant over the productions (a)-(h), and also completes the proof that the entire Invariant assertion is invariant.

6.9.4.2 Postcondition

Since Conjuncts 1-11 of the Postcondition are identical to those of the Invariant, we need only prove that Conjunct 12 holds when and if the PS terminates to complete the proof of weak correctness.

Conjunct 12 says that any two occurrences of V-terms \( v[i_1..i_2] \) and \( v[i_3..i_4] \) that are based on the same V-variable (or even the same occurrence merely considered twice) must either represent exactly the same vector, i.e. \( Y_{w_1} \chi_{i_1}^0 = i_3^0 \land (i_2+1)^0 = (i_4+1)^0 \), or they must represent disjoint or nonoverlapping parts of \( v \), i.e. \( Y_{w_1} (i_2+1)^0 \leq i_3^0 \lor (i_4+1)^0 \leq i_4^0 \).
At this point in the algorithm, $V$-terms of the form $v[i..j]$ can occur in only five contexts, namely, as arguments to $ord$, either side of $v$, $\{v\}$, or to $elem$. The first four cases are explicitly mentioned in productions, but the last is not. This is not an oversight; we will see that it is unnecessary to do anything to terms of the form $elem(v[i..j])$ in order to establish Conjunct 12. Let us call occurrences of $V$-terms as arguments to $ord$, $v$ or $\{v\}$ "A-occurrences", and we will call occurrences of $V$-terms as arguments to $elem$ "B-occurrences".

To prove that Conjunct 12 holds for all pairs of occurrences of vector terms having the appropriate forms, let us choose an arbitrary pair, which we denote by $v[i_1..j_2]$ and $v[i_3..j_4]$. There are four cases to consider:

1. $v[i_1..j_2]$ and $v[i_3..j_4]$ are both A-occurrences
2. $v[i_1..j_2]$ is an A-occurrence and $v[i_3..j_4]$ is a B-occurrence
3. $v[i_1..j_2]$ is a B-occurrence and $v[i_3..j_4]$ is an A-occurrence
4. $v[i_1..j_2]$, $v[i_3..j_4]$ are both B-occurrences

We will show that in all four cases, when the PS terminates one of the three conditions required by Conjunct 12, namely $Y_w^H(i_1^o=i_3^o \land (j_2+1)^o=(j_4+1)^o)$ or $Y_w^H((j_2+1)^o \leq i_3^o \land (j_4+1)^o \leq i_3^o)$, holds (which you will recall by the definition of $\mathcal{H}$ and $\mathcal{S}$, is the same as

$$Y_w^H((i_1^o=i_3^o \land (j_2+1)^o=(j_4+1)^o) \lor (j_2+1)^o \leq i_3^o \lor (j_4+1)^o \leq i_3^o).$$

Case 1

Whenever the PS halts, all of the patterns are unsatisfiable. Since each of the types of A-occurrences is covered by a pair of productions, one having $Y_w^H$ in the pattern and the other having $Y_w^{HEO}$ in the pattern, we know that $Y_w^{HEO}$ and $Y_w^H$ cannot be satisfied when $j_1$, $j_2$, $j_3$, and $j_4$ are substituted for $i_1$, $i_2$, $i_3$, and $i_4$ respectively in the definition of $HEO$ and $LEO$, and also that they cannot be satisfied for $i_1$, $i_2$, $i_3$, and $i_4$ respectively. (In other words, with $v[i_3..j_4]$ and $v[i_1..j_2]$ reversed). Expanded, this means that
\[ Y_{w} j_{1}^{0} < (j_{2}+1)^{0}, \] 
\[ Y_{w} j_{2}^{0} < (j_{4}+1)^{0} \land (j_{4}+1)^{0} < (j_{2}+1)^{0}, \]
\[ Y_{w} j_{3}^{0} < j_{1}^{0} \land j_{2}^{0} < (j_{4}+1)^{0}, \] 
and
\[ Y_{w} j_{3}^{0} < (j_{2}+1)^{0} \land (j_{2}+1)^{0} < (j_{4}+1)^{0}. \]

We can combine this with our knowledge from Conjunct 10 of the Invariant that
\[ Y_{w} j_{1}^{0} < (j_{2}+1)^{0}, \]
and
\[ Y_{w} j_{3}^{0} < (j_{4}+1)^{0}. \]

A lengthy and tedious manipulation will verify that the above six conditions imply
\[ Y_{w} (j_{1}^{0} = j_{3}^{0} \land (j_{2}+1)^{0} = (j_{4}+1)^{0}) \lor (j_{2}+1)^{0} \leq j_{3}^{0} \lor (j_{4}+1)^{0} \leq j_{1}^{0}, \]
which is what is required for Conjunct 12 of the Postcondition.

**Case 2**

When \( v[j_{1}..j_{2}] \) is an A-occurrence we know, as in Case 1, that because all patterns are unsatisfiable
\[ Y_{w} j_{1}^{0} < j_{2}^{0}, \]
and
\[ Y_{w} j_{3}^{0} < (j_{4}+1)^{0}. \]

Furthermore, from Conjunct 10 of the Invariant we have
\[ Y_{w} j_{1}^{0} < (j_{2}+1)^{0}. \]

Now \( v[j_{3}..j_{4}] \) is a B-occurrence, so it isn’t covered by any of the HEO/LEO conjuncts in any of the productions. But we do know from Conjunct 6 of the Invariant that \( j_{3} \) and \( j_{4} \) are identical terms for B-occurrences. This condition together with the three conditions above, are sufficient to prove that
\[ Y_{w} (j_{2}+1)^{0} \leq j_{3}^{0} \lor (j_{1}^{0} = j_{3}^{0} \land (j_{2}+1)^{0} = (j_{4}+1)^{0}) \lor (j_{4}+1)^{0} \leq j_{1}^{0} \]
as required.

**Case 3**

This case is symmetrical to Case 2; just exchange \( i_{1} \) with \( i_{3} \) and \( i_{2} \) with \( i_{4} \) in that proof.

**Case 4**

In the case that both \( v[j_{1}..j_{2}] \) and \( v[j_{3}..j_{4}] \) are B-occurrences we can apply Conjunct 6 of the Invariant twice, yielding the facts that \( j_{1} \) and \( j_{2} \) are identical terms, and \( j_{3} \) and \( j_{4} \) are identical terms as well. This is obviously sufficient to prove
\[ Y_{w} (j_{2}+1)^{0} \leq j_{3}^{0} \lor (j_{1}^{0} = j_{3}^{0} \land (j_{2}+1)^{0} = (j_{4}+1)^{0}) \lor (j_{4}+1)^{0} \leq j_{1}^{0} \]
6.9.4.2

as required.

This completes the proof that the Postcondition is established.

6.9.5 Termination

The termination argument for this step is quite elegant, making full use of the properties of
multiset well-orderings as described in [Dershowitz 78].

For a wff w, consider the multiset M(w) of occurrences of V-terms (of the form v[i..j]) in
w. Define the following ordering < on the set of elements in M(w): 
"v_1[i_1..i_2]" < "v_2[i_3..i_4]" if and only if these conditions hold:

1. v_1 and v_2 are the same V-variable, and
2. \( Y_w \cdot i_2^o \leq i_1^o \land (i_2+1)^o \leq (i_4+1)^o \land (i_2^o \neq i_3^o \lor (i_2+1)^o \neq (i_4+1)^o) \), meaning that the interval
   \([i_1..i_2]\) is strictly included
3. in the interval \([i_3..i_4]\) according to the information in the Y-part of w.

The number of elements in M(w) is finite, and it is easy to show that < is a well-founded
order on the set of terms underlying M.

Now, consider all multisets of such V-terms ordered by <<, the well-founded order on
multisets induced by the well-founded order < on their elements. (See [Dershowitz 78].)
Each production reduces M(w) according to <<.

For example, consider production (a). It has the following effect on M(w).

\[ M(w) \rightarrow M(w) - \{v[i_1..i_2]\} + \{v[i_1..i_3-1], v[i_1..i_3-1], v[i_3..i_2], v[i_3..i_2]\} \]

But from the pattern part of production (a) we know that it can only be executed when

\[ Y_w \cdot i_3^o < i_2^o \land i_3^o < (i_2+1)^o \]

From this we can deduce that

"v[i_1..i_3-1]" < "v[i_1..i_2]" and

"v[i_3..i_2]" < "v[i_1..i_2]"
Since the term being deleted from $M(w)$ is strictly greater than each of the terms being added (twice each) to $M(w)$, the new value of $M(w)$ is $\ll$ the old value according to the Theorem of Dershowitz and Manna.

Similar analyses apply to all of the other seven productions (b)-(h), completing the proof of termination.
6.9.5 225

6.10 Step 10: Remove occurrences of the restriction function

In this step we perform an enormous simplification: the removal of aliasing. We replace all occurrences of terms having the form \( v[i_1..i_2] \) with occurrences of new \( V \)-variables. Henceforth all \( V \)-terms will be simple \( V \)-variables.

We are able to do this because of what was achieved in the last step. Conjunct 10 of the Precondition says that each \( V \)-term takes a defined value, and Conjunct 13 says that any two \( V \)-terms occurring in \( w \) are either identical (under any assignment satisfying \( w \)) or are disjoint (under any assignment satisfying \( w \)). We will show how to use these facts in the Algorithm and Proof of Correctness subsections.

6.10.1 Specifications

**Precondition:**
Same as Postcondition for Step 9.

**Postcondition:**

1. \( \Omega \subseteq \text{BITZV} \) with elem; \( \Omega \) unquantified, finite.
2. \( \forall w \in \Omega \iff \forall \Omega \).
3. The only \( V \)-relevant functions occurring in \( \Omega \) are \( \varepsilon_V \), ord, lb, ub, elem and \( \{0\}_V \). [Note: \( \{0..0\} \) has been removed!]
4. All \( V \)-terms are simple \( V \)-vars.
5. All \( w \in \Omega \) are of the form
   \[ U \land EQ \land ORD \]
   where
   - \( U \) contains no occurrence of \( \varepsilon_V \) or ord
   - \( EQ \) is a conjunction of atoms of the form \( vvar_1=vvar_2 \)
   - \( ORD \) is a conjunction of atoms of the form \( \text{ord}(vvar) \).
6. For all terms \( \text{elem}(v)=w; \forall w \supset \text{lb}(v) = \text{ub}(v) \).
6.10.2 Notation

Define the following equivalence relation on the V-terms (other than simple variables) occurring in a wff \( w: v_1[i_1..i_2] \equiv_w v_2[i_3..i_4] \) if and only if

1. \( v_1 \) and \( v_2 \) are the same variable

2. \( Y_w w_i = i_3 \wedge (i_2+1)^0 = (i_4+1)^0 \)

It is simple to prove that \( \equiv_w \) is an equivalence relation for any \( w \in \Omega \). As such, it partitions the set of V-terms occurring in \( w \) into equivalence classes. Let \( K(w) \) be the number of such classes, and let us denote the classes by \( E_1^w, ..., E_{K(w)}^w \). Obviously there is a finite number of equivalence classes since the set of V-terms occurring in \( w \) is finite; furthermore each class is recursive (effectively computable) since the relation \( Y_w w \) is recursive, and \( K(w) \) is recursive. Hence we are free to use \( E_1^w, ..., E_{K(w)}^w \) and the function \( K(w) \) in our algorithm without compromising its effectiveness.

6.10.3 The Algorithm

\[
\text{for each } w \in \Omega \text{ perform the following transformation}
\]

\[
\text{for each equivalence class } E \in \{E_1^w, ..., E_{K(w)}^w \} \text{ do}
\]

\[
\begin{align*}
\text{begin} & \\
\text{A: let } v[1..u] \text{ be an arbitrary element of } E; & \\
\text{let } v' \text{ be a new V-variable not occurring in } w; & \\
\text{for all terms } vv \in E \text{ do} & \\
\text{w \leftarrow REPLALL(w, vv; \rightarrow v');} & \\
\text{w \leftarrow (lb(v') = 1 \wedge ub(v') + 1 = u + 1) \wedge w} & \\
\text{B:} & 
\end{align*}
\]

The conjuncts \( lb(v') = 1 \) and \( ub(v') + 1 = u + 1 \) must be included in the \( Y \)-part of the new value of \( w \) for each new V-variable \( v' \).

6.10.4 Proof of Weak Correctness

Most of the precondition of the previous step remains invariant throughout execution of
this step. However, we have not called it "the Invariant" because it is not sufficiently strong to be the loop Invariant for the program as written. We will supplement with informal reasoning to make up the deficiency.

6.10.4.1 Invariance

Conjuncts 1, 3, 4, 6 and 7 are trivially invariant across each step of the algorithm. Conjunct 5 is not invariant, but we don't want it to be.

The invariance of Conjunct 8 requires only a short proof. Obviously the EQ and ORD part of the pattern is not disturbed by the algorithm, and only the Y-part is in question. To understand what happens with it, we need to understand how one iteration of the loop body from A to B in the algorithm affects the set IX(w,v).

Removing terms of the form v[i_1..i_2] can only reduce the size of IX(w,v), and thus the Y-part of the formula will continue to retain complete information about the definedness, equality and order of terms in the "old" IX(w,v) which survive to the "new" IX(w,v). However, the new variable v' occurs in the new wff w, and we must be sure that the Y-part of the formula contains complete information about IX(w,v'). Adding the conjuncts lb(v')=1 and ub(v')=u+1 to the Y-part of the wff w ensures that it does, since IX(w,v') contains only the terms \{lb(v), ub(v)+1\}. So Conjunct 8 is invariant.

The invariance of Conjuncts 9, 10, 11 and 12 follow easily from that of Conjunct 8.

We must now show the invariance of Conjunct 2 over each step of the algorithm. To do this we will show that w∈Ω is satisfiable at point A if and only if w' (the value of w at point B) is satisfiable. Symbolically, we show

\[ \mathcal{E}w \iff \mathcal{E}w' \]

We will prove each direction of this co-implication separately.

Suppose \( \mathcal{E}w \), and that an assignment \( \phi \) satisfies \( w \) at point A in the program. Then by definition of \( \mathcal{E}w \), all of the terms in E have the same value under \( V_\phi \), the valuation induced
by \( \varphi \). Call this \( V \)-value \( \gamma \). Obviously \( \text{lb}(\gamma) = V_\varphi(l) \) and \( \text{ub}(\gamma) = V_\varphi(u) \).

Now consider an assignment \( \psi \) which agrees with \( \varphi \) everywhere, but in addition assigns the value \( \gamma \) to the new variable \( v' \), i.e. \( \psi(v') = \gamma \). This assignment is permissible because \( \gamma \) cannot be the value \( E_v \); Conjects 8 and 10 of the Invariant guarantee that. We claim the \( \psi \) satisfies \( w' \), and thus \( Ew' \). First, \( \psi \) satisfies the conjuncts \( \text{lb}(v') = l \) and \( \text{ub}(v') + 1 = u + 1 \). We can compute like this

\[
V_\psi(\text{lb}(v')) = \text{lb}(V_\psi(v')) = V_\psi(l) = V_\varphi(l)
\]

to show that \( \psi \) satisfies \( \text{lb}(v') = l \). The last line of the derivation takes advantage of the fact that \( v' \) does not occur in \( I \), and that \( \varphi \) and \( \psi \) agree on all variables other than \( v' \). A similar derivation would show that \( \psi \) satisfies \( \text{ub}(v') + 1 = u + 1 \).

We now must show that \( \psi \) satisfies the remainder of \( w \). But this is simple because \( \psi(v') = \gamma \) and for all \( e \in E \), \( \varphi(e) = \gamma \); hence for all \( e \in E \), \( \varphi(e) = \psi(v') \). Since \( v' \) has been substituted only for terms in \( E \), \( \psi \) must satisfy \( w' \). This concludes the proof that \( Ew \Rightarrow Ew' \).

Now we must prove the reverse implication: \( Ew' \Rightarrow Ew \). We will do this by assuming that an assignment \( \psi \) satisfies \( w' \) and constructing a variant \( \varphi \) of \( \psi \) which satisfies \( w \).

Let us denote \( \psi(v') \) by \( \gamma \), and let \( \varphi \) be an assignment which agrees with \( \psi \) everywhere except on the variable \( v \). The value of \( \varphi(v) \) is to be defined (implicitly) by the following identities:

\[
\begin{align*}
V_\varphi(\text{lb}(v)) &= V_\psi(\text{lb}(v)) \quad (= \text{lb}(\gamma)) \\
V_\varphi(\text{ub}(v)) &= V_\psi(\text{ub}(v)) \quad (= \text{ub}(\gamma)) \\
V_\varphi(v[i]) &= V_\psi(v[i]) \quad \text{if } i < \text{lb}(\gamma) \text{ or } \text{ub}(\gamma) < i \\
V_\varphi(v[i]) &= V_\psi(v'[i]) \quad \text{if } \text{lb}(\gamma) \leq i \leq \text{ub}(\gamma) \left(= \gamma[i] \right)
\end{align*}
\]

In other words, the vector assigned by \( \varphi \) to the variable \( v \) has the same bounds as that
vector assigned by \( \psi \), and has the same elements as well, except between \( \text{lb}(\gamma)\) and \( \text{ub}(\gamma)\) where it has the same elements as \( \gamma \).

Because of the way \( w' \) was formed from \( w \) and because \( \phi \) agrees with \( \psi \) everywhere except at \( v \), we can show that \( \phi \) satisfies \( w' \) if and only if for all \( e \in E, V_\phi(e) = V_\psi(e') \) and for all terms \( e' \) containing \( v \) such that \( e' \notin E, V_\phi(e') = V_\psi(e') \). Since all terms containing \( v \) are based on the terms \( \text{lb}(v) \) or \( \text{ub}(v) \) or on terms of the form \( v[i_1...i_2] \), we can further reduce the second part of that requirement to showing \( V_\phi(\text{lb}(v)) = V_\psi(\text{lb}(v)) \) and \( V_\phi(\text{ub}(v)) = V_\psi(\text{ub}(v)) \) and for \( v[i_1...i_2] \neq w \) but \( \notin E, V_\phi(v[i_1...i_2]) = V_\psi(v[i_1...i_2]) \). Therefore, to recapitulate, we must show four things:

1. \( V_\phi(\text{lb}(v)) = V_\psi(\text{lb}(v)) \),
2. \( V_\phi(\text{ub}(v)) = V_\psi(\text{ub}(v)) \),
3. \( V_\phi(v[i_1...i_2]) = V_\psi(v'(Y)) \) for \( v[i_1...i_2] \in E \), and
4. \( V_\phi(v[i_1...i_2]) = V_\psi(v[i_1...i_2]) \) for \( v[i_1...i_2] \neq w \) but \( \notin E \).

The first two are trivial, since they are included in the definition of \( V_\phi \) above.

We show the third identity, \( V_\phi(v[i_1...i_2]) = V_\psi(v'(Y)) \) for \( v[i_1...i_2] \in E \), with the following argument: \( \psi \) satisfies \( w' \) and hence satisfies the \( \gamma \)-part of \( w' \), including the conjuncts \( \text{lb}(v') = l \) and \( \text{ub}(v') = u \). Hence \( V_\psi(\text{lb}(v')) = V_\psi(l) = \text{lb}(\gamma) \) and \( V_\psi(\text{ub}(v')) = V_\psi(u) = \text{ub}(\gamma) \). But \( l \) and \( u \), being endpoints of some term in \( E \), cannot contain occurrences of \( v \) except possibly in the form \( \text{lb}(v) \) or \( \text{ub}(v) \), for to do so would contradict Conjunct 6 which states that \( \circ[^n] \) does not occur nested. Since \( V_\phi \) and \( V_\psi \) agree on all terms not involving \( v \), and also on the terms \( \text{lb}(v) \) and \( \text{ub}(v) \), they must agree on the terms \( l \) and \( u \). Hence \( V_\phi(l) = V_\psi(l) \) and \( V_\phi(u) = V_\psi(u) \).

Except for the conjuncts \( \text{lb}(v') = l \) and \( \text{ub}(v') = l + 1 = u + 1 \) the \( \gamma \)-part of \( w' \) is identical to the \( \gamma \)-part of \( w \); if it were not then there would have to have been an occurrence of \( \circ[^n..^n] \) in the \( \gamma \)-part of \( w \), contradicting Conjunct 6 again. Furthermore, since \( V_\phi \) and \( V_\psi \) agree on all variables but \( v \), and agree also on \( \text{lb}(v) \) and \( \text{ub}(v) \), \( \phi \) must satisfy the \( \gamma \)-part of \( w \). But the \( \gamma \)-part of \( w \) implies that \( i_1 = l \) and \( i_2 = u \); this follows from the definition of the equivalence class.
E over the relation $\equiv_w$, and from the fact that $v[i_1..i_2] \in E$ and $v[l..u] \in E$. We can now conclude that

\[ V_{\varphi}(\text{lb}(v[i_1..i_2])) = V_{\psi}(i_1) = V_{\psi}(\text{lb}(v')) = \text{lb}(\gamma), \]

and

\[ V_{\varphi}(\text{ub}(v[i_1..i_2])) = V_{\psi}(i_2) = V_{\psi}(\text{ub}(v')) = \text{ub}(\gamma). \]

In other words, the vectors $V_{\varphi}(v[i_1..i_2])$ and $V_{\psi}(v')$ have the same upper and lower bounds, namely $\text{lb}(\gamma)$ and $\text{ub}(\gamma)$ respectively.

But by the definition of $\varphi$, $\varphi(v)$ has the same elements as $\gamma$ between the indices $\text{lb}(\gamma)$ and $\text{ub}(\gamma)$. Hence

\[ V_{\varphi}(v[i_1..i_2]) = \gamma = V_{\psi}(v'), \]

which was our goal.

If there is a single climactic point in Chapter 6, here it is in the argument justifying the fourth identity, namely

\[ V_{\varphi}(v[i_1..i_2]) = V_{\psi}(v[i_1..i_2]) \text{ for } v[i_1..i_2] \approx w \text{ but } \not\in E. \]

If $i_1$ or $i_2$ contain any occurrences of $v$, it is only in the context of $\text{lb}(v)$ or $\text{ub}(v)$; otherwise there would be nesting of the $^o[.\.].^o$-function. Therefore we can conclude that $V_{\varphi}(i_1) = V_{\psi}(i_1)$ and $V_{\varphi}(i_2) = V_{\psi}(i_2)$. Thus, the two vectors $V_{\varphi}(v[i_1..i_2])$ and $V_{\psi}(v[i_1..i_2])$ have the same $\text{lb}$ and $\text{ub}$.

We now need to show that they have the same elements.

We already proved above that $\varphi$ must satisfy the $Y$-part of $w$. By Conjunct 12 either

\[ Y_w \approx i_1^o = 1 \land (i_2 + 1)^o = u + 1, \]  
\[ Y_w \approx u + 1 \leq i_1^o \lor (i_2 + 1)^o \leq l. \]

The first of these possibilities cannot hold because of the assumption that $v[i_1..i_2] \not\in E$. Therefore the second possibility holds. $\varphi$'s satisfying the $Y$-part of $w$ yields

\[ V_{\varphi}(u) < V_{\psi}(i_1) \lor V_{\varphi}(i_2) < V_{\psi}(l). \]

But

\[ V_{\varphi}(u) = V_{\psi}(u) = \text{ub}(\gamma) \]
and
\[ V\varphi(l_1) = V\psi(l_1) = \text{lb}(\gamma). \]

We then deduce that
\[ \text{ub}(\gamma) < V\varphi(i_1) \vee V\varphi(i_2) < \text{lb}(\gamma) \]

This means that all of the indices between \( V\varphi(i_1) \) and \( V\varphi(i_2) \) fall outside the bounds \( \text{lb}(\gamma), \text{ub}(\gamma) \) and by the third clause in the definition of \( \varphi \)
\[ V\varphi(v[i_1..i_2])[i] = V\psi(v[i_1..i_2])[i] \quad \text{for all } i \in V\varphi(i_1)_1..V\varphi(i_2), \]

which is the long way of writing
\[ V\varphi(v[i_1..i_2]) = V\psi(v[i_1..i_2]) \]

which was our goal.

This concludes the proof that \( Fw' \Rightarrow Fw' \), that \( Fw \Leftrightarrow Fw \), and that Conjunct 2 is invariant over the body of the middle loop of the algorithm.

6.10.4.2 Postcondition

We now show that the Postcondition asserted for this step is achieved by the Algorithm.

Conjuncts 1 and 2 were proved invariant, and hence hold at termination.

Conjunct 3 holds because no functions other than those mentioned were introduced by the algorithm, and because the algorithm cannot halt with any occurrences of \( o[\ldots o] \) remaining in \( \Omega \) since that would imply that some equivalence class \( E \) for some \( w \in \Omega \) had not been processed.

Conjunct 4 holds as a consequence of Conjunct 3, the fact that \( o[\ldots o] \) has been removed from all wffs. There are no V-valued function symbols remaining, so all V-terms must be simple variables.

Conjunct 5 follows from the invariance of the "old" Conjunct 8 and the fact that the only V-terms are V-variables. There will no longer be a need to distinguish the Y-part from the
rest of the wff, so it is from now on considered to be folded into U.

Conjunct 6 follows from the "old" Conjunct 6 of the Invariant. For V-variables which occurred in w before this step Conjunct 6 still applies directly, since every Y-conjunct of the Y-parts of wffs of Ω before this step are still conjuncts in the wffs of Ω after this step. For the new V-variables introduced by the algorithm of this step, Conjunct 6 holds because the atoms lb(v')=1 and ub(v')+1=u+1 were explicitly added in each iteration for each new variable v', and Conjunct 6 already required that Y_w(H=u; hence Y_wHb(v')=ub(v')).

6.10.5 Termination

Termination is a trivial consequence of the fact that Ω is finite, that for each w∈Ω there can be only a finite number of $\equiv_w$-equivalence classes, and each such class must be finite.
6.10.5 2

6.11 Step 11: Remove \( \varepsilon_V \)

At this stage all V-terms are simple V-variables and hence the left- and right-hand-sides of \( \varepsilon_V \)-atoms must be simple V-variables. Furthermore, \( \varepsilon_V \)-atoms occur only as unsigned top-level conjuncts. Under these conditions the \( \varepsilon_V \)-atom conjuncts can be removed by an easy variable substitution procedure detailed below, and after this step there will be no occurrences of the \( \varepsilon_V \)-function.

6.11.1 Specifications

**Precondition:** Same as Postcondition of previous step

**Invariant:** Same as Precondition

**Postcondition:**

1. \( \Omega \leq \text{BITZV} \) with \( \text{elem} \); \( \Omega \) unquantified, finite.

2. \( Fw_0 \iff \not \Omega \).

3. The only V-relevant functions occurring in \( \Omega \) are \( \text{ord} \), \( \text{lb} \), \( \text{ub} \), \( \text{elem} \) and \( \{^o\}_v \). [Note: \( \varepsilon_V \) has been removed.]

4. All V-terms are simple V-variables.

5. All \( w \in \Omega \) are of the form

\[
U \land \text{ORD}
\]

where \( U, \text{ORD} \) are as before.

6. For all variables \( v \) such that \( \text{elem}(v) \equiv w \in \Omega \): \( Fw \supset \text{lb}(v) = \text{ub}(v) \).

6.11.2 Notation

As usual, in the following algorithm the program variables \( \text{vvar} \), \( \text{vvar}_1 \) and \( \text{vvar}_2 \) must match only V-variables (of the object language), not general V-terms. However, since all V-terms are V-variables at this point, the distinction is academic.

The phrase "\( \text{vvar}_1 \) distinct from \( \text{vvar}_2 \)" means "the variable matched by \( \text{vvar}_1 \) is (an occurrence of) a different variable from the one matched by \( \text{vvar}_2 \)."
Production (a) is a standard production; production (b) is a tree-replacement production. There seems to be no reason not to mix the two as long as it is understood that the variable \( w \) is implicitly mentioned in production (b).

6.11.3 The Algorithm

For all \( w \in \Omega \) perform the following transformation

\[
\text{do} \begin{cases} 
(a) & (v_{var_1}=v_{var_2}=w) \land (v_{var_1} \text{ distinct from } v_{var_2}) \Rightarrow \\
& \quad w \cdot \text{subst}(w, v_{var_2}=v_{var_1}) \\
(b) & v_{var}=v_{var_1} \rightarrow \text{true} 
\end{cases} 
\text{end}
\]

6.11.4 Proof of Weak Correctness

The invariance of Conjuncts 1, 3, 4 and 5 of the Invariant is trivial.

The invariance of Conjunct 2 over production (b) follows from the fact that production (b) is an identity substitution.

The invariance of Conjunct 2 over production (a) depends on the invariance of Conjunct 5 which specifies that \( \varepsilon \)-atoms occur only as unsigned top-level conjuncts. From this we reduce the invariance of Conjunct 2 to the truth of the following biconditional schema

\[ \Phi(w \land v_{var_1}=v_{var_2} \leftrightarrow \Phi_{\text{subst}}(ww,v_{var_2}=v_{var_1}) \land v_{var_1}=v_{var_1}) \]

for any wff \( ww \) and \( V \)-variables \( v_{var_1} \) and \( v_{var_2} \).

This schema can be proved easily. If \( \varphi \) is an assignment satisfying \( ww \land v_{var_1}=v_{var_2} \) then \( \varphi \) also satisfies \( \text{subst}(ww, v_{var_2}=v_{var_1}) \land v_{var_1}=v_{var_1} \). This proves the \( \Rightarrow \) direction. Furthermore, if \( \varphi \) satisfies \( \text{subst}(w, v_{var_2}=v_{var_1}) \land v_{var_1}=v_{var_1} \) then \( \psi \) satisfies \( ww \land v_{var_1}=v_{var_2} \), where \( \psi \) assigns the same values to variables as \( \varphi \) except that \( \psi(v_{var_2})=\varphi(v_{var_1}) \). This proves the \( \Leftarrow \) direction.
Conjunct 6 of the Invariant is invariant over production (b) because the production is an identity replacement. For invariance over production (a) we prove that for any wff \( \varphi \), and any wff \( A \) not containing \( \text{vvar}_2 \),

\[
\mathcal{E}(\varphi \land \text{vvar}_1 \equiv \text{vvar}_2) \Rightarrow \mathcal{E}(\text{subs}_1(\varphi, \text{vvar}_2 := \text{vvar}_1) \land \text{vvar}_1 \equiv \text{vvar}_1) \Rightarrow A.
\]

This covers Conjunct 6 where \( A \equiv \text{lb}(v) = \text{ub}(v) \). The proof is trivial. If any assignment \( \varphi \) fails to satisfy the wff in the right-hand-side, then an assignment \( \psi \) would fail to satisfy the left-hand-side, where \( \psi \) agrees with \( \varphi \) except that \( \psi(\text{vvar}_2) = \varphi(\text{vvar}_1) \). This completes the proof that the Invariant is indeed invariant.

All of the Postcondition is included in the Invariant except the requirement (Conjunct 3) that there be no occurrences of \( \varepsilon_v \) at termination. It is easy to prove that this must hold. Any occurrence of \( \varepsilon_v \) has two arguments which are either the same V-variable or different V-variables. In the first case production (b) applies and in the second case production (a) applies; hence the PS cannot terminate in either case.

6.11.5 Termination

Each production reduces the sum of the number of distinct V-variables occurring in \( w \) and the number of occurrences of \( \varepsilon_v \) in \( w \).
6.12 Step 12: Remove ORD

In this step we remove all occurrences of the ord function. We do this simply by substituting true for every ord-atom. Intuitively, the reason we can do this is that the values returned by all of the other V-relevant functions remaining at this point, namely lb, ub, elem and $\{o\}_v$, do not depend upon the order of the elements in the vector. Thus, a wff over these V-functions is satisfiable in the presence of additional ord-constraints if and only if it is satisfiable without them.

The constraints on the order of the elements in vectors are not being lost when we delete the ord-atoms. All of the constraints are still encoded in other parts of the wff in inequalities of the form $\max_z\{\{v_1\}\} \leq \min_z\{\{v_2\}\}$ which were introduced in Step 9, productions (a) and (b). The ord-atoms are simply redundant now.

6.12.1 Specifications

Precondition: Same as Postcondition of previous step

Invariant: Same as Precondition

Postcondition:

1. $\Omega \subseteq \text{BITZV}$ with elem; $\Omega$ unquantified, finite.
2. $\emptyset \models w \iff \not \in \Omega$.
3. The only V-relevant functions occurring in $\Omega$ are lb, ub, elem and $\{o\}_v$. [Note: ord has been dropped.]
4. All V-terms are simple V-variables.
5. For all v such that elem(v) $\models w \in \Omega$: $\emptyset \supseteq lb(v) = ub(v)$.

6.12.2 The Algorithm

For all $w \in \Omega$ perform the following transformation:

$$\text{do}($$

$$\text{ord}(v) :\rightarrow \text{true}$$
6.12.3 Proof of Weak Correctness

Conjuncts 1, 3, 4 and 5 of the Invariant are trivially invariant.

The invariance of Conjunct 2 is proved in the following manner. We show that

$$\Downarrow U \land \text{ORD} \land \text{ord}(v) \iff \Uparrow U \land \text{ORD}$$

where \( v \) is any \( V \)-variable, \( \text{ORD} \) is a conjunction of unsigned atoms of the form \( \text{ord}(v_1) \), and \( U \) is the rest of the wff, containing no occurrences of any \( V \)-relevant functions except \( \text{lb}, \text{ub}, \text{elem} \) and \( \{\circ\}_v \).

Obviously any assignment \( \varphi \) satisfying the wff on the left-hand-side also satisfies the wff on the right-hand-side. This proves the \( \Rightarrow \) direction.

Conversely, if there is any assignment \( \psi \) satisfying the right-hand-side, then there is a \( \varphi \) satisfying the left-hand-side, where \( \varphi \) is defined as follows:

\[
\varphi(x) = \psi(x), \quad \text{for all variables } x \text{ except Type-}V\text{-variables} \\
\varphi(v) = \text{sort}(\psi(v)) \quad \text{for all } V\text{-variables}.
\]

The function "sort" returns a vector with the same \( \text{lb} \) and \( \text{ub} \) as its argument, but with the elements rearranged into nondecreasing order according to \( <_T \).

Obviously \( \varphi \) satisfies \( \text{ORD} \) and \( \text{ord}(v) \); the only question is whether or not it satisfies \( U \). To prove this, we take advantage of the fact that the only \( V \)-relevant functions occurring in \( U \) are \( \text{lb}, \text{ub}, \text{elem} \) and \( \{\circ\}_v \). It is obvious that for any vector value \( \gamma \),

\[
\begin{align*}
\text{lb}(\text{sort}(\gamma)) &= \text{lb}(\gamma), \\
\text{ub}(\text{sort}(\gamma)) &= \text{ub}(\gamma), \\
\text{elem}(\text{sort}(\gamma)) &= \text{elem}(\gamma), \quad \text{and} \\
\{\text{sort}(\gamma)\}_v &= \{\gamma\}_v.
\end{align*}
\]

We thus conclude that

\[
V\varphi(U) = V\psi(U)
\]
and hence $\varphi$ satisfies $U$ if $\psi$ satisfies $U$. This completes the proof in the $\Leftarrow$ direction, and completes the proof of invariance for Conjunct 2.

The argument that Conjunct 6 is invariant hinges on proving that

$$\exists U \land \text{ORD} \land \text{ord}(v_1) \Rightarrow \text{lb}(v_2) = \text{ub}(v_2) \Rightarrow \exists U \land \text{ORD} \land \text{lb}(v_2) = \text{ub}(v_2)$$

for any wffs $U$, ORD satisfying the pattern of Conjunct 5 of the Invariant, and any V-variables $v_1$ and $v_2$. The proof is similar to that for Conjunct 2. If an assignment $\psi$ fails to satisfy the wff on the right-hand-side, then another assignment $\varphi$ fails to satisfy the wff on the left-hand-side, where $\varphi$ agrees with $\psi$ except that $\varphi(v_1) = \text{sort}(\psi(v_1))$.

The difference between the Postcondition and the Invariant is the requirement in Conjunct 3 that there be no occurrences of ord in any $w \in \Omega$ upon termination. Obviously, if there were any such occurrences the production would apply and the PS would not halt.

6.12.4 Termination

Each production execution reduces the number of occurrences of ord in $w$. 
6.13 Step 13: Remove elem

In this step we will remove all occurrences of the elem function and also some occurrences of the \{o\}-function.

By Conjunct 5 of the Precondition to this step we know that whenever elem(v) occurs in some w \in \Omega, the rest of the formula "requires" that v have length exactly one. Hence, the term elem(v) acts like a simple T-variable. This should not be too surprising since early on (Step 1) we converted indexed terms of the form v[i] into terms of the form elem(v[i..]), and we are familiar with the phenomenon of indexed terms acting like simple T-variables. We take advantage of this by replacing terms of the form elem(v) by new T-variables and corresponding terms of the form \{v\}_v by terms based on the function \{o\}_T.

6.13.1 Specifications

Precondition: Same as Postcondition for Step 12
Invariant: Same as Precondition
Postcondition:

1. \(\Omega \subseteq \text{BITZV}; \Omega \) unquantified, finite.
2. \(\exists w_0 \iff \# \Omega\).
3. The only V-relevant functions occurring in \(\Omega\) are \(\text{lb, ub}\) and \(\{o\}_v\). [Note: elem has been removed.]
4. All V-terms in \(\Omega\) are simple V-variables.

6.13.2 Notation

In the next algorithm \(t'\) represents a new T-variable not occurring in \(w\). A different \(t'\) is created for each production execution; however, the \(t'\) which replaces occurrences of \(\text{elem}(v)\) and the \(t'\) in the term \(\{t'_1\}_T\) which replaces \(\{v\}_v\) are the same variable.

The notation REPLALL(\(w, a\rightarrow b, c\rightarrow d\)) means the result of replacing all occurrences of \(a\) in \(w\)
by b, and all occurrences of c in w by d, simultaneously. No ambiguity can arise in the case below since no two terms of the form elem(v) or \( \{v\}_v \) can overlap.

6.13.3 The Algorithm

For each \( w \in \Omega \) perform the following transformation

\[
\text{do(}
\quad \text{elem}(v) \Rightarrow \\
\quad w \leftarrow \text{lb}(v) = \text{ub}(v) \land \\
\quad \text{REPLALL}(w, \text{elem}(v) \Rightarrow t', \{v\}_v \Rightarrow \{t'\}_T) \\
\text{od)
}\]

6.13.4 Proof of Weak Correctness

Conjuncts 1, 3 and 4 of the Invariant are trivially invariant. We need only prove invariance of Conjuncts 2 and 5.

We prove invariance of Conjunct 5 first. We must show that if \( \text{elem}(v) = w \) and for all \( V \)-variables \( v \)

\[
\text{elem}(v) = w \Rightarrow \exists w' \forall \text{lb}(v) = \text{ub}(v) \\
\text{then, for all } V \text{-variables } v_2
\]

\[
\text{elem}(v_2) = w' \Rightarrow \exists w' \forall \text{lb}(v_2) = \text{ub}(v_2)
\]

where \( w' \) is

\[
\text{lb}(v) = \text{ub}(v) \land \text{REPLALL}(w, \text{elem}(v) \Rightarrow t', \{v\}_v \Rightarrow \{t'\}_T).
\]

Let us use the notation \( \varphi \models w \) to mean assignment \( \varphi \) satisfies \( w \), and \( \varphi \not\models w \) to mean \( \varphi \) does not satisfy \( w \) (i.e. yields \text{false} or \text{error} when \( w \) is evaluated using \( \varphi \).)

We will actually prove a form of the contrapositive, namely if \( \text{elem}(v) = w \) and if there is a variable \( v_2 \) and an assignment \( \varphi' \) such that

\[
\text{elem}(v_2) = w' \text{ and } \varphi' \models w' \text{ and } \varphi' \not\models \text{lb}(v_2) = \text{ub}(v_2)
\]
proves (2).

Part (3) is trivial. Since \( v_1 \) is \( v_2 \) and neither is the same variable as \( v \), \( \varphi \) and \( \varphi' \) agree on the value assigned to \( v_1 \) and \( v_2 \). Hence, since \( \varphi' \neq \varphi \), \( \varphi' \neq lb(v_1) = ub(v_1) \). This proves (3), and concludes the proof of the invariance of Conjunct 5.

To prove the invariance of Conjunct 2 we will use that of Conjunct 5. We must show that

\[ \equiv w \iff \equiv w', \] which means

\[ \equiv w \iff lb(v) = ub(v) \land REPLAIB(w, \text{elem}(v) \mapsto t', \{v\} \mapsto \{t'\}) \]

To show the \( \Rightarrow \) direction we note that if \( \varphi \equiv w \), then \( \varphi(v) \) is a one-element vector. (Here we used Conjunct 5 of the Invariant.) Thus, if we define an assignment \( \varphi' \) such that

\[ \varphi'(x) = \varphi(x) \text{ for all } x \text{ except } t' \]

\[ \varphi'(t') = \text{elem}(\varphi(v)) \]

it is clear that \( \varphi' \equiv w' \).

To show the \( \Leftarrow \) direction we do essentially the reverse. If \( \varphi' \) satisfies \( w' \), then define \( \varphi \) by

\[ \varphi(x) = \varphi'(x) \text{ for all } x \text{ except } v \]

\[ \varphi(v) = \gamma \text{ such that } lb(\gamma) = V_{\varphi'}(lb(v)) \]

\[ \quad \text{ub}(\gamma) = V_{\varphi'}(ub(v)) \]

\[ \quad \text{elem}(\gamma) = \varphi'(t') \]

This definition is well-defined since \( \varphi' \equiv w' \) and \( w' \) includes the conjunct \( lb(v) = ub(v) \). And clearly \( \varphi \equiv w \) if \( \varphi' \equiv w' \) by arguments similar to those given before for Conjunct 5. This completes the proof of the invariance of Conjunct 2, and the invariance of the entire Invariant.

The only significant difference between the Invariant and the Postcondition is the requirement (in Conjunct 3) that there be no occurrences of \( \text{elem} \) at termination. Obviously the PS cannot terminate with any occurrence of \( \text{elem} \) in \( w \), because the pattern in the (only) production matches any occurrence of \( \text{elem} \).
6.13.5 Termination

Each production reduces the number of occurrences of elem in $\Omega$. 
6.14 Step 14: Remove \{v\}

In this step we replace all occurrences of terms of the form \{v\}_v with new Z-variables z', conjoining the side conditions that size(z')=ub(v)-lb(v)+1 and that \(\phi_Z \leq z'\).

6.14.1 Specifications

Precondition: Same as Postcondition for Step 13

Invariant: Same as Precondition

Postcondition:

1. \(\Omega \subseteq \text{BITZV}; \Omega\) unquantified, finite.
2. \(\forall w_0 \iff \not\exists \Omega\).
3. The only V-relevant functions occurring in \(\Omega\) are \(lb\) and \(ub\).
   [Note: \(\{\circ\}_v\) has been removed.]
4. A V-terms are simple V-variables.

6.14.2 The Algorithm

For each \(w \in \Omega\) perform the following transformation:

\[
do\left\{{v}\right\}_v = w \Rightarrow \\
\quad w \leftarrow \text{size}(z')=ub(v)-lb(v)+1 \land \phi_Z \leq z' \land \\
\quad \text{REPLALL}(w, \{v\}_v :\rightarrow z') \\
\od
\]

Note: \(z'\) is a new Z-variable for each production execution.

6.14.3 Proof of Weak Correctness

The invariance of Conjuncts 1, 3 and 4 of the Invariant is trivial.

For Conjunct 2 we must show that

\[
\vdash w \iff \vdash \text{size}(z')=ub(v)-lb(v)+1 \land \phi_Z \leq z' \land \text{REPLALL}(w, \{v\} :\rightarrow z')
\]
We will use the fact that the only $V$-relevant functions occurring in $w$ are $lb$, $ub$ and $\{v\}$. To prove the $\Rightarrow$ direction we suppose that $\varphi \models w$. We will construct an assignment $\psi$ such that $\psi \models w'$ where $w'$ is

\[
\text{size}(z') = \text{ub}(v) - \text{lb}(v) + 1 \land \phi_2 \leq z' \land \text{REPLALL}(w, \{v\} \mapsto z')
\]

Define $\psi$ so that it agrees with $\varphi$ everywhere except possibly at $z'$. Define $\psi(z') = V_{\varphi}(\{v\})$. Then we claim $\psi \models w'$. First,

\[
\psi \models \text{size}(z') = \text{ub}(v) - \text{lb}(v) + 1
\]

as shown by the following derivation.

\[
\begin{align*}
V_{\psi}(\text{size}(z')) &= \text{size}(\psi(z')) \\
&= \text{size}(V_{\varphi}(\{v\})) \\
&= V_{\varphi}(\text{size}(\{v\})) \\
&= V_{\varphi}(\text{ub}(v) - \text{lb}(v) + 1) \\
&= V_{\psi}(\text{ub}(v) - \text{lb}(v) + 1)
\end{align*}
\]

The last step is justified because $\varphi$ and $\psi$ agree (by definition) on $v$.

To show $\psi \models \phi_2 \leq z'$ we use the following derivation.

\[
\begin{align*}
V_{\psi}(\phi_2) \leq V_{\psi}(z') & \iff \\
V_{\psi}(\phi_2) \leq \psi(z') & \iff \\
V_{\psi}(\phi_2) \leq V_{\varphi}(\{v\}) & \iff \text{true}
\end{align*}
\]

The last step is valid because the zset of elements in any vector is a genuine multiset (with no negative multiplicities). Hence, it dominates the empty zset.

To complete the proof that $\psi \models w'$ we point out that

\[
\psi \models \text{REPLALL}(w, \{v\} \mapsto z')
\]

because $\varphi \models w$, because $w$ contains no occurrence of $z'$ and because $\psi(z') = V_{\varphi}(\{v\})$. This completes the proof that $\models w \Rightarrow \models w'$.

To prove that $\models w' \Rightarrow \models w$, we assume that some assignment $\psi$ satisfies $w'$ and construct an assignment $\varphi$ such that $\varphi \models w$. Let $\varphi$ agree with $\psi$ everywhere except (possibly) on $v$. Define $\varphi(v)$ as follows:
\[
\begin{align*}
\text{lb}(\phi(v)) &= \text{lb}(\psi(v)) \\
\text{ub}(\phi(v)) &= \text{ub}(\psi(v)) \\
(\phi(v))[\text{lb}(\phi(v))-1+i] &= \text{i}th \text{ smallest element of } \psi(z') \\
&= \text{ub}(\psi(v)) \\
&= \text{ub}(\psi(v))
\end{align*}
\]

according to \(\leq_T\) for \(1 \leq i \leq \text{size}(\psi(z'))\).

We must first argue that the third requirement is well-defined. Since \(\psi = w'\), \(\psi(z')\) is a multiset, having no elements of negative multiplicity. Hence, it makes sense to refer to the \(i\)th smallest element of \(\psi(z')\), for \(1 \leq i \leq \text{size}(\psi(z'))\). We know also (because \(\psi = w'\)) that \(\text{size}(\psi(z'))\) is exactly \(\text{len}(\psi(v)) - \text{len}(\phi(v))\). Thus, the elements of \(\psi(z')\) exactly "fill" the space in \(\phi(v)\), and the third requirement is well-defined.

This definition for \(\phi(v)\) implies that
\[
V_{\phi} \{v\} = V_{\psi}(z').
\]

Furthermore, on any term not involving \(\{v\}\) or \(z'\), \(V_{\phi}\) and \(V_{\psi}\) agree. (And \(w\) does not contain \(z'\).) We can therefore derive the following:

\[
\begin{align*}
\psi &= w' \\
\Rightarrow \psi &= \text{REPLALL}(w, \{v\} : z') \\
\Rightarrow \phi &= w \\
\end{align*}
\]

This completes the proof that \(\equiv w' \Rightarrow \equiv w\), that \(\equiv w \iff \equiv w'\), that Conjunct 2 is invariant, and that the entire Invariant assertion is invariant.

The difference between the Invariant and the Postcondition is in the requirement that there be no occurrences of \(\{v\}_v\) upon termination. Obviously any such occurrence would make the (single) production eligible to fire, and thus the PS could not terminate.

6.14.4 Termination

Each production execution reduces the number of occurrences of \(\{v\}_v\) in \(w\).
6.15 Step 15: Remove lb and ub

Here we remove all occurrences of the last remaining V-relevant functions by replacing them with new integer variables. This is the final stage in the Type Reduction of BITZV to BITZ.

6.15.1 Specifications

Precondition: Same and Postcondition for Step 14

Invariant: Same as Precondition

Postcondition:

1. \( \Omega \subseteq \text{BITZ}; \Omega \) unquantified, finite.

2. \( \exists w_0 \iff \# \Omega. \)

3. There are no V-relevant functions occurring in any \( w \in \Omega. \)  
   [Note: \( lb \) and \( ub \) have been removed.]

6.15.2 The Algorithm

For all \( w \in \Omega \) perform the following transformation

\[
\text{do } \left( \begin{array}{l}
(a) \quad \text{lb}(v) \neq w \text{ and } \text{ub}(v) \neq w \Rightarrow \\
\quad \mu \leftarrow \text{REPLALL}(\mu, \text{lb}(v) : \rightarrow i') \ \\
(b) \quad \text{ub}(v) = w \text{ and } \text{lb}(v) \neq w \Rightarrow \\
\quad \mu \leftarrow \text{REPLALL}(\mu, \text{ub}(v) : \rightarrow i') \ \\
(c) \quad \text{lb}(v) = w \text{ and } \text{ub}(v) = w \Rightarrow \\
\quad \mu \leftarrow i' \leq i'' + 1 \land \\
\quad \text{REPLALL}(\mu, \text{lb}(v) : \rightarrow i', \text{ub}(v) : \rightarrow i'')
\end{array} \right) \ \\
\text{od}
\]

The symbols \( i' \) and \( i'' \) represent two new distinct integer variables not occurring in \( w. \) As usual, they are created anew each production cycle.
6.15.3 Proof of Weak Correctness

Conjuncts 1, 3 and 4 of the invariant are obviously invariant over the three productions. Hence, as usual, all the work involves showing that Conjunct 2 is invariant.

We will show that Conjunct 2 is invariant over productions (a) and (c) only, since (b) is similar to (a).

For production (a) we must show that if \( \text{lb}(v) \neq w \) and \( \text{ub}(v) \neq w \), then

\[
\forall w \iff \forall \text{REPLALL}(w, \text{lb}(v) \rightarrow i')
\]

For the \( \rightarrow \) direction, let \( \varphi \models w \). Define \( \psi \) to agree with \( \varphi \) everywhere except (possibly) on \( i' \), where we define \( \psi(i') = V_{\varphi}(\text{lb}(v)) \). Since \( i' \) does not occur in \( w \), it is trivial that \( \psi \notmodels \text{REPLALL}(w, \text{lb}(v) \rightarrow i') \).

For the \( \leftarrow \) direction, assume \( \psi \notmodels \text{REPLALL}(w, \text{lb}(v) \rightarrow i') \). Define \( \varphi \) to agree with \( \psi \) everywhere except (possibly) on \( v \). Define \( \varphi(v) \) to be any vector whose \( \text{lb} \) is \( \psi(i') \). Since by Conjunct 3 and by the hypothesis that \( \text{ub}(v) \neq w \) we know that all occurrences of \( v \) are in the context \( \text{lb}(v) \), we conclude that \( \varphi \notmodels w \). Hence, Conjunct 2 is invariant over production (a).

In order for Conjunct 2 to be invariant over production (c) we show that

\[
\forall w \iff \exists i'' + 1 \land \text{REPLALL}(w, \text{lb}(v) \rightarrow i', \text{ub}(v) \rightarrow i'').
\]

We will abbreviate the proof.

If \( \varphi \models w \), define \( \psi \) to agree with \( \varphi \) except possibly on \( i' \) and \( i'' \). Let \( \psi(i') = V_{\varphi}(\text{lb}(v)) \) and \( \psi(i'') = V_{\varphi}(\text{ub}(v)) \). Then \( \psi \notmodels i'' + 1 \land \text{REPLALL}(w, \text{lb}(v) \rightarrow i', \text{ub}(v) \rightarrow i'') \), proving the \( \Rightarrow \) direction.

For the \( \leftarrow \) direction, let \( \psi \notmodels i'' + 1 \land \text{REPLALL}(w, \text{lb}(v) \rightarrow i', \text{ub}(v) \rightarrow i'') \). Define \( \varphi \) to agree with \( \psi \) except (possibly) on \( v \). Define \( \varphi(v) \) to be any vector whose \( \text{lb} \) is \( \psi(i') \) and whose \( \text{ub} \) is \( \psi(i'') \). Then \( \varphi \models w \). This completes the proof of the invariance of Conjunct 2, and of the entire Invariant assertion.
The Postcondition differs from the invariant in the requirement that the lb and ub functions not occur in any w ∈ Ω. Obviously if either function did occur, one of the three productions (a)-(c) would apply, thereby preventing termination.

6.15.4 Termination

Each production reduces the total number of occurrences of lb and ub in Ω.
6.16 The Type Reduction is Complete

At the end of the last step we achieved the following state of affairs.

1. $\Omega \subseteq \text{BITZV}; \Omega$ unquantified, finite.
2. $\models w_0 \iff \not\exists \Omega$.
3. There are no $V$-relevant functions occurring in any $w \in \Omega$.

But if $\Omega \subseteq \text{BITZV}$ and there are no $V$-relevant functions (and hence no $V$-terms) occurring in any $w \in \Omega$, then in fact $\Omega \subseteq \text{BITZ}$. We can conclude that

1. $\Omega \subseteq \text{BITZ}; \Omega$ unquantified, finite, and
2. $\models w_0 \iff \not\exists \Omega$.

This is exactly what we set out to achieve at the beginning of Chapter 6. We have completed the Type Reduction from $\text{BITZV}$ to $\text{BITZ}$. 
7. Type Reduction: BITZ to BIT

In this Chapter we will present an algorithm for taking an arbitrary wff \( w_0 \in \text{BITZ} \) and constructing a finite set \( \Omega \subseteq \text{BIT} \) such that \( \forall \Omega \leftrightarrow \exists w_0 \). The format will be quite similar to that of the previous Chapter in that each Section represents one step of the Algorithm along with the accompanying Specifications and Correctness Proof. Although the Algorithm and Specification parts will still be given in great detail, the Correctness Proofs will be highly compressed. In most cases the techniques will be the same as those illustrated in the previous Chapter, the major exception being Step 14, for which a detailed proof is included.

The global specifications for the Algorithm in this Chapter are as follows:

**Precondition:** \( w_0 \in \text{BITZ} \), unquantified.

**Postcondition:**

1. \( \Omega \subseteq \text{BIT} \); \( \Omega \) finite, unquantified.
2. \( \exists w_0 \iff \forall \Omega \).

We now proceed with the algorithm.
7.1 Step 1: Make arguments to \( \max_z \) and \( \min_z \) be simple \( Z \)-variables

The \( \max_z \) and \( \min_z \) functions are the most troublesome we will have to deal with in Chapter 7. The first step is to alter the formula so that the \( \max_z \) and \( \min_z \) functions only occur with simple \( Z \)-variables as arguments.

7.1.1 Specifications

**Precondition:** \( w_0 \in \text{BITZV}, \) unquantified

**Invariant:**

1. \( w \in \text{BITZ}, \) unquantified
2. \( \forall w \Leftrightarrow \forall w_0 \)

**Postcondition:**

1. \( w \in \text{BITZ}, \) unquantified
2. \( \forall w \Leftrightarrow \forall w_0 \)
3. All arguments to \( \min_z \) or \( \max_z \) are simple \( Z \)-variables

7.1.2 The Algorithm

\(~\)

\( w \leftarrow w_0; \)

**do**

\( (\min(z) \leftrightarrow w \) or \( \max(z) \leftrightarrow w \)) and \( z \) not a variable \( \Rightarrow \)

\( w \leftarrow (\#z \)

\( | \quad z' = z \Rightarrow \text{REPLALL}(w, \min(z) : \rightarrow \min(z'), \)

\( \quad \max(z) : \rightarrow \max(z')) \)

\( | \quad \text{REPLALL}(w, \min(z) : \rightarrow E_T, \)

\( \quad \max(z) : \rightarrow E_T) \)

**od**
Note: $z'$ is a new $Z$-variable created each iteration.

7.1.3 Proof of Weak Correctness

The assignment statement establishes the identity of $w$ and $w_0$, so the Invariant is trivially true at the beginning of the loop.

The invariance of the first conjunct of the Invariant is also obvious, since no quantifiers or operators outside of BITZ are introduced.

The invariance of the second conjunct follows from the metatheoretic truth that for any wffs $w_1$, $w_2$ and variable $x$,

$$\text{if } \exists(w_1 = \forall x.w_2) \text{ then } \exists w_1 \iff \exists w_2$$

and from the identity schema

$$\exists w = \forall z'. \left( w_z' = z \Rightarrow \text{REPLALL}(w, \min(z) \mapsto \min(z'), \max(z) \mapsto \max(z')) \right)$$

which holds for any wff $w$, $Z$-term $z$ and $Z$-variable $z'$ not occurring in $w$ or $z$. Without a proof theory for BITZ we cannot prove the validity of the schema, but we can informally justify it: 1) When $z$ is defined and $z'$ equals $z$, then $\min(z)$ and $\max(z)$ have the same values respectively as $\min(z')$ and $\max(z')$, and thus the replacements are value-preserving; (2) when $z$ is undefined, then $\min(z)$ and $\max(z)$ have the same values as $E_T$, and once again the replacements are value-preserving.

To demonstrate that the Postcondition holds at termination we note that the first two conjuncts follow from the Invariant. The third conjunct holds because, if it did not, one of the two productions would be eligible to fire.
7.1.4 Termination

Each iteration reduces the size of the set of all (distinct) Z-terms which are not simple variables but which occur as arguments to \( \min_z \) and/or \( \max_z \).
7.2 Step 2: Make sure that the \( \text{min}(z) \), \( \text{max}(z) \), \( z^{<t>} \) and \( \{t\} \) functions do not occur nested in one another

In Step 1 of Chapter 6 we removed all cases in which the \( o^{[o..o]} \)-function occurred nested in \( w \) in such a way that no two index terms (i.e. the terms in \( I(w,v) \) for some \( v \)) would be nested. Here we will do essentially the same thing.

The terms in a set called \( TX(w) \) (to be defined later in Step 9) play a role analogous to that played by \( I(w,v) \) in the last chapter, and we therefore want to remove nesting of the terms in \( TX(w) \). However, unlike Chapter 6 where we could accomplish the goal by unnesting occurrences of one function symbol, namely \( o^{[o..o]} \), here we must unnest occurrences of four symbols: \( \text{min}_z \), \( \text{max}_z \), \( o^{<o>} \) and \( \{o\} \).

7.2.1 Specifications

**Invariant:**

1. \( w \in \text{BITZ} \), unquantified
2. \( \forall w \leftrightarrow \forall w_0 \)
3. All arguments to \( \text{min}_z \) and \( \text{max}_z \) are simple \( z \)-variables.

**Postcondition:**

1. \( w \in \text{BITZ} \), unquantified
2. \( \forall w \leftrightarrow \forall w_0 \)
3. All arguments to \( \text{min}_z \) and \( \text{max}_z \) are simple \( z \)-variables
4. The symbols \( \text{min}_z \), \( \text{max}_z \), \( o^{<o>} \) and \( \{o\} \) do not occur nested in \( w \).
7.2.2 The Algorithm

do(
(a) \((\min(z_1) \prec z_2 < t \text{ and } z_2 < t < u) \text{ or } \)
\((\min(z_1) \prec \{t\} \text{ and } \{t\} \prec u) \Rightarrow \)
\(w \leftarrow (\#\min(z_1))\)
\(\quad | t' = \min(z_1) \Rightarrow \text{REPLALL}(\omega, \min(z_1) : t')\)
\(\quad | \text{REPLALL}(\omega, \min(z_1) : E_1)\)
\)
(b) \((\max(z_1) \prec z_2 < t \text{ and } z_2 < t < u) \text{ or } \)
\((\max(z_1) \prec \{t\} \text{ and } \{t\} \prec u) \Rightarrow \)
\(w \leftarrow (\#\max(z_1))\)
\(\quad | t' = \max(z_1) \Rightarrow \text{REPLALL}(\omega, \max(z_1) : t')\)
\(\quad | \text{REPLALL}(\omega, \max(z_1) : E_1)\)
\)
(c) \((z_1 < t_1 \prec \{t_2\} \text{ and } \{t_2\} \prec u) \text{ or } \)
\((z_1 < t_1 \prec z_2 < t_2 \text{ and } z_1 < t_1 > \text{ different from } z_2 < t_2 \text{ and } z_2 < t_2 \prec u) \Rightarrow \)
\(w \leftarrow (\#z_1 < t_1)\)
\(\quad | i' = z_1 < t_1 \Rightarrow \text{REPLALL}(\omega, z_1 < t_1 : i')\)
\(\quad | \text{REPLALL}(\omega, z_1 < t_1 : E_1)\)
\)
)
(d) \( \{t_1\} = z^{<t_2>} \text{ and } z^{<t_2>} = \{t_1\} \text{ or} \)
\[
\{t_1\} = \{t_2\} \text{ and } t_1 \text{ different from } t_2 \text{ and } \{t_2\} = \{t_1\} \Rightarrow \\
\neg \exists f(t_1) \\
| \neg \exists f(t_2) \\
| \neg \exists f(z') \\
| \neg \exists f(z) \\
| \neg \exists f(t_1) \\
| \neg \exists f(t_2) \\
| \neg \exists f(z') \\
| \neg \exists f(z) \\
| \neg \exists f(t_1) \\
\]
}\)

Note: \( t', i' \) and \( z' \) are all new variables, created each production cycle and distinct from all other variables occurring in \( w \) at the time of creation.

### 7.2.3 Proof of Weak Correctness

The Invariant holds at the beginning because it is the Postcondition of the previous step.

It is obvious that Conjuncts 1 and 3 remain invariant over each of the four productions in this step, and the only question is about the invariance of Conjunct 2. We will demonstrate this informally for production (a) only, since the proof for the other cases is similar.

For production (a) the proof boils down to the claim that
\[
Fw \equiv \forall t'. (\#min(z_1)) \\
| t' = min(z_1) \Rightarrow REPLALL(w, min(z_1); -t') \\
| REPLALL(w, min(z_1); -c_T) \\
\]
for any wff \( w \) and new variable \( t' \),
from which we can conclude
\[
Fw \iff F(\#min(z_1)) \\
| t' = min(z_1) \Rightarrow REPLALL(w, min(z_1); -t') \\
| REPLALL(w, min(z_1); -c_T) \\
\]
which is what is required.

The argument for the claim is essentially identical to the argument given in Subsection 7.1.3. The equivalence is informally justified as follows: 1) When \( \text{min}(z_1) \) is \text{true} and \( t' = \text{min}(z_1) \) is \text{true} then \( t' \) and \( \text{min}(z_1) \) have the same value and one may be replaced by the other. This justifies the first branch of the conditional. 2) When \( \text{min}(z_1) \) is \text{false}, \( \text{min}(z_1) \) has the same value as \( E_T \), and hence one may be replaced by the other. This justifies the second branch of the conditional. 3) \( \text{min}(z_1) \) never takes the value \text{error}, so we do not have to consider a third branch to the conditional. Actually, the only reason we have to consider even two branches is that the new variable \( t' \) cannot take the error values under any assignment, whereas the term \( \text{min}(z_1) \) can, so an exception has to be made for the case where \( \text{min}(z_1) = E_T \).

This completes the proof that the Invariant is invariant over production (a), and, because of their similarity, over the other productions as well.

The first three conjuncts of the Postcondition hold at termination because they are part of the Invariant. The fourth conjunct follows from the fact that at termination all of the patterns must be unsatisfiable.

By Conjunct 3 of the Invariant and of the Postcondition, nothing of the form \( \text{min}(z_1) \) can occur as a \text{proper} subterm of any part of \( w \) of the form \( \text{min}(z_2) \) or \( \text{max}(z_2) \). Hence, \( \text{min} \) does not occur nested below \( \text{min} \) or \( \text{max} \) in \( w \). Furthermore, the pattern of production (a) prevents \( \text{min} \) from occurring below either \( o < o > \) or \( \{ o \} \) at termination, or else the pattern would be satisfiable and the PS could not have halted. Therefore \( \text{min} \) does not occur nested below any of \( \text{min}, \text{max}, o < o > \) or \( \{ o \} \) in \( w \).

Identical arguments with respect to production (b) demonstrate that \( \text{max} \) cannot occur below \( \text{min}, \text{max}, o < o > \) or \( \{ o \} \) in \( w \).

The function \( o < o > \) cannot occur below \( \text{min} \) or \( \text{max} \) in \( w \), again because of Conjunct 3 of the Invariant and Postcondition. And the pattern-part of production (c) guarantees that the PS
cannot terminate with $\circ<\circ>$ occurring below $\{\circ\}$ or below another occurrence of $\circ<\circ>$. Hence, $\circ<\circ>$ does not occur nested below $\text{min}$, $\text{max}$, $\circ<\circ>$ or $\{\circ\}$ in $w$.

And an identical argument applies to $\{\circ\}$ with respect to production (d).

Therefore, at termination none of the four functions $\text{min}_z$, $\text{max}_z$, $\circ<\circ>$ or $\{\circ\}$ can occur below any other occurrences of $\text{min}_z$, $\text{max}_z$, $\circ<\circ>$, $\{\circ\}$, as required. This establishes Conjunct 4 of the Postcondition.

7.2.4 Termination

Consider the set $S$ of all distinct terms $u$ of the forms $\text{min}(z)$, $\text{max}(z)$, $z^{<t>}$ or $\{t\}$ such that one or more occurrences of $u$ are proper subterms of another term of the form $\text{min}(z)$, $\text{max}(z)$, $z^{<t>}$ or $\{t\}$. The size of set $S$ decreases with each production execution.
7.3 Step 3: Remove \( = \) and the \( \mathbb{Z} \)-conditional functions

In this step we remove the conditional functions \( (\circ | \circ)_{\mathbb{Z}}, (\circ | \circ | \circ)_{\mathbb{Z}} \) and \( (\circ | \circ | \circ | \circ)_{\mathbb{Z}} \) from the wff \( w \). We will not introduce them again.

At the same time we will remove all occurrences of the \( \equiv_{\mathbb{Z}} \) function. In Chapter 6 we removed \( \equiv_{\mathbb{V}} \) in favor of \( \equiv_{\mathbb{V}} \); here we do the reverse, removing \( \equiv_{\mathbb{Z}} \) in favor of \( \equiv_{\mathbb{Z}} \). The main reason is that \( \equiv_{\mathbb{Z}} \) is a "weak" operation (takes the value error when either argument is \( \equiv_{\mathbb{Z}} \)), as is \( \leq_{\mathbb{Z}} \). Since we will be operating with equalities and inequalities in the \( \mathbb{Z} \)-domain in similar ways, it is convenient to have a pair of weak functions with identical error properties rather than to have one weak and one strong function.

7.3.1 Specifications

**Invariant (a):** Same as Postcondition of previous step.

**Invariant (b):** Same as Invariant (a) with additional conjunct that there are no occurrences of \( \equiv_{\mathbb{Z}} \).

**Invariant (c):**

1. \( \mathbb{W} \in \mathbb{BITZ}, \) unquantified.
2. \( \mathbb{F} w \iff \mathbb{F} w_0. \)
3. All arguments to \( \min_{\mathbb{Z}}, \max_{\mathbb{Z}} \) are simple \( \mathbb{Z} \)-variables.
4. The symbols \( \min_{\mathbb{Z}}, \max_{\mathbb{Z}}, <>, \circ \) and \( \{\circ\} \) do not occur nested in \( w \).
5. The only \( \mathbb{Z} \)-relevant functions occurring in \( w \) are \( \#_{\mathbb{Z}}, =, \leq, <>, \circ \), size, \( \min_{\mathbb{Z}}, (\circ | \circ | \circ | \circ)_{\mathbb{Z}}, \equiv_{\mathbb{Z}}, \phi, \neg, +, -, *, \{\circ\}_{\mathbb{T}}. \) [Note: \( \equiv_{\mathbb{Z}} \) and \( (\circ | \circ)_{\mathbb{Z}} \) and \( (\circ | \circ | \circ)_{\mathbb{Z}} \) have been removed.]

**Postcondition:**

1. \( \mathbb{W} \in \mathbb{BITZ}, \) unquantified.
2. \( \mathbb{F} w \iff \mathbb{F} w_0. \)
3. All arguments to \( \min_Z \) or \( \max_Z \) are simple \( Z \)-variables.

4. The only \( Z \)-relevant functions occurring in \( w \) are \( \#z, =, \leq, \lt, \lt \), size, min, max, \( E_Z \), \( \phi \), neg, \( +, - \), \( \{ . \} \). [Note: \( =_Z \), \( (o \lt o) \), \( (o \lt o) \) and \( (o \lt o) \) have been removed.]

### 7.3.2 The Algorithm

(a) \[
\begin{align*}
    z_1 = z_2 & \rightarrow (\#z_1 \land \#z_2 \mid z_1 = z_2 \mid \neg \#z_1 \land \neg \#z_2 )_B \\
\end{align*}
\]  \( \text{do} \);  

(b) \[
\begin{align*}
    (b \mid z)_\gamma & \rightarrow (b \mid z \mid E_\gamma \mid E_\gamma)_\gamma \emptyset \\
    (b \mid z_1 \mid z_2)_Z & \rightarrow (b \mid z_1 \mid z_2 \mid E_Z)_Z \\
\end{align*}
\]  \( \text{do} \);
do 

(c1) \( \#(b \mid z_1 \mid z_2 \mid z_3)_z \Rightarrow (b \mid \#z_1 \mid \#z_2 \mid \#z_3)_B \) ⊗

(c2) \( (b \mid z_1 \mid z_2 \mid z_3)_z \Rightarrow (b \mid z_1 = z_4 \mid z_2 = z_4 \mid z_3 = z_4)_R \) ⊗

(c3) \( z_4 = (b \mid z_1 \mid z_2 \mid z_3)_z \Rightarrow \text{Same as c2} \)

(c4) \( (b \mid z_1 \mid z_2 \mid z_3)_z \leq z_4 \Rightarrow (b \mid z_1 \leq z_4 \mid z_2 \leq z_4 \mid z_3 \leq z_4)_R \) ⊗

(c5) \( z_1 \leq (b \mid z_2 \mid z_3 \mid z_4)_z \Rightarrow (b \mid z_1 \leq z_2 \mid z_1 \leq z_3 \mid z_1 \leq z_4)_R \) ⊗

(c6) \( (b \mid z_1 \mid z_2 \mid z_3)_z \leq t \Rightarrow (b \mid z_1 \leq t \mid z_2 \leq t \mid z_3 \leq t)_1 \)

(c7) \( \text{size}(b \mid z_1 \mid z_2 \mid z_3)_z \Rightarrow (b \mid \text{size}(z_1) \mid \text{size}(z_2) \mid \text{size}(z_3))_1 \)

(c8) \( (b \mid z_1 \mid z_2 \mid z_3)_z + z_4 \Rightarrow (b \mid z_1 + z_4 \mid z_2 + z_4 \mid z_3 + z_4)_z \)

(c9) \( z_4 + (b \mid z_1 \mid z_2 \mid z_3)_z \Rightarrow \text{Same as c8} \)

(c10) \( \neg((b \mid z_1 \mid z_2 \mid z_3)_z) \Rightarrow (b \mid \neg(z_1) \mid \neg(z_2) \mid \neg(z_3))_z \)

(c11) \( (b \mid z_1 \mid z_2 \mid z_3)_z \leq z_4 \Rightarrow (b \mid z_1 \leq z_4 \mid z_2 \leq z_4 \mid z_3 \leq z_4)_z \)

(c12) \( z_1 - (b \mid z_2 \mid z_3 \mid z_4)_z \Rightarrow (b \mid z_1 - z_2 \mid z_1 - z_3 \mid z_1 - z_4)_z \)

(c13) \( i \times (b \mid z_1 \mid z_2 \mid z_3)_z \Rightarrow (b \mid i \times z_1 \mid i \times z_2 \mid i \times z_3)_z \)

) od

7.3.3 Proof of Weak Correctness

We will omit the proofs for parts (a) and (b) since they follow established patterns; we will only treat part (c).

The invariance of Conjunct 1 is trivial by inspection.

The invariance of Conjunct 2 follows from the fact that each production is an identity replacement.
The invariance of Conjuncts 3, 4 and 5 are also trivial by inspection.

The Postcondition differs from the Invariant only in the requirement that there be no occurrences of $(\circ | \circ | \circ | \circ)\_z$. If at termination there are any occurrences of $(\circ | \circ | \circ | \circ)\_z$, then the outermost occurrences must be as arguments to $\#$, $=\$, $\leq$, $\circ<\circ$, size, $\ast$, neg, $-$ or $\ast$. (They cannot be arguments to min or max by Conjunct 3.) Each of the possibilities is covered by one or two productions $(c1) - (c13)$. Hence, the TRS cannot halt with any occurrences of $(\circ | \circ | \circ | \circ)\_z$.

7.3.4 Termination

Part (a) terminates by the Dershowitz and Manna nested-multiset argument [Dershowitz 78], where the elements of the nested-multisets are counts of $\#$-symbols.

Part (b) terminates because the total number of $(\circ | \circ)\_z$ and $(\circ | \circ | \circ)\_z$-symbols occurring in $w$ is reduced by each production.

Part (c): [Omitted]
7.4 Step 4: Put \(-T(w)\) in DNF and separate the disjuncts

As in Section 6.2, it is convenient here in Section 7.4 to convert the problem from a validity problem to an unsatisfiability problem, and to split the formula into a set of disjuncts.

7.4.1 Specifications

**Precondition:**

Same as Postcondition of previous step.

**Postcondition:**

1. \(\Omega \subseteq \text{BITZ, finite, unquantified.}\)
2. \(\exists w_0 \iff \not \in \Omega.\)
3. All arguments to \(\min_z\) and \(\max_z\) are simple \(Z\)-variables.
4. The symbols \(\min_z, \max_z, \circ \circ, \circ \circ \circ\) and \(\{ \circ \}\) do not occur nested in any \(w \in \Omega.\)
5. The only \(Z\)-relevant functions occurring in \(w \in \Omega\) are \# , =, \leq , \circ \circ > , size, \min, \max, E_z, \phi, \neg \circ , \circ , \circ \circ \circ, \{ \circ \}_T.\)
6. All \(w \in \Omega\) are of the form

\[ U \land EQ \land EQ^\circ \land EQ^\# \land EQ^{\circ \circ} \land \]
\[ LEQ \land LEQ^\circ \land LEQ^\# \land LEQ^{\circ \circ} \]

where

- \(U\) contains no occurrence of \(=z\) or \(\leq z\)
- \(EQ\) is a conjunction of literals of the form \(z_1=z_2\)
- \(EQ^\circ\) is a conjunction of literals of the form \(\sim(z_1=z_2)\)
- \(EQ^\#\) is a conjunction of literals of the form \(\#(z_1=z_2)\)
- \(EQ^{\circ \circ}\) is a conjunction of literals of the form \(\sim \#(z_1=z_2)\)
- \(LEQ\) is a conjunction of literals of the form \(z_1 \leq z_2\)
- \(LEQ^\circ\) is a conjunction of literals of the form \(\sim(z_1 \leq z_2)\)
- \(LEQ^\#\) is a conjunction of literals of the form \(\#(z_1 \leq z_2)\)
- \(LEQ^{\circ \circ}\) is a conjunction of literals of the form \(\sim \#(z_1 \leq z_2)\)
7.4.2 The Algorithm

\( \omega \leftarrow \neg \neg \omega \); 

\( \text{do(} 
\quad \text{Productions for DNF} 
\text{)od;} \)

\( \Omega \leftarrow \{ \omega \}; \)

\( \text{do(} 
\quad (\omega_1 \lor \omega_2) \in \Omega \Rightarrow \Omega \leftarrow (\Omega \setminus \{\omega_1 \lor \omega_2\}) \cup \{\omega_1, \omega_2\} \)
\( \text{)od} \)

7.4.3 Proof of Correctness

The proof is essentially identical to that for Step 2, Chapter 6. None of Conjuncts 1-5 of the Invariant are disturbed by the conversion to DNF. Note that here, as in Chapter 6, the algorithm does more than necessary. Full DNF is not really required; all that is necessary to satisfy the Postcondition is to separate the literals containing \( = \) and \( \leq \), making them top-level conjuncts.
7.5 Step 5: Remove EQ#, EQ^#, LEQ^#, LEQ^-#

This step is all but identical to Step 3 of Chapter 6. Here we remove EQ#, EQ^#, LEQ^# and LEQ^-# and we do it in essentially the same way that we removed EQ#, EQ^-#, ORD# and ORD^-# in Chapter 6.

7.5.1 Specifications

Precondition: Same as Postcondition of the previous step

Invariant: Same as Precondition

Postcondition:

1. $\Omega \subseteq \text{BITZ}$, finite, unquantified.
2. $\exists w_0 \iff \not\in \Omega$.
3. All arguments to $\text{min}_z$ and $\text{max}_z$ are simple Z-variables.
4. The symbols $\text{min}_z$, $\text{max}_z$, $\circ < \circ$, and $\{ \circ \}$ do not occur nested in any $w \in \Omega$.
5. The only Z-relevant functions occurring in $w \in \Omega$ are $\#, =, \leq, \circ < \circ$, size, min, max, $E_z$, neg, +, -, $\circ$, $\{ \circ \}_T$, $\{ \circ \}_V$.
6. All $w \in \Omega$ are of the form
   
   $U \land \text{EQ} \land \text{EQ}^- \land \text{LEQ} \land \text{LEQ}^-$
   
   where
   
   a. $U$ contains no occurrence of $=z$ or $\leq z$
   
   b. EQ is a conjunction of literals of the form $z_1 = z_2$
   
   c. EQ^- is a conjunction of literals of the form $\neg (z_1 = z_2)$
   
   d. LEQ is a conjunction of literals of the form $z_1 \leq z_2$
   
   e. LEQ^- is a conjunction of literals of the form $\neg (z_1 \leq z_2)$.
7.5.2 The Algorithm

For each $u \in \Omega$ perform the following transformation:

\[ \text{do} \]

\( (a) \quad # (z_1 = z_2) \rightarrow # z_1 \land # z_2 \mid \)

\( (b) \quad # (z_1 \leq z_2) \rightarrow # z_1 \land # z_2 \)

\( \text{od} \)

7.5.3 Proof of Correctness

All conjuncts of the Invariant are trivially preserved except Conjunct 2, which is preserved because both productions are identity replacements.

The Postcondition is established by the Invariant and by the fact that the TRS could not halt with any conjuncts of $\text{EQ}^n$, $\text{EQ}^m$, $\text{LEQ}^n$ and $\text{LEQ}^m$ remaining.

Termination is trivial, since each production reduces the total number of occurrences of $=$ and $\leq$ in $\Omega$. 
7.6 Step 6: Remove EQ\(^{-}\) and LEQ\(^{-}\)

Here we remove negated occurrences of \(=_{z}\) and \(\leq_{z}\), leaving only the positive occurrences. This step is quite similar in substance to Step 4 of Chapter 6 in which we removed negated occurrences of \(=_{\nu}\) and ord.

7.6.1 Specifications

**Precondition:** Same as Postcondition for previous step.

**Postcondition:**

1. \(\Omega \subseteq \text{BITZ}, \text{finite, unquantified.}\)
2. \(\exists w_{0} \iff \neg \Omega.\)
3. All arguments to \(\min_{z}\) and \(\max_{z}\) are simple Z-variables.
4. The symbols \(\min_{z}, \max_{z}, ^{o}<^{o}\) and \(\{^{o}\}\) do not occur nested in any \(w \in \Omega.\)
5. The only Z-relevant function symbols occurring in \(w \in \Omega\) are \(-z, =z, \leq z, ^{o}<^{o}, \text{size, min, max, E}_{z}, \phi, \text{neg}, +, -, \mathbb{R}\) and \(\{^{o}\}_{T}.\)
6. All \(w \in \Omega\) are of the form

   \(U \land \text{EQ} \land \text{LEQ}\)

   where

   - \(U\) contains no occurrences of \(=_{z}\) or \(\leq_{z}\)
   - EQ is a conjunction of literals of the form \(z_{1}=z_{2}\)
   - LEQ is a conjunction of literals of the form \(z_{1}\leq z_{2}\)

7.6.2 The Algorithm

For each \(w \in \Omega\) perform the following transformation:

\[
\text{do(}
\begin{align*}
\text{(a)} & \quad \neg z_{1} = z_{2} \iff z_{1}<t'> \land z_{2}<t' \cup \\
\text{(b)} & \quad \neg z_{1} \leq z_{2} \iff z_{1}<t' \lor z_{2}<t'
\end{align*}
\text{)}
\]
then there is a variable $v_1$ and an assignment $\varphi$ such that

$$\text{elem}(v_1)'w \text{ and } \varphi' \text{ lb}(v_1) = \text{ub}(v_1)$$

Let us assume the hypothesis about the existence of $v_2$ and $\varphi'$ and prove the conclusion. Let $v_1$ be $v_2$ and let $\varphi$ be defined from $\varphi'$ as follows:

$$\varphi(x) = \varphi'(x) \text{ for all variables } x \text{ except } v$$

$$\varphi(v) = \gamma \text{ such that } \text{lb}(\gamma) = \varphi'(\text{lb}(v))$$
$$\text{ub}(\gamma) = \varphi'(\text{ub}(v))$$
$$\text{elem}(\gamma) = \varphi'(t')$$

We must first show that this is a well-defined definition. It is well-defined provided that $\text{ub}(\gamma) = \text{lb}(\gamma)$ so that $\text{elem}(\gamma)$ can be a non-error value. But this means that $V_\varphi(\text{lb}(v)) = V_{\varphi'}(\text{ub}(v))$ and we know by hypothesis that $\varphi' = w'$ and $w'$ contains (by definition) the conjunct $\text{lb}(v) = \text{ub}(v)$. Hence $\varphi$ is well-defined.

With $v_1$ and $\varphi$ defined we need to show three things:

1. $\text{elem}(v_1) = w$
2. $\varphi = w$
3. $\varphi \not\equiv \text{lb}(v_1) = \text{ub}(v_1)$.

By hypothesis $\text{elem}(v_2) = w'$. Hence $v_2$ cannot be $v$, since all occurrences of $\text{elem}(v)$ are removed from $w'$. Thus, $v_1$ is not $v$ either since $v_1$ and $v_2$ are the same. Thus $\text{elem}(v_1) = w$, because otherwise we could not have $\text{elem}(v_2) = w'$. This proves (1).

From the definition of $\varphi$ it is obvious that

$$V_\varphi(\text{elem}(v)) = V_{\varphi'}(t')$$
$$V_\varphi(\{v\}) = V_{\varphi'}(\{t'\})$$
$$V_\varphi(\text{lb}(v)) = V_{\varphi'}(\text{lb}(v))$$
$$V_\varphi(\text{ub}(v)) = V_{\varphi'}(\text{ub}(v))$$

Because all occurrences of $v$ in $w$ are in one of these four contexts, and because $\varphi$ and $\varphi'$ agree on all variables other than $v$, it is clear that $\varphi \equiv w$ as a consequence of $\varphi' \equiv w'$. This
Note that $t'$ must be a new variable with each iteration.

7.6.3 Proof of Weak Correctness

Each conjunct of the Invariant except Conjunct 2 is trivially invariant over each production.

To show the invariance of Conjunct 2, we will illustrate using production (a). The proof for production (b) is almost identical.

If production (a) is eligible to execute on some $w \in \Omega$, then, by Conjunct 5 of the Invariant, $w$ can be expressed as

$$w' \land \neg z_1 z_2$$

We now use two identities

$$=w' \land (\exists t'. z_1 \lt t'\gt \neq z_2 \lt t'\gt )$$

$$= \exists t'. (w' \land z_1 \lt t'\gt \neq z_2 \lt t'\gt )$$

where $t'$ does not occur free in $w'$ or $z_1$ or $z_2$, and conclude that

$$=w' \land \neg z_1 z_2 \iff =w' \land z_1 \lt t'\gt \neq z_2 \lt t'\gt .$$

Thus, the replacement of $\neg z_1 = z_2$ by $z_1 \lt t'\gt \neq z_2 \lt t'\gt$ (for a new variable $t'$) preserves the satisfiability/unsatisfiability of $\Omega$, and leaves Conjunct 2 invariant.

The Postcondition differs from the Invariant only in the requirement that the $\text{EQ}^{-}$ and $\text{LEQ}^{-}$ parts of the wffs in $\Omega$ be empty. It is obvious that if one of the parts were not empty, then one or the other of the two productions would apply, preventing termination.

7.6.4 Termination

The termination argument is trivial: each production reduces the total number of occurrences of $\lnot z$ and $\leq z$ in $\Omega$. 

)}
7.7 Step 7: Remove $#_z$

In this step we remove all occurrences of the $#_z$ function.

7.7.1 Specifications

Precondition: Same as the Postcondition of previous step.

Invariant: Same as the Precondition.

Postcondition:

1. $\Omega \subseteq \text{BITZ}$, finite, unquantified.
2. $\forall w \in \Omega \iff \neg \exists z$.
3. All arguments to $\text{min}_z$ and $\text{max}_z$ are simple $Z$-variables.
4. The symbols $\text{min}_z$, $\text{max}_z$, $\circ < o >$, and $\{ o \}$ do not occur nested in any $w \in \Omega$.
5. The only $Z$-relevant function symbols occurring in $w \in \Omega$ are $=_z$, $\leq_z$, $\circ < o >$, size, min, max, $E_z$, $\phi$, neg, $\oplus$, $\ominus$, and $\{ o \}$.
   [Note: $#_z$ has been removed.]
6. All $w \in \Omega$ are of the form
   
   $U \wedge \text{EQ} \wedge \text{LEQ}$
   
   where $U$, EQ and LEQ are as before.

7.7.2 The Algorithm

For all $w \in \Omega$ perform the following transformation:

\begin{verbatim}
\begin{enumerate}
  \item $\text{#zvar} \mapsto \text{true}$
  \item $\text{#E}_z \mapsto \text{false}$
  \item $\text{#}\phi \mapsto \text{true}$
  \item $\text{#}(\neg z) \mapsto \text{#z}$
  \item $\text{#}(z_1 + z_2) \mapsto \text{#z}_1 \land \text{#z}_2$
  \item $\text{#}(z_1 - z_2) \mapsto \text{#z}_1 \land \neg \text{#z}_2$
  \item $\text{#}(z_1 \cdot z_2) \mapsto \text{#z}_1 \land \text{#z}_2$
  \item $\text{#}\{ t \} \mapsto \text{#}\_t$
\end{enumerate}
\end{verbatim}
7.7.3 Proof of Weak Correctness

Conjuncts 1, 3, 4, 5 and 6 of the Invariant are trivially invariant. To show Conjunct 2 is invariant we need only note that each production represents an identity replacement.

The difference between the Invariant and the Postcondition is that the latter requires that there be no occurrences of the $z$ function. From Conjunct 4 of the Invariant we know that any $Z$-term must either be a $Z$-variable (production (a)) or be constructed with one of the function symbols $E_z$, $\phi$, $\text{neg}$, $+$, $-$, $\ast$ or $\{ o \}^T$ at its root (productions (b)-(h)). If $z$ occurs at all in a formula, it occurs with one of these eight kinds of $Z$-terms as its argument, and since each case is covered by some production (a)-(h), the PS cannot halt with any occurrence of $z$ remaining in $\Omega$.

7.7.4 Termination

The PS terminates on every $w \in \Omega$ because each production reduces the sum, over all occurrences of $z$ in $w$, of the number of other nodes below $z$ in the tree.
7.8 Step 8: Make Z-arguments to the Size and the \( z^\circ \) functions be Z-variables

In this step we will simplify the structure of terms based on the size function and the \( z^\circ \)-function. Afterwards all arguments to the size function and all Type-Z arguments to the \( z^\circ \) function will be simple Z-variables.

7.8.1 Specifications

**Precondition:**   
Same as Postcondition of previous step.

**Invariant:**   
Same as Precondition.

**Assumptions**

1. \( \Omega \subseteq \text{BITZ} \), finite, unquantified.
2. \( \not\exists w_0 \iff \not \Omega \).
3. All Z-arguments to \( \min_z, \max_z, \text{size} \) and \( z^\circ \) are Z-variables.
4. The symbols \( \min_z, \max_z, z^\circ \) and \( \{ z \} \) do not occur nested in any \( w \in \Omega \).
5. The only Z-relevant functions occurring in \( w \in \Omega \) are \( =, \leq, z^\circ, \text{size}, \min, \max, \text{E}_z, \phi, \text{neg}, +, -, \{ z \} \).
6. All \( w \in \Omega \) are of the form

\[ U \land \text{EQ} \land \text{LEQ} \]

where \( U, \text{EQ} \) and \( \text{LEQ} \) are as before.

7.8.2 The Algorithm

For each \( w \in \Omega \) perform the following transformation:

\[
\text{do(}
\begin{align*}
\text{size}(E_z) & : \rightarrow E_1 \\
\text{size}(\phi) & : \rightarrow \emptyset \\
\text{size}(-z) & : \rightarrow -\text{size}(z) \\
\text{size}(z_1 + z_2) & : \rightarrow \text{size}(z_1) + \text{size}(z_2) \\
\text{size}(z_1 - z_2) & : \rightarrow \text{size}(z_1) - \text{size}(z_2)
\end{align*}
\text{)}
\]
7.8.2

```
size(i \times z) :- i \times size(z)
size(\{ t \}) :- 1
E_t \rightarrow E_t
\phi \rightarrow \emptyset
(-z) \rightarrow -(z\langle t \rangle)
(z_1 + z_2) \rightarrow z_1 + z_2\langle t \rangle
(z_1 - z_2) \rightarrow z_1 - z_2\langle t \rangle
\langle i \times z \rangle \rightarrow i \times (z\langle t \rangle)
\{ t \} \rightarrow (t_1 = t_2 | 1 | \emptyset)
```

7.8.3 Proof of Weak Correctness

Conjuncts 1, 3, 4, 5 and 6 of the Invariant are trivially invariant. Conjunct 2 is maintained because each of the productions is an identity replacement.

Conjuncts 1, 2, 4, 5 and 6 of the Postcondition, and half of Conjunct 3 are part of the Invariant, so the only question arises with the requirement in the other half of Conjunct 3 that arguments to \( o \circ o \) and size be \( Z \)-variables. Since according to Conjunct 5 there are only seven function symbols left from which to build \( Z \)-terms, namely \( E_z, \phi, \text{neg}, +, -, \ast \) and \( \{ o \} \), and since one size-production and one \( o \circ o \)-production exist for each of those seven function symbols, the TRS cannot halt with occurrences of size or \( o \circ o \) unless the \( Z \)-arguments are simple \( Z \)-variables, (i.e. unless they do not contain any of the seven possible function symbols.)

7.8.4 Termination

The termination argument is simple: Each iteration reduces the sum, over all occurrences in \( \Omega \) of the size- or \( o \circ o \)-functions, of the number of nodes in the \( Z \)-arguments to the size- or \( o \circ o \)-function.
7.9 Step 9: Hypothesize all possible definedness and equality relations between TX-terms

This step is quite similar in spirit to Step 6 in Chapter 5 (and to Step 5 of Chapter 6).

In Step 6 of Chapter 5 we considered all terms that occurred as indices to a V-variable \( v \) in a wff \( w \) and called that set IX(\( w,v \)). We then divided the wff \( w \) into cases, one case corresponding to each distinct way the terms in IX(\( w,v \)) could be ordered by = and <. This gave each \( w\in\Omega \) a Y-part containing complete information about the equality and inequality of the terms in IX(\( w,v \)). In particular, the Y-part contained information determining whether two terms of the form \( v[i_1] \) and \( v[i_2] \) referred to the same element of vector \( v \) (because \( i_1=i_2 \)) or to different elements (because \( i_1\neq i_2 \)). Since the Y-part contained complete information about the equality or inequality of the index terms, there was no third case.

In Step 5 of Chapter 6 we did a similar thing. There we were concerned not with elements of vectors, but with subintervals of vectors, i.e. terms of the form \( v[i_1...i_2] \). We needed to know whether two such terms referred to the same interval, to overlapping intervals or to disjoint intervals, etc. Once again, we defined a set called IX(\( w,v \)) containing all of the I-terms involved in those notions, and divided each wff \( w \) into cases, one case for each distinct way that the terms in IX(\( w,v \)) could be configured according to the =, =, and < relations. Each \( w\in\Omega \) had a Y-part giving complete information about the definedness, equality in order of the endpoints of the V-terms occurring in \( w \).

Here in Chapter 7 will perform a third variation on this same theme. At first glance it might not seem reasonable, since in Chapters 5 and 6 the type of interest was a vector type, whereas here the type in question is Type-Z. Questions regarding whether or not two parts of a vector are the same or overlap or are disjoint might seem to have little to do with zsets.

Actually, however, there is a close analogy. Vectors basically represent maps from Type-I to Type-T. Conversely, zsets can be viewed as representing maps from Type-T to Type-I. So when we need to know something about the relations between I-terms to in the process
of removing V-terms from a language, it should come as no surprise that we have to have similar information about the relations between T-terms in order to remove Z-terms from the language.

In this step we define a set TX(w) analogous to the sets IX(w,v) defined in Chapters 5 and 6. It contains all T-terms which serve as "indices" into z-sets, either explicitly (in terms such as z<t>) or implicitly (in terms such as min(z), max(z) or {t}). We then consider all of the ways such terms can be configured with respect to #T, =T and ≠T, and divide the wff into cases in much the same way as was done in the two previous chapters. After this step each w∈Ω will have a Y-part containing complete information about the definedness and equality of terms in TX(w). All of this is made more explicit (and more formal) in the next subsections.

7.9.1 The sets TX(w), TCS(w) and TCF(w)

This subsection parallels Subsections 6.5.1 and 6.5.2 in which the notions IX(w,v), ICS(w,v) and ICF(w,v) were defined and their properties explored. We will define similar notions TX(w), TCS(w) and TCF(w), and draw by analogy on the theory presented earlier to shorten the presentation here.

In Chapter 6 we began by defining a simplification operator, ⁰, because the terms included in IX(w,v) were simplifications of terms occurring in w. Here that will not be necessary; we can include in TX(w) raw T-terms just as they occur in w (although a good simplification operator could improve the performance of the algorithm).

**Definition 1:** The set of T-terms TX(w) is defined as follows:

1. If min(z) ∈ w for any Z-variable z, then min(z)∈TX(w);
2. If max(z) ∈ w for any Z-variable z, then max(z)∈TX(w);
3. If z<t> ∈ w for any Z-variable z, then t∈TX(w);
4. If {t} ∈ w, then t∈TX(w);

No other terms are in TX(w). □
This definition is only intended to apply to wffs satisfying the Precondition to this step. For all such wffs the only kind of arguments permitted to the min and max functions, and the only Z-arguments permitted to the \( \circ \circ \)-function are simple Z-variables. Therefore, the phase "Z-variable" occurring in the definition is not really restrictive.

The set \( TX(w) \) includes all terms occurring in \( w \) that act as "indices" in any way to Z-variables, i.e. they are T-terms having some role as "members" of zsets. This is, of course, only a heuristic explanation for why this particular definition was selected; the final explanation is simply that the proofs we are constructing in this Chapter work with this definition.

There are several small points about the definition which deserve note. First, \( TX(w) \) is a function only of the wff \( w \); unlike \( IX(w,v) \), it does not take a second argument. \( TX(w) \) contains all of the terms that serve as indices to any Z-variable occurring in \( w \).

Second, we should notice that if \( \text{min} \), \( \text{max} \), \( \circ \circ \) and \( \{0\} \) do not occur in \( w \), then \( TX(w) \) is empty.

Third, we need to point out that no term \( t_1 \in TX(w) \) is a proper subterm of another term \( t_2 \in TX(w) \). This is a direct consequence of Conjunct 5 of the Precondition to this step.

We are now ready to define the set \( TCS(w) \). \( TCS(w) \) is a set of sets of wffs over the elements of \( TX(w) \), each set of wffs in \( TCS(w) \) corresponding to a distinct way that the terms of \( TX(w) \) can behave under \( \equiv_T \) and \( =_T \).

**Definition 2**: A T-index Constraint Set (TCS) for a wff \( w \) (satisfying the Precondition for this step) is a set of formulae over the terms in \( TX(w) \) with the following properties. (The set will be called \( Y \).)

1. All elements of \( Y \) are of the form \( \neg t_1 = t_2 \) or \( t_1 \neq t_2 \) for some \( t_1, t_2 \in TX(w) \).
2. If \( \neg t_1 \not\in Y \) and \( \neg t_2 \not\in Y \) then exactly one of \( t_1 = t_2 \in Y \) or \( t_1 \neq t_2 \in Y \) holds.
3. If \( t_1 = t_2 \in Y \) or \( t_1 \neq t_2 \in Y \) then \( \neg t_1 \not\in Y \) and \( \neg t_2 \not\in Y \).
4. If \( \neg t \not\in Y \) for \( t \in TX(w) \), then \( t \equiv_T Y \).
5. If \( t_1 = t_2 \in \mathcal{E} \) then \( t_2 = t_1 \in \mathcal{E} \).

6. If \( t_1 = t_2 \in \mathcal{E} \) and \( t_2 = t_3 \in \mathcal{E} \) then \( t_1 = t_3 \in \mathcal{E} \). □

The motivation for this definition is similar to that for the definition of \( \text{ICS}(w,v) \), namely if all of the elements of some TCS \( Y \) are \textit{true} under some valuation \( V_\varphi \), then we can compute very simply the truth value of any (three-valued) propositional combination of the elements of \( Y \) under the same valuation \( V_\varphi \); furthermore, the values of the terms in \( \text{TX}(w) \) satisfy the constraints in at least one TCS.

Notice that if \( \text{TX}(w) \) is empty then there is only one TCS, namely the empty set of wffs.

**Definition 3:** Let \( \text{TCS}(w) \) be the set of all TCS's for the wff \( w \). □

This definition, like the previous ones, applies only to wffs \( w \) satisfying the Precondition for this Step.

We now prove some lemmas and theorems about \( \text{TX}(w) \) and \( \text{TCS}(w) \) which are similar to those proved about \( \text{IX}(w,v) \) and \( \text{ICS}(w,v) \) in Subsection 6.5.2.

**Lemma 4:** \( \text{TX}(w) \) is a finite set.

**Proof:** Trivial. Wff \( w \) is of finite length, and every term \( t \in \text{TX}(w) \) is a subterm of \( w \). □

**Lemma 5:** Any \( \mathcal{E} \in \text{TCS}(w) \) is finite.

**Proof:** Trivial. \( \text{TX}(w) \) is finite, and \( \mathcal{E} \) is a set of terms of the form \(-w_1, t_1 = t_2 \) or \( t_1 \neq t_2 \) where \( t, t_1, t_2 \in \text{TX}(w) \). There can only be a finite number of such terms. □

**Lemma 6:** The members of \( \mathcal{E} \), where \( \mathcal{E} \in \text{TCS}(w) \), need not be simultaneously satisfiable.

**Proof:** Consider two distinct T-terms which are equivalent, e.g. \( t \) and \( \min(t,t) \) for some T-variable \( t \). Let \( w \) be a wff in which \( \text{TX}(w) = \{ t, \min(t,t) \} \). For example, \( w \) could be the wff \( z < t > = z < \min(t,t) > \). Consider the following sets \( Y_1 \) and \( Y_2 \):
\[ Y_1 = \{ \neg \#t, \neg \#\min(t,t) \} \]
\[ Y_2 = \{ t = t, \min(t,t) = \min(t,t), t \neq \min(t,t) \} \]

\( Y_1 \) is not satisfiable because \( \neg \#t \) is unsatisfiable (variable \( t \) can only be assigned a defined value). \( Y_2 \) is not satisfiable because \( t \neq \min(t,t) \) unsatisfiable.

**Definition 7:** Let TCU(w), the T-index Constraint Universe for wff \( w \), be the set of all terms of the form \( \neg \#t, t_1 = t_2 \) or \( t_1 \neq t_2 \) for \( t, t_1, t_2 \in \text{ETX}(w) \).

**Lemma 8:** If \( Y \in \text{TCS}(w) \) then \( Y \in \text{TCU}(w) \).

**Proof:** Immediate from the definitions of TCS(\( w \)) and TCU(\( w \)).

**Definition 9:** For any assignment \( \varphi \), let \( Y_\varphi(w) \) the set of all terms \( a \in \text{TCU}(w) \) such that \( V_\varphi(a) = \text{true} \).

**Theorem 10:** For any assignment \( \varphi \) and wff \( w \) the set \( Y_\varphi(w) \) is a TCS.

**Proof:** We examine one by one the clauses in the definition of a TCS to see if they are satisfied by \( Y_\varphi(w) \) (which we abbreviate to \( Y \)).

1. The first clause holds by definition of \( Y \) and TCU(\( w \)).
2. If \( \neg \#t_1 \) and \( \neg \#t_2 \) are not in \( Y \), then it must be because they are not true under \( V_\varphi \). Hence, \( \#t_1 \) and \( \#t_2 \) are true under \( V_\varphi \) or \( t_1 \neq t_2 \) is true under \( V_\varphi \). Whichever one is true must be a member of \( Y \).
3. If \( t_1 = t_2 \in Y \) or \( t_1 \neq t_2 \in Y \) then one of them must have the value true under \( V_\varphi \). This can only happen if \( \neg \#t_1 \) and \( \neg \#t_2 \) have the value false under \( V_\varphi \), and are thus not members of \( Y \).
4. If \( \neg \#t \notin Y \) then \( \#t \) must be false, and hence \( \#t \) is true under \( V_\varphi \). But \( \#t \neq t \), so \( t = t \) is also true under \( V_\varphi \).
5. If \( t_1 = t_2 \) is true under \( V_\varphi \), then \( t_2 = t_1 \) is true under \( V_\varphi \).
6. If \( t_1 = t_2 \) and \( t_2 = t_3 \) are both true under \( V_\varphi \), then \( t_1 = t_3 \) is also true.

Hence, \( Y \) satisfies all clauses of the definition of a TCS.
We now continue the development of this theory toward the Complete Information Theorem similar to the theorem of the same name in Subsection 6.5.2.

Definition 11: Let \( A(w) \) be the set of all Boolean terms (atomic formulae) constructable from the T-terms in \( TX(w) \) and the symbols \( \#_1 \) and \( =_1 \). \( \square \)

Definition 12: Let \( L_{TX}(w) \) be the language of all unquantified formulae constructable from the atoms in \( A(w) \) and the three-valued propositional connectives. \( \square \)

Theorem 13: Complete Information Theorem: A TCS for a wff \( w \) whose elements are simultaneously satisfiable contains complete information about the truth values of all wffs in \( L_{IX}(w) \). In other words, if \( \varphi \) and \( \psi \) are two assignments each satisfying all elements of a TCS \( Y(w) \), then \( V_{\varphi} \) and \( V_{\psi} \) assign the same truth values to all formulae in \( L_{IX}(w) \).

Proof: We show that \( \varphi \) and \( \psi \) must agree on all of the atoms of \( L_{IX}(w) \). The truth values of all wffs in \( w \) can be computed by truth table under either assignment, thereby assuring that \( V_{\varphi} \) and \( V_{\psi} \) agree.

Suppose \( a \in A(w) \), i.e., an atom in \( L_{TX}(w) \). Then a must be of the form \( \#t \) or \( t_1 = t_2 \) for some \( t, t_1, t_2 \in TX(w) \). We will show in either case that \( V_{\varphi} \) and \( V_{\psi} \) agree on the value they assign to \( a \).

Case 1: \( a \) is \( \#t \). Either \( \neg \#t \in Y \) or \( \neg \#t \notin Y \). If \( \neg \#t \in Y \) then, because \( \varphi \) and \( \psi \) both satisfy all elements of \( Y \), \( V_{\varphi} \) and \( V_{\psi} \) must both assign true to \( \#t \), and hence assign false to \( \#t \), thus agreeing. If, on the other hand, \( \neg \#t \notin Y \), then by Clause 2 of the definition of a TCS, either \( t=t \in Y \) or \( t \notin Y \). Without loss of generality we can assume that \( t=t \in Y \). Then \( V_{\varphi} \) and \( V_{\psi} \) must assign true to \( t=t \), and this is only possible if they both assign true to \( \#t \). Therefore, \( V_{\varphi} \) and \( V_{\psi} \) agree in this case also.

Case 2: \( a \) is \( t_1 = t_2 \). If \( t_1 = t_2 \in Y \) then \( V_{\varphi} \) and \( V_{\psi} \) both assign true to it, and thus agree. If \( t_1 = t_2 \notin Y \) then there are two subcases to consider according to Clause 2 of the definition of a TCS: either \( \neg t_1 \) and/or \( \neg t_2 \) are in \( Y \), or else \( \neg t_1 \notin Y \) and \( \neg t_2 \notin Y \) but \( t_1 \neq t_2 \in Y \).

Case 2a: Suppose, without loss of generality that \( \neg t_1 \in Y \). Then \( V_{\varphi} \) and \( V_{\psi} \) both assign true to \( \neg t_1 \), and hence must agree in assigning error to \( t_1 = t_2 \).

Case 2b: Suppose \( \neg t_1 \notin Y \) and \( \neg t_2 \notin Y \) and \( t_1 \neq t_2 \in Y \). Since \( t_1 \neq t_2 \in Y \), \( V_{\varphi} \) and \( V_{\psi} \) both assign \( t_1 \neq t_2 \) the value true; hence they must agree that the value of \( t_1 = t_2 \) is false. \( \square \)
We can give information about the size of TCS(w) to help with an analysis of the performance of the algorithm.

**Lemma 14:** For each w, TCS(w) is finite.

**Proof:** Trivial; the number of sets satisfying Clause 1 alone is finite, and the other clauses only reduce the number. □

**Lemma 15:** Let n be the size of TX(w). Then the size of TCS(w) is ψ(n) where

\[ ψ(n) = \sum_{0 \leq k \leq \binom{n}{m}} \binom{n}{m} \binom{m}{k} \]

where \( \binom{n}{m} \) is a binomial coefficient and \( \binom{m}{k} \) is a Stirling number of the second kind.

**Proof sketch:** We view the members Y of TCS(w) as being formed by a sequential process wherein we first select a subset of TX(w) to be undefined, i.e. to have \(-\#t \in Y\), and then select an equivalence relation on the remaining elements to determine which atoms of the form \( t_1^m \) are to be included in Y.

The number of subsets of TX(w) having size exactly m is \( \binom{n}{m} \), and the number of ways to divide the remaining terms into distinct equivalence classes is

\[ \sum_{0 \leq k \leq n-m} \binom{n-m}{k} \]

This leads to the following argument.

\[ ψ(n) = \sum_{0 \leq m \leq n} \left[ \binom{n}{m} \sum_{0 \leq k \leq n-m} \binom{n-m}{k} \right] \]
\[ = \sum_{0 \leq m \leq n} \left[ \binom{n}{m} \sum_{0 \leq k \leq m} \binom{m}{k} \right] \]
\[ = \sum_{0 \leq m \leq n} \binom{n}{m} \sum_{0 \leq k \leq m} \binom{m}{k} \]
\[ = \sum_{0 \leq k \leq m} \binom{n}{k} \binom{m}{k} \] □

As we did in Chapter 6, we will introduce the symbol \( \vdash \) to be a binary relation between wff sets \( Y \in \text{TCS}(w) \) and wffs \( u \in \text{L}_T \). We will abbreviate the development of theory of the \( \vdash \) relation because it so closely parallels that given in Chapter 6.

**Definition 16:** Let \( Y \in \text{TCS}(w) \) and \( u \in \text{L}_T \). Define \( Y \vdash u \) to mean "for all assignments \( \varphi \), if \( \varphi \) satisfies all elements of \( Y \), then \( \varphi \) satisfies \( u \)." □
Theorem 17: There is an algorithm for deciding whether $Y\vdash u$, i.e., $\vdash u$ is a computable relation.

Proof: The algorithm is implicit in the proof of the Complete Information Theorem, 13. \[\square\]

Definition 18: A T-index Constraint formula for wff $w$ is a conjunction (using $\land$, not $\land_\wedge$) of all of the elements of some $Y\epsilon TCS(w)$. If a TCS is empty, then the corresponding TCF is the wff true. \[\square\]

Definition 19: Let $TCF(w)$ be the set of all distinct (up to commutativity and associativity of $\land$) TCFs for wff $w$. \[\square\]

Definition 20: For $Y\epsilon TCF(w)$ and $u\epsilon L_TX(w)$ define $Y\vdash u$ to mean that for the $Y\epsilon TCS(w)$ which is the set of conjuncts of $y$, $Y\vdash u$. [Note: we are just extending the use of the $\vdash$ notation.] \[\square\]

The following Lemma and Theorem were proved in Subsection 6.5.2 for $IX(w,v)$, $ICS(w,v)$, $ICF(w,v)$, $L_{IX}(w,v)$ and for the notion of $\vdash$ defined there. We will restate the theorems here with respect to $TX(w)$, $ICS(w)$, $TCF(w)$, $L_{TX}(w)$ and our current definition of $\vdash$. The proofs of these Lemmas and Theorems follows closely those given in Chapter 6, so we will not repeat them.

Theorem 21: Let $TCF(w)$ be a collection $\{y_1, \ldots, y_n\}$ of TCFs. Then $\vdash y_{1\leq i \leq n}(y_i)$. 

Proof: Omitted. \[\square\]

Lemma 22: For $y\epsilon TCF(w)$ and $u_1, u_2\epsilon L_{TX}(w)$,

$Y\vdash u_1 \land u_2 \iff Y\vdash u_1$ and $Y\vdash u_2$

$Y\vdash u_1 \lor u_2 \iff Y\vdash u_1$ or $Y\vdash u_2$
Lemma 23: If $y \in \text{TCF}(w)$ then $\text{TX}(y \wedge w) = \text{TX}(w)$.

Proof: Since by Conjunct 5 of the Precondition the functions $\min$, $\max$, $\circ \circ \circ$ and $\{\circ\}$ do not occur nested, none of the elements of $\text{TX}(w)$ contain any occurrences of $\min$, $\max$, $\circ \circ \circ$ or $\{\circ\}$. Therefore $\text{TX}(y)$ is empty. And since $\text{TX}(y \wedge w) = \text{TX}(y) \cup \text{TX}(w)$ (a simple consequence of the definition of $\text{TX}$) we conclude that $\text{TX}(y \wedge w) = \text{TX}(w)$. □

Lemma 24: If $y \in \text{ICF}(w)$ then

\begin{align*}
\text{TCS}(y \wedge w) &= \text{TCS}(w) \\
\text{TCF}(y \wedge w) &= \text{TCF}(w)
\end{align*}

Proof: Immediate from the previous lemma. □

7.9.2 Summary of Step 9

Informally, what happens in Step 9 is this. Each wff $w \in \Omega$ is replaced by a finite set of wffs $\{w_1 \ldots w_n\}$ having two properties.

1. Mutual unsatisfiability: $\not \models w \iff \not \models \{w_1 \ldots w_n\}$ (where $\not \models \{w_1 \ldots w_n\}$ means $\not \models w_1$ and $\not \models w_2$ and ... $\not \models w_n$).

2. Each $w_i$ has a $Y$-part with complete information about definedness and equality among the terms of $\text{TX}(w)$.

We do this essentially the way we did in Chapter 6. For a wff $w \in \Omega$ we first construct the set $\text{TX}(w)$, then the set $\text{TCF}(w) = \{y_1 \ldots y_n\}$. We then form

\begin{align*}
w_1 &= y_1 \wedge w \\
w_2 &= y_2 \wedge w \\
&\ldots \\
w_n &= y_n \wedge w.
\end{align*}

The set $\{w_1 \ldots w_n\}$ now has the required properties.

1. The mutual unsatisfiability property holds as a consequence of Theorem 21. We can demonstrate this as follows:
\[ F y_1 \lor \ldots \lor y_n \] ! Theorem 21
\[ F w = (y_1 \lor \ldots \lor y_n) \land w \] ! Prop. Calc.
\[ F w = (y_1 \land w) \lor \ldots \lor (y_n \land w) \] ! Distributivity
\[ F w = w_1 \lor \ldots \lor w_n \] ! Definition
\[ F w \iff F(w_1 \lor \ldots \lor w_n) \] ! Metatheorem
\[ \forall w \iff \forall w_1 \text{ and } \ldots \text{ and } \forall w_n \] ! Metatheorem
\[ \exists w \iff \exists \{w_1, \ldots, w_n\} \] ! Def of \( \exists \{w_1, \ldots, w_n\} \)

2. Each \( w \), i.e. \( y_i \land w \), obviously has a \( Y \)-part with complete information about \( TX(w) \). However, we need to show that it has complete information about \( TX(y_i \land w) \). This follows from Lemma 23, since \( TX(y_i \land w) \) is the same as \( TX(w) \).

We can now proceed to presenting Step 9 and its correctness proof.

### 7.9.3 Specifications

**Precondition:**
Same as Postcondition for Step 8.

**Postcondition:**

1. \( \Omega \subseteq \text{BiTZV} \), unquantified, finite.
2. \( Fw_0 \iff \neg \Omega \).
3. All \( Z \)-arguments to \( \text{min}, \text{max}, \text{size and } ^{o<o} \) are simple \( Z \)-variables.
4. The symbols \( \text{min}, \text{max}, ^{o<o} \) and \( \{^o\} \) do not occur nested in any \( w \in \Omega \).
5. The only \( Z \)-relevant symbols occurring in any \( w \in \Omega \) are \( \Rightarrow, \leq, ^{o<o}, \text{size, min, max, } E_Z, \phi, \neg, +, -, \{^o\} \).
6. All \( w \in \Omega \) are of the form
   \[ y \land U \land EQ \land LEQ \]
   where
a. EQ is a conjunction of atoms of the form $z_1 = z_2$;
b. LEQ is a conjunction of atoms of the form $z_1 \leq z_2$;
c. $U$ contains no occurrences of $=$ or $\leq$;
d. $Y \in TCF(w)$ [Note: $TCF(w) = TCF(Y \land U \land EQ \land LEQ) = TCF(U \land EQ \land LEQ)$]

7.9.4 The Algorithm

$$\Lambda := \Omega;$$

For each $w \in \Omega$
do

$$\Lambda := (\Lambda - \{w\}) \cup \{y \land w \mid y \in TCF(w)\};$$
$$\Omega := \Lambda$$

7.9.5 Weak Correctness

This proof follows closely the discussion in Subsections 7.9.1 - 7.9.2. We will establish each of the conjuncts of the Postcondition in turn.

First, $\Omega \subseteq \text{BITZ}$ because each wff is of the form $y \land w$ where $w$ was in $\Omega$ before execution and $y \in TCF(w)$ contains no operators outside BITZ. $\Omega$ remains unquantified obviously, since no quantifiers are introduced. And $\Omega$ is still finite because it is formed by substituting a finite set of wffs for each of the finite number of wffs that were in $\Omega$ before this step.

The second conjunct is established by the following argument. We showed in Subsection 7.9.1 that

$$\not\vdash w \iff \not\vdash \{y \land w \mid y \in TCF(w)\}.$$ 

Hence, the following statement is invariant over the loop in the algorithm:

$$\not\vdash \Omega \land \not\vdash U \iff \not\vdash \Lambda$$

Since it is invariant, it holds at the completion of the loop, and we can establish Conjunct 2 if we establish the following Hoare statement:

$$\not\vdash \Omega \land \not\vdash U \iff \not\vdash \Lambda \{\Omega := \Lambda\} \not\vdash \Omega \iff \not\vdash \Omega.$$
This, however, is a trivial consequence of the Hoare assignment axiom and of the transitivity of $\iff$.

Conjuncts 3, 4 and 5 are obviously true at termination, since they are clearly invariant over the course of the loop (when viewed as properties of $\Lambda$ rather than of $\Omega$).

Conjunct 6 is a consequence of the fact that each $w\in\Omega$ was already of the form $U \land EQ \land LEQ$

before Step 9, and that the substance of the loop is simply to tack $Y$-parts onto copies of the elements of $\Omega$. See the discussion in Subsection 7.9.2.

7.9.6 Termination

Because $\Omega$ is finite the loop only executes a finite number of iterations. Each iteration is also finite since $TCF(w)$ is always a finite set.
7.10 Step 10: Remove those terms in TX(w) which are undefined

The Y-part of a wff w∈Ω may specify, by the inclusion of the conjunct ¬w1, that a term t∈TX(w) is undefined. In this step we rewrite the formulae in Ω so that all occurrences of such undefined TX-terms are removed. Each wff w is transformed into an equivalent wff w' in such a way that TX(w') ⊆ TX(w). The terms in TX(w)−TX(w') are exactly the terms which the Y-part of w specified to be undefined. The Y-part of w' specifies that all terms in TX(w') are defined.

7.10.1 Specifications

Precondition: Same as the Postcondition for Step 9.
Invariant: Same as the Precondition.
Postcondition:

1. Ω≤BITZ, unquantified, finite.
2. ∃w₀ ↔ ∅ Ω.
3. All Z-arguments to min, max, size and oo are simple Z-variables.
4. The symbols min, max, oo and {o} do not occur nested in any w∈Ω.
5. The only Z-relevant function symbols occurring in any w∈Ω are =z₁≤z₂ oo, size, min, max, E₂, φ, neg, +, −, *, {o}.
6. For all w∈Ω the top level conjuncts of w can be rearranged to be of the form
   \[ Y \land U \land EQ \land LEQ \]

where

a. EQ is a conjunction of atoms of the form z₁=z₂;
b. LEQ is a conjunction of atoms of the form z₁≤z₂;
c. U contains no occurrences of =z or ≤z;
d. \( Y \in TCF(w) \). [Note: \( TCF(w)=TCF(Y \land U \land EQ \land LEQ)=TCF(U \land EQ \land LEQ) \)]
7.10.1

7. For all $t \in T_X(w)$: $Y_w \vdash t$.

7.10.2 The Algorithm

For each $w \in \Omega$ perform the following transformation:

do(
(a) $Y_w \vdash \# \min(z) \lor \neg \# \max(z) \Rightarrow$

$$u \leftarrow \text{REPLALL}(u, \min(z):=-E_T, \max(z):=-E_T) \land z=\phi$$

(b) $Y_w \vdash \# t$ and $z < t \Rightarrow$

$$u \leftarrow \text{REPLALL}(u, z < t):=-E_1$$

c) $Y_w \vdash \# t$ and $\{ t \} u \Rightarrow$

$$u \leftarrow \text{REPLALL}(u, \{ t \}:=-E_2)$$

)od

7.10.3 Proof of Weak Correctness

The Invariant holds at the start of the PS because it is the same as the Postcondition of the previous step.

Conjuncts 1, 3, 4 and 5 of the Invariant are clearly invariant over each of the three productions. We therefore concentrate on proving the invariance of Conjuncts 2 and 6.

To prove the invariance of Conjunct 2 we show that each production replaces $w$ by an equivalent formula. We must consider each production in turn.

For production (a), let $w'$ be

$$\text{REPLALL}(w, \min(z):=-E_T, \max(z):=-E_T) \land z=\phi.$$ 

We must show that whenever the pattern for the production is satisfiable, then $w$ is equivalent to $w'$, i.e.
\[(Y_w \vdash \neg \min(z) \lor \neg \max(z)) \Rightarrow \forall w \equiv w'.\]

If the hypothesis of this implication holds, then it must be because \(Y_w\) contains as a conjunct either \(\neg \min(z)\), \(\neg \max(z)\) or both. Therefore, we can write \(w\) in one of the following three forms:

\[\neg \min(z) \land A\left(\min(z), \max(z)\right)\]
\[\neg \max(z) \land A\left(\min(z), \max(z)\right)\]
\[\neg \min(z) \land \neg \max(z) \land A\left(\min(z), \max(z)\right)\]

where \(A\left(\min(z), \max(z)\right)\) stands for the "rest" of \(w\), with possible occurrences of \(\min(z)\) and \(\max(z)\) highlighted. Without loss of generality we can consider just the first case, since the other two are just like it.

The first thing to notice is that \(z\) is a simple variable; this is because of Conjunct 3 of the invariant. Therefore we can assert that

\[\exists w \equiv \neg \min(z) \equiv (z=0).\]

(If \(z\) were not known to be a variable we would have to write \(\neg \min(z) \equiv (\neg z \lor z=\phi).\)) We can therefore conclude that

\[\exists w \equiv \neg \min(z) \land A\left(\min(z), \max(z)\right) \land z=\phi.\]

And since \(\exists z=\phi \Rightarrow \min(z)=E_T\) and \(\exists z=\phi \Rightarrow \max(z)=E_T\) we can reduce this further to

\[\exists w \equiv \neg E_T \land A(E_T, E_T) \land z=\phi.\]

which is exactly the same as

\[\exists w \equiv w'\]

as required. Hence, Conjunct 2 is invariant over production (a).

Productions (b) and (c) are similar to one another, so we will prove the Invariance of Conjunct 2 only over production (b). We need to show that

\[Y_w \vdash \neg \alpha t \text{ and } z<_t \alpha w \Rightarrow \exists w \equiv \text{REPLACE}(w, z<_t \neg \alpha E_T).\]

This is simple, because if \(Y_w \vdash \neg \alpha t\) then \(w\) can be written as
\[\neg\#t \land A(z^{<t>})\]

where \(A(z^{<t>})\) is the rest of the wff with the occurrences of the term \(z^{<t>}\) emphasized. Since

\[\text{E-\neg\#t} \supset (z^{<t>}:E_{\neg})\]

we can conclude by propositional manipulation that

\[\text{Ew} = (\neg\#t \land (z^{<t>}:E_{\neg}) \land A(z^{<t>})).\]

By equivalence substitution we get

\[\text{Ew} = (\neg\#t \land (z^{<t>}:E_{\neg})) \land A(E_{\neg})\]

and by propositional manipulation again because \(\text{E-\neg\#t} \supset (z^{<t>}:E_{\neg})\) we get

\[\text{Ew} = (\neg\#t \land A(E_{\neg})).\]

We are now finished, because

\[\neg\#t \land A(E_{\neg})\]

is REPLALL \((w, z^{<t>}:E_{\neg})\) is \(w'\)

so we have proved

\[\text{Ew} = w'\]

as required. This completes the proof of invariance for Conjunct 2.

We now proceed to the proof of invariance of Conjunct 6. Once again, since production (b) is so similar to production (c), we will omit consideration of production (c).

In cases where production (a) applies we can view \(w\) as being of one of the forms

\[\neg\#\text{min}(z) \land Y^* \land U(\text{min}(z)) \land \text{EQ} \land \text{LEQ},\]

\[\neg\#\text{max}(z) \land Y^* \land U(\text{max}(z)) \land \text{EQ} \land \text{LEQ},\]

\[\neg\#\text{min}(z) \land \neg\#\text{max}(z) \land Y^* \land U(\text{min}(z)) \land \text{EQ} \land \text{LEQ},\]

depending on whether \(\text{min}(z)\) or \(\text{max}(z)\) or both occur in \(w\). Here we denote by \(Y^*\) all of the \(Y\)-part of \(w\) except the conjuncts involving \(\text{min}(z)\) and \(\text{max}(z)\). We also use \(U(\text{min}(z))\), \(U(\text{max}(z))\) and \(U(\text{min}(z), \text{max}(z))\) to emphasize occurrences of \(\text{min}(z)\) and \(\text{max}(z)\) in the \(U\)-part of \(w\). (There can be no occurrences of \(\text{min}\) or \(\text{max}\) in \(\text{EQ}\) or \(\text{LEQ}\).)
After applying production (a) w is of one of the following forms:

\[ \neg \#E_T \land Y^* \land U(E_T) \land EQ \land LEQ \land z=\phi; \]
\[ \neg \#E_T \land Y^* \land U(E_T) \land EQ \land LEQ \land z=\phi; \]
\[ \neg \#E_T \land \neg \#E_T \land Y^* \land U(E_T, E_T) \land EQ \land LEQ \land z=\phi. \]

Since the first two are identical (as forms) we will consider them to be one case. The two remaining cases can be rearranged as

\[ Y^* \land (U(E_T) \land \neg \#E_T) \land (EQ \land z=\phi) \land LEQ; \]
\[ Y^* \land (U(E_T, E_T) \land \neg \#E_T \land \neg \#E_T) \land (EQ \land z=\phi) \land LEQ. \]

Now our goal, of course, is to show that Conjunct 6 is invariant over production (a), so we must show that \( w' \), which must be in one of these two forms, matches the form

\[ Y' \land U' \land EQ' \land LEQ' \]

where \( Y', U', EQ' \) and \( LEQ' \) satisfy the appropriate restrictions cited in Conjunct 6. It is easy to see that if we view

\[ Y' \text{ as } Y^*, \]
\[ U' \text{ as } (U(E_T) \land \neg \#E_T) \text{ or as } (U(E_T, E_T) \land \neg \#E_T \land \neg \#E_T), \]
\[ EQ' \text{ as } EQ \land z=\phi, \] and
\[ LEQ' \text{ as } LEQ \]

then \( w' \) is in fact of the appropriate form. [Note: At any time one may simplify \( \neg \#E_T \) to \( \text{true} \), but that fact is irrelevant to the point of this proof.] This completes the proof of invariance of Conjunct 6 over production (a).

The proof that Conjunct 6 is invariant also over production (b) (and, \textit{mutatis mutandis}, production (c)) is similar to the one just given for production (a). If \( Y_{w} \vdash \#t \), then \( w \) can be written as

\[ \neg \#t \land Y^* \land U(z<\!t>) \land EQ \land LEQ. \]

Once again, we let \( Y^* \) denote all of the \( Y \)-part of \( w \) except the conjunct \( \neg \#t \), and we let
$U(z^{<t>})$ denote the U-part of $w$ with any occurrences of the term $z^{<t>}$ emphasized. ($z^{<t>}$ cannot occur in EQ or LEQ.)

After applying production (b), $w$ is transformed into

$$\neg \#t \land Y^* \land U(E_T) \land EQ \land LEQ.$$ 

Now, unlike the previous proof, there are two cases to consider here:

1. $TX(w')=TX(w)-\{t\}$, i.e. in the course of removing all occurrences of the term $z^{<t>}$ we have removed all occurrences of the term $t$ which would qualify it for membership in $TX(w')$.

2. $TX(w')=TX(w)$, i.e. even though all occurrences of the term $z^{<t>}$ have been removed during the construction of $w'$, there may be other occurrences of the term $t$, such as in the context $\{t\}$ or in the context $z_2^{<t>}$ where $z_2$ is different from $z$, which qualify the term $t$ for membership in $TX(t)$.

In order to show that $w'$ fits the pattern

$$Y' \land U' \land EQ' \land LEQ'$$

we treat the two cases separately.

1. In the first case we view

   $Y'$ as $Y^*$,

   $U'$ as $\neg \#t \land U(E_T)$,

   $EQ'$ as $EQ$, and

   $LEQ'$ as $LEQ$.

   This partitioning into $Y'$, $U'$, $EQ'$ and $LEQ'$ parts satisfies the requirements of Conjunct 6.

2. In the second case we view

   $Y'$ as $\neg \#t \land Y^*$,

   $U'$ as $U(E_T)$,

   $EQ'$ as $EQ$, and

   $LEQ'$ as $LEQ$.

   This partitioning of $w'$ also satisfies the requirements for Conjunct 6.
Hence, in either case, w', the result of applying production (b) to w, satisfies Conjunct 6. This completes the proof of invariance for Conjunct 6, and also for the entire Invariant.

The proof that the Postcondition holds at termination boils down to a proof that Conjunct 7 holds, since Conjuncts 1-6 of the Postcondition are also conjuncts of the Invariant. Conjunct 7, in turn, is a consequence of the fact that at termination none of the production patterns are satisfiable.

Suppose Conjunct 7 were false at termination, and there exists a term t such that t\in TX(w) but \( Y_w \neq t \) at termination. Then, by the fact that \( Y_w \) contains complete information about the definedness of terms in TX(w), we deduce that \( Y_t = t \).

Now, since t\in TX(w) it is either of the form min(z) or max(z) (for some variable z) or it occurs in w in the context z<\langle t \rangle (for some variable z) or in the context \{ t \}. This is simply the definition of TX(w).

But if t is min(z) or max(z) then \( Y_t \neq \text{min}(z) \lor \text{max}(z) \) and hence production (a) applies, contradicting the assumption that the PS terminated.

Similarly, if t occurs in w in the context z<\langle t \rangle or \{ t \} then production (b) or (c) applies, once again contradicting the hypothesis that the PS terminated.

Hence, our supposition that Conjunct 7 is false at termination must be erroneous, and Conjunct 7 is in fact established at termination.

7.10.4 Termination

Each iteration reduces the total number of occurrences of terms t such that t\in TX(w). [Notice that a single iteration of productions (b) or (c) may not reduce the size of TX(w), but they do reduce the number of occurrences of members of TX(w).]
7.11 Step 11: Remove $E_z$

In this step we remove all occurrences of $E_z$ from each $w \in \Omega$. $E_z$ will not be reintroduced in any subsequent steps.

7.11.1 Specifications

**Precondition:** Same as the Postcondition of Step 10.

**Invariant:** Same as the Precondition.

**Postcondition:**

1. $\Omega \subseteq \text{BITZ}$, unquantified, finite.
2. $\mathcal{F}_w \iff \forall \Omega$.
3. All $Z$-arguments to $\text{min}$, $\text{max}$, $\text{size}$ and $\circ \circ \circ$ are simple $Z$-variables.
4. The symbols $\text{min}$, $\text{max}$, $\circ \circ \circ$ and $\{ \circ \}$ do not occur nested in any $w \in \Omega$.
5. The only $Z$-relevant function symbols occurring in any $w \in \Omega$ are $=z$, $\leq z$, $\circ \circ \circ$, $\text{size}$, $\text{min}$, $\text{max}$, $\phi$, $\text{neg}$, $+$, $-$, $\ast$, $\{ \circ \}$. [Note: $E_z$ has been removed.]
6. For all $w \in \Omega$ the top-level conjuncts of $\Omega$ can be rearranged to be of the form
   \[ Y \land U \land \text{EQ} \land \text{LEQ} \]
   where
   a. EQ is a conjunction of atoms of the form $z_1 = z_2$;
   b. LEQ is a conjunction of atoms of the form $z_1 \leq z_2$;
   c. U contains no occurrence of $=z$ or $\leq z$;
   d. $Y \in \text{TCF}(w)$
7. For all $t \in \text{TX}(w)$: $Y_w \vdash t$
7.11.2 The Algorithm

For each $\mu \in \Omega$ the following transformation:

\( \mu \rightarrow \text{REPLALL}(\mu, \text{var} \rightarrow \text{term}) \)

(a) $E_z = z \mu \Rightarrow \mu \rightarrow \text{REPLALL}(\mu, \text{var} : z \rightarrow E_B)$

(b) $z = E_z \mu \Rightarrow \mu \rightarrow \text{REPLALL}(\mu, \text{var} : E_z \rightarrow E_B)$

(c) $E_z \leq z \mu \Rightarrow \mu \rightarrow \text{REPLALL}(\mu, \text{var} : E_z \leq z \rightarrow E_B)$

(d) $z \leq E_z \mu \Rightarrow \mu \rightarrow \text{REPLALL}(\mu, \text{var} : z \leq E_z \rightarrow E_B)$

(e) $\neg (E_z \mu) \Rightarrow \mu \rightarrow \text{REPLALL}(\mu, \text{var} : \neg (E_z) \rightarrow E_z)$

(f) $E_z + z \mu \Rightarrow \mu \rightarrow \text{REPLALL}(\mu, \text{var} : E_z + z \rightarrow E_z)$

(g) $z + E_z \mu \Rightarrow \mu \rightarrow \text{REPLALL}(\mu, \text{var} : z + E_z \rightarrow E_z)$

(h) $E_z - z \mu \Rightarrow \mu \rightarrow \text{REPLALL}(\mu, \text{var} : E_z - z \rightarrow E_z)$

(i) $z - E_z \mu \Rightarrow \mu \rightarrow \text{REPLALL}(\mu, \text{var} : z - E_z \rightarrow E_z)$

(j) $\mu \cdot E_z \mu \Rightarrow \mu \rightarrow \text{REPLALL}(\mu, \text{var} : \mu \cdot E_z \rightarrow E_z)$

7.11.3 Proof of Weak Correctness

The invariant holds at the beginning of the PS because it is the same as the Precondition.

Conjuncts 1, 3, 4 and 5 are trivially invariant over all of the rules.

Conjunct 2 is invariant because each production is an identity substitution, e.g. production (a) is based on the identity

$E(E_z = z) \cdot E_B$.

(This is an identity only because $z$ is a variable, and thus cannot take the value $E_z$).

To prove the invariance of Conjunct 6 we must show that $w'$, the result of applying one of the rules to wff $w$, can be rearranged to fit the form $Y' \land U' \land EQ' \land LEQ'$ where $Y'$, $U'$, $EQ'$
and \( \text{LEQ}' \) are as described in Conjunct 6. We divide the proof into two parts, one covering productions (a)-(d) and the other covering productions (e)-(j).

For productions (a)-(d) we will use (a) as the typifying example without loss of generality. The substitution of production (a) occurs in the \( \text{EQ} \)-part of \( w \), replacing a conjunct of \( \text{EQ} \) by \( \text{EB} \). We will identify \( \text{LEQ}' \) with \( \text{LEQ} \) and \( \text{EQ}' \) with \( \text{EQ} \) minus the replaced conjunct. It is tempting to identify \( Y' \) with \( Y \) and \( U' \) with \( U \land \text{EB} \), but such an identification may not satisfy the necessary restrictions on \( Y' \). The reason is that in replacing \( E_z = z \) by \( \text{EB} \) one occurrence of the term \( z \) has been removed. The term \( z \) may contain an occurrence of a subterm of the form \( \{t\} \) which qualifies the term \( t \) for membership in \( \text{TX}(w) \). But it may happen that the occurrence of \( \{t\} \) in \( z \) is the only occurrence of \( t \) in \( w \) which qualifies it for membership in \( \text{TX}(w) \), and since \( z \) does not occur in \( w' \), \( t \) is not a member of \( \text{TX}(w') \). The identification of \( Y' \) with \( Y \) would then be incorrect. So we make the following argument. Let \( t_1 \ldots t_n \) be all of the terms in \( \text{TX}(w) - \text{TX}(w') \). We then identify \( Y' \) with \( Y \) minus the conjuncts mentioning \( t_1 \ldots t_n \), and we identify \( U' \) with \( U \land \text{EB} \land [\text{the conjuncts of } Y \text{ involving } t_1 \ldots t_n] \). Under these identifications \( Y', U', \text{EQ}' \) and \( \text{LEQ}' \) satisfy the restrictions of Conjunct 6. Hence, Conjunct 6 is invariant over productions (a)-(d).

To prove the invariance of Conjunct 6 over productions (e)-(j) we will choose (f) as a typifying example. [Note: we cannot select (e) as typical because it does not involve the elimination of an entire term such as the \( z \) or \( i \) of the other productions. Note also that everything said about the term \( z \) in the discussion of production (f) holds \textit{mutatis mutandis} for the term \( i \) when production (j) is in question.]

For production (f) we can start by making the following identifications:

\[ \text{EQ}' \text{ is identified with } \text{REPLALL}(\text{EQ}, E_z + z \rightarrow E_z), \text{ and} \]

\[ \text{LEQ}' \text{ is identified with } \text{REPLALL}(\text{LEQ}, E_z + z \rightarrow E_z). \]

These identifications are obviously necessary since the substitution of \( E_z \) for \( E_z + z \) conserves \( = \)-atoms and \( \leq \)-atoms. What is not so clear is how to identify \( Y' \) and \( U' \) with conjuncts from \( Y \) and \( U \). The basic problem is that the substitution of \( E_z \) for \( E_z + z \) may disturb the set \( \text{TX}(w) \) in
any of the following ways:

1. The occurrences of the term \( z \) which are being eliminated may contain all as subterms of the occurrences of some term \( t \in \text{TX}(w) \), or at least all occurrences of \( t \) which qualify \( t \) for membership in \( \text{TX}(w) \). Hence, although \( t \in \text{TX}(w) \), \( t \not\in \text{TX}(w') \).

2. Some of the occurrences of \( E_2+z \) may be as subterms of some \( t \in \text{TX}(w) \). Changing \( E_2+z \) will convert the term \( t \) to a new term \( t' \). The new term \( t' \) may be distinct from all other terms in \( \text{TX}(w) \), in which case \( \text{TX}(w')=\text{TX}(w)-\{t\}+\{t'\} \).

3. Alternatively, \( t' \) might be identical to some other term \( t_2 \in \text{TX}(w) \), and thus \( \text{TX}(w')=\text{TX}(w)-\{t\} \).

4. And, of course, the substitutions might not affect \( \text{TX}(w) \) at all, leaving \( \text{TX}(w')=\text{TX}(w) \).

Regardless of which of these circumstances occurs in the application of production (f), or which combination of these circumstances, it will always be possible to identify \( Y' \) with some subset of \( \text{REPLALL}(Y, E_2+z \rightarrow z) \) and to identify \( U' \) with \( U \) together with the remaining conjuncts of \( \text{REPLALL}(Y, E_2+z \rightarrow z) \). We will omit the details.

One comment about the form of the algorithm is worth mentioning before we proceed to the proof that the Postcondition holds at termination. We wrote the productions as PS-rules even though it might appear that TRS-rules are more natural. For example, production (f) is

\[
E_2+z \approx w \Rightarrow w \rightarrow \text{REPLALL}(w, E_2+z \rightarrow E_2). 
\]

This is quite similar to the TRS-rule

\[
E_2+z \rightarrow E_2, 
\]

which, when translated to its "equivalent" PS-rule form is

\[
E_2+z \approx w \Rightarrow w \rightarrow \text{REPL}(w, E_2+z \rightarrow E_2). 
\]

The difference is only in the use of \( \text{REPLALL} \) in the PS-production rather than \( \text{REPL} \) in the TRS-production. The difference is significant, however. If, in the course of one production execution we only changed one occurrence of \( E_2+z \) to \( E_2 \), that occurrence might be a subterm of some occurrence of \( t \in \text{TX}(w) \), changing that one occurrence into \( t' \). This would increase the size of \( \text{TX}(w') \) and would prevent Conjunct 6 of the loop Invariant from being invariant. The proof of correctness could only be saved by proving some kind of Church-Rosser property.
for the PS in this step, and although that is possible in this case, it generally complicates matters so that we wish to avoid it. This is why we wrote the algorithm as we did.

To show that the Postcondition holds at termination we need only show that the constant $E_z$ does not occur in any $w \in \Omega$; all of the rest of the Postcondition is part of the Invariant.

If $E_z$ were to occur in some $w \in \Omega$, it would have to be as an argument to one of the $Z$-related functions remaining which take arguments of Type-$Z$ are $=Z$, $\leq Z$, $o<o>$, size, min, max, neg, +, - and *. Of these, four are known (by Conjunct 3) to take only $Z$-variables as arguments, so that $E_z$ cannot be an argument to them, nor can $E_z$ become an argument to them at some intermediate stage through the action of these productions. The functions eliminated by Conjunct 3 are $o<o>$, size, min and max, leaving only $=Z$, $\leq Z$, neg, +, - and * under consideration. These six functions take $Z$-arguments at ten different argument positions. Each of those argument positions is covered by one of the productions (a)-(j). Hence, the PS cannot terminate if $E_z$ occurs in one of those ten positions, which is to say it cannot terminate if $E_z$ occurs at all in any $w \in \Omega$. This establishes the Postcondition.

7.11.4 Termination

Let $M(w)$ be a multiset of natural numbers created from $w$ in the following way: each occurrence of $E_z$ in $w$ contributes to $M(w)$ one occurrence of the number which measures its depth (distance from the root) in the tree form of $w$. Let $\ll$ be the standard well-founded order on multisets of natural numbers which is induced by the standard well-ordering $<$ on the natural numbers as described by [Dershowitz 78]. Then $M(w') \ll M(w)$ for each production. Hence the production system always terminates, and since $\Omega$ is finite, the entire algorithm terminates.
7.12 Step 12: Divide TX(w) into equivalence classes; make distinct members of TX(w) take distinct values

In Step 10 we arranged matter so that for every wff w∈Ω and for every assignment φ, if φ⊨w then Vφ assigned non-error values to each t∈TX(w). We now wish to extend this property so that any φ satisfying w assigns distinct non-error values to distinct terms t∈TX(w). We do this by dividing TX(w) into equivalence classes defined by the relation t₁ is equivalent to t₂ iff VwH₁=t₂, and systematically replacing certain occurrences of all members of an equivalence class by occurrences of a single representative member of the class.

7.12.1 Specification

Precondition: Same as the Postcondition of Step 11.

Invariant: Same as the Precondition.

Postcondition:

1. Ω∈BITZ, unquantified, finite.

2. Fw₀ ⇔ 0 Ω.

3. All Z-arguments to min, max, size, and o<o> are simple Z-variables.

4. The symbols min, max, o<o> and {o} do not occur nested in any w∈Ω.

5. The only Z-relevant function symbols occurring in any w∈Ω are =z, ≤z, o<o>, size, min, max, φ, neg, +, -, * and {o}.

6. For all w∈Ω the top level conjuncts of Ω can be rearranged to be of the form
   | Y ∧ U ∧ EQ ∧ LEQ
   where
   a. EQ is a conjunction of atoms of the form z₁=z₂;
   b. LEQ is a conjunction of atoms of the form z₁≤z₂;
   c. U contains no occurrences of =z or ≤z;
d. \( Y \in TCF(w) \).

7. For all \( t \in TX(w) \): \( Y_w \vdash t \).

8. For all \( t_1, t_2 \in TX(w) \): \( t_1 \) different from \( t_2 \) \( \Rightarrow Y_w(t_1) \neq Y_w(t_2) \).

7.12.2 Notation

For \( w \in \Omega \), define an equivalence relation \( \equiv \) on the set \( TX(w) \) such that \( t_1 \equiv t_2 \Rightarrow Y_w(t_1) = Y_w(t_2) \).

Let \( n \) be the number of equivalence classes for \( t \in TX(w) \).

Let \( \|t\| \) be the \( \equiv \)-equivalence class for \( t \in TX(w) \).

Let \( E[t] \) be a function returning a representative member of the \( \|t\| \); for convenience let us require that \( E[t] \) be a term not of the form \( \min(z) \) or \( \max(z) \) unless there are no other terms in \( \|t\| \) to choose.

Let \( Y_w \) be the \( Y \)-part of \( w \), \( U_w \) be the \( U \)-part of \( w \), \( EQ_w \) be the \( EQ \)-part of \( w \), and \( LEQ_w \) be the \( LEQ \)-part of \( w \).

7.12.3 The Algorithm

For each \( w \in \Omega \) perform the following transformation:

\[ \text{do} \]

(a) \( \min(z) \alpha U_w \wedge EQ_w \wedge LEQ_w \) and \( \min(z) \) different from \( E[\min(z)] \) \( \Rightarrow w \leftarrow Y_w \wedge \text{REPLALL}(U_w \wedge EQ_w \wedge LEQ_w, \min(z) \rightarrow E[\min(z)]) \)

(b) \( \max(z) \alpha U_w \wedge EQ_w \wedge LEQ_w \) and \( \max(z) \) different from \( E[\max(z)] \) \( \Rightarrow w \leftarrow Y_w \wedge \text{REPLALL}(U_w \wedge EQ_w \wedge LEQ_w, \max(z) \rightarrow E[\max(z)]) \)

(c) \( \{t\} \alpha w \) and \( t \) different from \( E[t] \) \( \Rightarrow w \leftarrow \text{REPLALL}(w, \{t\} \rightarrow \{E[t]\}) \)

(d) \( z \prec t \alpha w \) and \( z \) different from \( E[t] \) \( \Rightarrow w \leftarrow \text{REPLALL}(w, z \prec t \rightarrow z \prec E[t]) \)}
7.12.4 Proof of Weak Correctness

The Invariant holds at the beginning of the PS because it is the same as the Postcondition of the previous step.

Conjuncts 1, 3 and 5 are obviously invariant.

To show that Conjunct 4 is invariant we show that none of the functions min, max, \(\circ\circ\) or \(\{\circ\}\) can become nested as a result of the action of any of the productions. By Conjunct 3 none of the four symbols can become nested inside min or max, so the only question is whether any of the four productions can produce nesting inside occurrences of \(\circ\circ\) or \(\{\circ\}\).

Clearly the first two productions cannot create nesting, because if \(E[\text{min}(z)]\) or \(E[\text{max}(z)]\) is inserted into a position below \(\circ\circ\) or \(\{\circ\}\), it could only be because the term \(\text{min}(z)\) or \(\text{max}(z)\) occurred below \(\circ\circ\) or \(\{\circ\}\) before the execution of the production, contradicting the inductive hypothesis.

We are now reduced to asking whether nesting below the function symbols \(\circ\circ\) or \(\{\circ\}\) can be created as a result of productions (c) or (d). Since neither \(\circ\circ\) nor \(\{\circ\}\) can occur in any term \(t\in\text{TX}(w)\), the only nesting that might be created is if a term of the form \(\text{min}(z)\) or \(\text{max}(z)\) were substituted for an argument to \(\circ\circ\) or \(\{\circ\}\). This, however, is prevented by the definition of \(E[t]\). \(E[t]\) cannot be of the form \(\text{min}(z)\) or \(\text{max}(z)\), unless there are no other kinds of terms to choose from, in which case there can be no occurrences of \(\circ\circ\) or \(\{\circ\}\) in \(w\). Either way, nesting cannot be introduced. This completes the proof that Conjunct 4 is invariant.

The Invariance of Conjunct 6 is in two parts, one for productions (a) and (b) and the other for (c) and (d). For the first part we will use production (a) as the typical example, since (b) is so similar.

As usual, the problem in proving the invariance of Conjunct 6 is to identity \(Y'\), \(U'\), \(E\)'- and \(\text{LEQ}'\)-parts of \(w'\). In the case of production (a) this is easy: we make the following
identifications:

\[ Y' \text{ is } Y; \]
\[ U' \text{ is } \text{REPLALL}(U, \text{min}(z):=E[\text{min}(z)]); \]
\[ EQ' \text{ is } \text{REPLALL}(EQ, \text{min}(z):=E[\text{min}(z)]); \text{ and} \]
\[ LEQ' \text{ is } \text{REPLALL}(LEQ, \text{min}(z):=E[\text{min}(z)]). \]

These assignments satisfy the requirements of Conjunct 6, and so Conjunct 6 is invariant over production (a) and (b). There is a slightly tricky part to this proof regarding the identification of \( Y' \) with \( Y \). In \( w' \), the only occurrences of \( \text{min}(z) \) are in \( Y' \). One usually thinks of the \( Y \)-part of a wff as describing relations between special \( T \)-terms occurring in the rest of the wff. Nevertheless, the definition of \( Y_w \) requires that it contain complete information about the definedness and equality relations on \( TX(w) \), not just on \( TX(U_w \land EQ_w \land LEQ_w) \), and thus the fact \( \text{min}(z) \) occurs only in the \( Y \)-part of \( w' \) does not violate any conditions or definitions in the proof.

To prove the invariance of Conjunct 6 over production (c) we note that \( TX(w') = TX(w) \cup \{t\} \), i.e. that production (c) has the effect of removing the term \( t \) from the \( TX \)-set. We can thus make the following identifications:

\[ Y' \text{ is } Y \text{ without the conjuncts involving } t \text{ (which is the same as } \text{REPLALL}(Y, \{t\}:=\{E[t]\}) \text{ without the conjuncts involving } t, \text{ because } \{o\} \text{ does not occur in } Y_w \). \]
\[ U' \text{ is } \text{REPLALL}(U, \{t\}:=\{E[t]\}) \text{ together with the transformed conjuncts removed from the } Y \text{-part.} \]
\[ EQ' \text{ is } \text{REPLALL}(EQ, \{t\}:=\{E[t]\}) \]
\[ LEQ' \text{ is } \text{REPLALL}(LEQ, \{t\}:=\{E[t]\}) \]

These identifications satisfying the requirements for Conjunct 6. The proof of invariance for Conjunct 6 over production (d) is similar, and therefore the proof of invariance for Conjunct 6 is complete.

The Invariance of Conjunct 7 is derived from that of Conjunct 6. Scrutiny of that proof will reveal that no conjuncts of the form \( \lnot t \) are ever introduced into the \( Y \)-parts of any
$w \in \Omega$. Hence there can be no $t \in TX(w)$ for which $Y_w \vdash t$ at any time during the execution of the PS.

Finally, we must prove the invariance of Conjunct 2. The basic argument is that each production is an identity replacement. In the case of production (a) we know by the definition of $E$ that

$$Y_w \vdash \text{min}(z) = E[\text{min}(z)],$$

and by Conjunct 7 of the Invariant we know that

$$Y_w \vdash \text{min}(z) \text{ and } Y_w \vdash E[\text{min}(z)].$$

We can thus conclude

$$\forall Y_w \exists \text{min}(z) = E[\text{min}(z)]$$

from which we derive

$$Fw = (Y_w \land \text{REPLALL}(U_w \land EQ \land LEQ, \text{min}(z) : E[\text{min}(z)]))$$

is an identity replacement. The proof for production (b) is essentially the same. Therefore, Conjunct 2 is invariant over productions (a) and (b).

In the case of production (c) we use a similar proof. We know from the definition of $E$ that

$$Y_w \vdash t = E[t],$$

and from Conjunct 7 that

$$Y_w \vdash t \text{ and } Y_w \vdash E[t].$$

We conclude, similar to the previous proof, that

$$\forall Y_w \exists t = E[t].$$

Now this is sufficient to conclude, again as before, that

$$Fw = (Y_w \land \text{REPLALL}(U_w \land EQ \land LEQ, \{t\} : E[t])).$$

But now we invoke the fact that $\{\circ\}$ does not occur in $Y_w$ (else there would be nesting among min, max, $\circ <$ $\circ$ $>$, and $\{\circ\}$). We therefore finish with this development:
\[ \begin{align*}
& \iff Y_w \text{REPLALL}(Y_w, \{t\} \mapsto \{E[1]\}) \\
& \iff w \text{REPLALL}(Y_w, \{t\} \mapsto \{E[1]\}) \land \\
& \quad \text{REPLALL}(U_w \land EQ_w \land LEQ_w, \{t\} \mapsto \{E[1]\}) \\
& \iff w \text{REPLALL}(Y_w \land U_w \land EQ_w \land LEQ_w, \{t\} \mapsto \{E[1]\}) \\
& \iff w \text{REPLALL}(w, \{t\} \mapsto \{E[1]\})
\end{align*} \]

The last line indicates that production (c) is an identity replacement, and therefore that
Conjunct 2 is invariant over production (c). The proof for production (d) is similar. This
therefore completes the proof of invariance for Conjunct 2 and for all of the conjuncts.

We now proceed to the proof that the Postcondition holds at termination. All conjuncts of
the Postcondition are part of the Invariant except Conjunct 8, so we need only address
Conjunct 8.

Suppose at termination that there are two terms \(t_1, t_2 \in TX(w)\). Such that \(Y_w \not\equiv t_1 \not\equiv t_2\) does not hold. We will show that \(t_1\) and \(t_2\) must be the same term.

If \(Y_w \not\equiv t_1 \not\equiv t_2\) does not hold, then by the definition of \(Y_w\) and of \(\equiv\) it must be that \(Y_w \not\equiv t_1\) or
\(Y_w \not\equiv t_2\) or \(Y_w \equiv t_1 = t_2\). The first two possibilities are eliminated by Conjunct 7, so it must be
that \(Y_w \equiv t_1 = t_2\). By the definition of \(\equiv\) and \(E\) we conclude that \(E[t_1]\) is the same term as \(E[t_2]\).

At termination, of course, all patterns are unsatisfiable. The key insight is that when they
are all unsatisfiable, the pattern "\(t\) different from \(E[1]\)" is unsatisfiable as well, because if it
were satisfiable then either \(t\) is of the form \(\min(z)\) or \(\max(z)\), in which case the pattern part
of production (a) or (b) is satisfiable, or else \(\{t\}\) or \(z < t\) occurs in \(w\), in which case the
pattern for production (c) or (d) is satisfiable. Since "\(t\) different from \(E[1]\)" is unsatisfiable we
conclude that all \(\equiv\)-equivalence classes are one-element classes.

The rest is easy. Since \(t_1 \equiv t_2\), they are both in the same single-element equivalence class.
Thus, \(t_1\) and \(t_2\) must be the same term, which is what we set out to prove. This establishes
Conjunct 8 of the Postcondition.
7.12.5 Termination

Let $M(w)$ be the number of occurrences of the symbols min and max in $w$.

Let $T(w)$ be the size of $TX(w)$.

Then productions (a) and (b) reduce $M(w)$ leaving $T(w)$ invariant, while productions (c) and (d) reduce $T(w)$ leaving $M(w)$ invariant. Consequently the sum $M(w)+T(w)$ is reduced each iteration of the PS, and the PS must terminate. Also, since $\Omega$ is finite, the outer loop must be executed only a finite number of times.
7.13 Step 13: Remove neg and -z; put EQ and LEQ into Polynomial Normal Form

In this step we first remove all occurrences of -z and neg, converting them to occurrences of +z, *z and neg. Thus, the term

\((-z_1)z_2\)

is converted to

\((-1)z_1 + (-1)z_2\).

We then put each equation of EQ and each inequality of LEQ into a canonical form which we call polynomial normal form. Basically, all terms involving Z-variables are moved to the left-hand-sides of the equations and inequalities, and all terms involving the \(\{o\}\)-function are moved to the right-and-sides. We then collect "similar" terms in a fashion quite analogous to the usual techniques of elementary algebra.

7.13.1 Polynomial Normal Form

At this point in the algorithm the Z-relevant function symbols which can occur in a wff \(w(\Omega)\) are from the following list (according to Conjunct 5): =, \(\leq\), \(<\), \(\leq\), \(\lt\), min, max, \(\phi\), neg, +, -, * and \(\{o\}\).

But not all of these can occur in the EQ- or LEQ-parts of \(w\). Let us ask which of these symbols can occur on the right- or left-hand sides of EQ-atoms or LEQ-atoms. Certainly = \(z\) and \(\leq\) \(z\) cannot occur; they occur at the roots of EQ and LEQ atoms, and any occurrence below would violate Conjunct 6. Certainly \(\phi\), neg, +, -, * and \(\{o\}\) can occur. This leaves open the question of whether size, \(\circ\), min or max can occur in EQ or LEQ. A little thought reveals that they cannot, for if they did, they would have to occur somewhere in the expression tree below the \(*\)-function or the \(\{o\}\)-function, both of which are prohibited by Conjunct 4. Therefore, we conclude that EQ and LEQ contain equations and inequalities between Z-terms are constructed as follows:
1. There are "primitive" Z-terms which are either Z-variables or the constant \( \phi \) or are terms of the form \( \{t\} \).

2. These Z-terms are combined using the neg, +, - and \( \ast \)-functions.

3. In a term of the form \( \{t\} \) there are no Z-relevant function symbols occurring in \( t \); in a term of the form \( i \ast z \) there are no Z-relevant function symbols occurring in \( i \).

Under these conditions the EQ and LEQ atoms act like linear equations over a linear space, and in fact that is exactly what they are. We remarked in Chapter 4 that the zsets form a module over the ring of integers (where a module is like a vector space except that the scalars do not have to form a field but only a ring). In the module of zsets-over-the-integers (Z-over-I) the \(+_z\), \(-_z\) and neg\(_z\) operations play the role of vector addition, subtraction and negation, while the \( \ast_z\)-operation plays the role of scalar multiplication. The Z-variables play the role of vector variables in the polynomials, and the \( \{t\}\)-terms play the role of vector constants (in fact, basis vectors). The constant \( \phi \) plays the role of the zero-vector.

For an equation from EQ or an inequality from LEQ we can now define polynomial normal form (PNF).

**Definition 25:** An atomic formula of EQ\(_w\) or LEQ\(_w\) (satisfying the precondition to this Step 13) is in polynomial normal form if it matches one of the following patterns:

\[
\sum_{0 \leq i \leq m} c_i z_i = \sum_{0 \leq j \leq n} d_j \{t_j\}, \text{ or }
\sum_{0 \leq i \leq m} c_i z_i \leq \sum_{0 \leq j \leq n} d_j \{t_j\}
\]

where

1. The summations refer to zset addition.
2. The "variables" \( i, j, m \) and \( n \) are metavariables, not variables in the object formula. \( n \) is simply the number of summands on the left-hand-side and \( m \) is the number of summands on the right-hand-side.
3. An empty sum (where \( n=0 \) or \( m=0 \)) is the constant \( \phi \).
4. The terms \( c_i \) and \( d_j \) are I-terms not containing any Z-relevant function symbols.
5. The \( z_i \) are Z-variables.
6. The \( t_j \) are T-terms not containing any Z-relevant function symbols. Of course for
each $j$, $l_j\cdot TX(w)$.

7. The $\ast$-symbol is zset scalar multiplication, not integer multiplication.

8. All the variables $z_i$ are distinct, and all the terms $t_j$ are distinct terms. In other words, "like" terms have been collected on both sides of the equation or inequality. $\Box$

We will refer to the $t$-terms $c_j$ and $d_j$ as coefficients.

The entire content of this step rests on the following theorem.

**Theorem 26: Polynomial Normal Form:** Under the conditions in the Precondition to Step 13 any equation of $EQ_w$ or inequality of $LEQ_w$ can be put in Polynomial Normal Form.

**Proof:** It would be possible to give an actual production system to perform the transformation of an arbitrary $EQ$- or $LEQ$-atom to polynomial normal form, but the requirement of collecting like terms becomes rather clumsy in that notation. We will therefore rely on the reader’s long experience with linear polynomials from other domains to supply most of the algorithmic details. The knowledge expressed in the following identity schemas is all that is necessary.

(a) $\equiv i\ast(z_1 + z_2) \equiv i\ast z_1 + i\ast z_2$

(b) $\equiv i\ast z + i_2\ast z \equiv (i_1 + i_2)\ast z$

(c) $\equiv i_1\ast(i_2\ast z) \equiv (i_1\ast i_2)\ast z$

(d) $\equiv \text{neg}(z) \equiv (-1)\ast z$

(e) $\equiv z_1\ast z_2 = z_1 + (-1)\ast z_2$

(f) $\equiv (z_1 + z_2 = z_3) \equiv (z_1 = z_3\ast z_2)$

(g) $\equiv (z_1 + z_2 \leq z_3) \equiv (z_1 \leq z_3\ast z_2)$

(h) $\equiv z_1 \leq z_2 \equiv (-1)\ast z_2 \leq (-1)\ast z_1$

(i) $\equiv z + z \equiv z$

(j) $\equiv z_1\ast z_2 \equiv z_2\ast z_1 \square$

Here in this step we finally can justify the use of zsets instead of multisets. Schemas (d) through (h) in which zset negation and subtraction appear simply do not apply to multisets,
and it is not possible to put multiset equations into any kind of Polynomial Normal Form. The handling of multiset equations is far messier than the handling of zset equations (although, of course, it is possible).

The reader should be cautioned not to use any sort of cancellation law in the course of putting an equation into PNF without considering the possibility of error values. For example, the following transformations are not sound:

\[ i*z_3 + (-i)*z_3 \Rightarrow \phi \]
\[ z_1 + i*(z_3) = z_2 + i*(z_3) \Rightarrow z_1 = z_2 \]

The reason is that the term \( i*z_3 \) may take the error value \( E_z \). The Precondition to Step 13 guarantees that \( z_3 \) cannot be \( E_z \) provided that \( z_3 \) is a Z-variable or is \( \phi \) or is of the form \( \{t\} \).

But nothing in the Precondition says that the term \( i \) cannot take the value \( E_i \). For similar reasons, the transformation

\[ i*\phi \Rightarrow \phi \]

is also unsound. So, unless all the terms in question are known not to take error values, the above transformations must be rewritten as

\[ i*z_3 + (-i)*z_3 \Rightarrow (i-i)*z_3 \]
\[ z_1 + i*(z_3) = z_2 + i*(z_3) \Rightarrow z_1 + (i-i)*(z_3) = z_2 \]
\[ i*\phi \] left unchanged.

One other point should be noted. In the course of putting an EQ-atom or LEQ-atom into PNF, all occurrences of \( neg_z \) and \( -z \) are removed from the atom. Since those two symbols cannot occur elsewhere in a wff (by Conjuuncts 3, 4 and 5), this amounts to removing all occurrences of \( neg_z \) and \( -z \) from all wffs \( w(\Omega) \).

### 7.13.2 Specifications

**Precondition:** Same as Postcondition of Step 12.

**Postcondition:**
1. $\Omega \subseteq \mathbb{BITZ}$, unquantified, finite.

2. $\not\exists \omega_0 \iff \not \exists \Omega$.

3. All $Z$-arguments to $\text{min}$, $\text{max}$, $\text{size}$ and $\text{o<o>}$ are simple $Z$-variables.

4. No $Z$-related function symbol occurs below any occurrence of $\text{o<o>}$ or $\{\text{~}\}$.

5. The only $Z$-relevant function symbols occurring in any $w \in \Omega$ are $\text{z}$, $\leq_z$, $\text{o<o>}$, $\text{size}$, $\text{min}$, $\text{max}$, $\phi$, $\text{+}$, $\text{*}$ and $\{\text{~}\}$.

6. For all $w \in \Omega$ the top level conjuncts of $\Omega$ can be rearranged to be of the form $Y \land U \land EQ \land LEQ$

where

a. $EQ$ is a conjunction of atoms of the form

$$\sum_{0 \leq i \leq n} c_i z_i = \sum_{0 \leq j \leq m} d_j \{t_j\}$$

where

i. $z_i$ are distinct $Z$-variables;

ii. $t_j$ are distinct $T$-terms;

iii. $c_i$ and $d_j$ are $I$-terms;

iv. $i, j, m$ and $n$ are meta-variables in the $\sum$-notation, not object language variables.

b. $LEQ$ is a conjunction of atoms of the form

$$\sum_{0 \leq i \leq n} c_i z_i \leq \sum_{0 \leq j \leq m} d_j \{t_j\}$$

where $z_i, t_j, c_i, d_j, i, j, m, n$ are as in EQ above.

c. $U$ contains no occurrences of $\text{z}$ or $\leq_z$

d. $Y \in TCF(w)$

7. For all $t \in TX(w)$: $Y_w \vdash t$

8. For all $t_1, t_2 \in TX(w)$: $t_1$ different from $t_2$ $\Rightarrow$ $Y_w \vdash t_1 \not\equiv t_2$
7.14 Step 14: Remove all occurrences of size, \(z_t\), \(\min\) and \(\max\), thereby removing all occurrences of \(Z\)-variables as well.

This step is surely the key one of Chapter 7. Here we remove all occurrences of the function symbols size, \(o^<o>\), \(\min\) and \(\max\). As a side-effect, because of Conjuncts 3, 4 and 5, this process removes all occurrences of \(Z\)-variables as well. Thus, after this step the only \(Z\)-terms remaining will be those constructed from \(\phi, \{o\}, +\) and \(\ast\).

In the presentation of this step we are forced to abandon our usual notation. In the next subsection we define the notation to be used in this step, and outline some of the mathematical considerations for this step.

7.14.1 Notation

In the Specifications we will use these notations.

1. \(t_1 \ldots t_n\) will represent the terms in \(TX(w)\). We will usually be explicitly emphasizing occurrences of a \(Z\)-variable \(z_1\), and in any such notation we will use these further conventions:
   a. If the term \(\min(z_1)\) occurs in \(w\) but \(\max(z_1)\) does not (or if \(\max(z_1)\) occurs but \(\min(z_1)\) does not) then we will assume that the terms in \(TX(w)\) are numbered so that \(t_1\) is \(\min(z_1)\) or \(\max(z_1)\).
   b. If both \(\min(z_1)\) and \(\max(z_1)\) occur in \(w\) then we will assume that \(t_1\) is \(\min(z_1)\) and \(t_2\) is \(\max(z_1)\).

2. The notation \(\text{DISTINCT}(t_1 \ldots t_n)\) is an abbreviation for the formula
   \[
   \land_{1 \leq i \neq j \leq n} (t_i/t_j).
   \]
   This is a new notation for what has heretofore been called the Y-part of \(w\).

3. The formula \(U\) is what used to be called the U-part of \(w\). We will usually be interested in occurrences of a \(Z\)-variable \(z_1\), so we will emphasize all such occurrences by writing
   \[
   U(\text{size}(z_1), z_1^{<t_1>}, \ldots, z_1^{<t_n>}).
   \]
   If \(\min(z_1), \max(z_1)\) or both occur in \(w\) we will write
\[ U(\text{size}(z_1), z_1^{<t_1^>}, \ldots z_n^{<t_n^>}, \min(z_1)), \]
\[ U(\text{size}(z_1), z_1^{<t_1^>}, \ldots z_n^{<t_n^>}, \max(z_1)), \]
\[ U(\text{size}(z_1), z_1^{<t_1^>}, \ldots z_n^{<t_n^>}, \min(z_1), \max(z_1)). \]

By Conjunctions 3, 4 and 5 of the Precondition these formal notations display all of the kinds of occurrences that the variable \(z_1\) may have in the U-part of a formula \(w \in \Omega\). Notice that in the first line the expression \(t_1\) denotes the same term as \(\min(z_1)\) by our earlier convention. The point of this is two-fold. First, we want a uniform notation for the elements of \(TX(w)\) which includes the terms \(\min(z_1)\) and \(\max(z_1)\) if they occur. Second, we want to explicitly emphasize those occurrences of \(z_1\) which are used as arguments to \(\min\) or \(\max\) but in other contexts besides \(\min(z_1)\rangle\) and \(z_1^{<\max(z_1)}>\).

4. The symbol \(\sim\) stands for either \(\equiv\) or \(\leq\). With this notation we lump together the EQ-part of \(w\) with the LEQ-part. We will only perform operations on \(\sim\)-atoms which are sound for both \(\equiv\) and \(\leq\).

5. We assume that all of the \(Z\)-variables occurring in a \(w\) are among \(z_1, \ldots z_m\).

6. The symbols \(c_{ik} \sim c_{mk}\) and \(d_{ik} \sim d_{mk}\) denote, for each \(k\), arbitrary 1-terms subject to the restrictions of the precondition, i.e. that they contain no occurrences of any \(Z\)-related function symbols. If a variable \(z_i\) does not occur in a particular equation or inequality, we assume that the term denoted by \(c_{ik}\) is the term 0; similarly, if a term \(\{1, 2\}\) does not occur, we assume that the corresponding \(d_{ik}\) is 0.

7. Obviously the "variables" \(i, j, k, m, n, p\) are all metalinguistic devices allowing use of the \(\Lambda\) and \(\Sigma\) metalinguage notations. They are not, of course, part of the object language, nor do they directly denote any object language expressions.

7.14.2 Specifications

**Precondition**

1. \(\Omega \subseteq \text{BITZ}\), unquantified, finite

2. \(\forall w \in \Omega \implies \not\forall \Omega\)

3. All \(Z\)-arguments to \(\min\), \(\max\), \(\text{size}\) and \(\text{c}<\circ>\) are simple \(Z\)-variables

4. No \(Z\)-relevant function symbol occurs below \(\min\), \(\max\), \(\text{c}<\circ>\)
or \(\{\circ\}\) in any \(w \in \Omega\)
5. The only Z-relevant function symbols occurring in any \( w \in \Omega \) are \( =_z, \leq_z, \cdot \in \alpha, \text{size}, \min, \max, \phi, +, \cdot \) and \{ o \}.

6. If no Z-variables occur in \( w \in \Omega \), then \( w \) can be rearranged to be of the form

\[
\text{DISTINCT}(t_1 ... t_n) \land \bigwedge_{1 \leq k \leq p} \left( \phi \sim \sum_{1 \leq j \leq n} d_{\mu} \cdot \{ i_j \} \right).
\]

7. If a Z-variable \( z_i \) occurs in \( w \in \Omega \), but neither \( \min(z_i) \) nor \( \max(z_i) \) occurs in \( w \), then \( w \) can be rearranged to be of the form:

\[
\text{DISTINCT}(t_1 ... t_n) \land \\
\bigcup \left( \text{size}(z_i), z_i < t_1, ..., z_i < t_n \right) \land \\
\bigwedge_{1 \leq k \leq p} \left( \sum_{1 \leq j \leq n} c_{k} \cdot z_i \sim \sum_{1 \leq j \leq n} d_{\mu} \cdot \{ i_j \} \right).
\]

8. If a Z-variable \( z_i \) occurs in \( w \in \Omega \), and \( \min(z_i) \) occurs in \( w \) but \( \max(z_i) \) does not, then \( w \) can be rearranged to be of the form:

\[
\text{DISTINCT}(t_1 ... t_n) \land \\
\bigcup \left( \text{size}(z_i), z_i < t_1, ..., z_i < t_n, \min(z_i) \right) \land \\
\bigwedge_{1 \leq k \leq p} \left( \sum_{1 \leq j \leq n} c_{k} \cdot z_i \sim \sum_{1 \leq j \leq n} d_{\mu} \cdot \{ i_j \} \right).
\]

9. If a Z-variable \( z_i \) occurs in \( w \in \Omega \) and \( \max(z_i) \) occurs in \( w \) but \( \min(z_i) \) does not, then \( w \) can be rearranged to be of the previous form, but with "max" instead of "min".

10. Finally, if a Z-variable \( z_i \) occurs in \( w \in \Omega \) and both \( \min(z_i) \) and \( \max(z_i) \) occur in \( w \), then \( w \) can be rearranged to be of the following form:

\[
\text{DISTINCT}(t_1 ... t_n) \land \\
\bigcup \left( \text{size}(z_i), z_i < t_1, ..., z_i < t_n, \min(z_i), \max(z_i) \right) \land \\
\bigwedge_{1 \leq k \leq p} \left( \sum_{1 \leq j \leq n} c_{k} \cdot z_i \sim \sum_{1 \leq j \leq n} d_{\mu} \cdot \{ i_j \} \right).
\]

**Invariant:**

Same as Precondition

**Postcondition:**
1. \( \Omega \subseteq \text{BITZ} \), unquantified, finite

2. \( \forall w \in \Omega \implies \neg \exists \w \in \Omega \)

3. No \( Z \)-relevant function symbol occurs below \( \{ o \} \) in any \( w \in \Omega \)

4. The only \( Z \)-relevant function symbols occurring in any \( w \in \Omega \)
   are \( =_Z, \leq_Z, \phi, +, * \) and \( \{ o \} \).

5. All \( w \in \Omega \) can be rearranged to be of the form

\[
\text{DISTINCT}(t_1 \ldots t_n) \wedge U \wedge \\
\Lambda_{1 \leq k \leq p} (\phi \sim \sum_{1 \leq j \leq n} d_{k} \{ t_j \})
\]

### 7.14.3 Summary of Step 14

Before describing the actual algorithm and proving its correctness we should make a few introductory remarks about what will be done here.

The basic loop in this step will be to take a \( wff \ w \in \Omega \), which is in a special form (satisfying the Precondition/Invariant) and which contains occurrences of some \( Z \)-variable \( z_1 \), and transform it into a new \( wff \ w' \) which is in the same special form, which satisfies \( \forall w \iff \forall w' \), and which contains no occurrences of \( z_1 \). If we repeat this basic step once for each \( Z \)-variable occurring in \( w \), forming \( w', w'', \ldots, w^{(m)} \), the last formula in the sequence will contain no \( Z \)-variables at all, but it will still satisfy the Invariant and it will still satisfy \( \forall w \iff \forall w^{(m)} \).

As a side-effect of this process we will end up removing all occurrences of size, \( o < o' \), \( \text{min} \) and \( \text{max} \). This is because of Conjunct 3 of the Invariant which says that the Type-Z arguments to those four function symbols must be simple \( Z \)-variables. If there are no occurrences of any \( Z \)-variables, there cannot be any occurrences of those function symbols either. (Of course there will still be other \( Z \)-terms, based on the \( \phi- \) and \( \{ o \} \)-functions.)

In this transformation we make heavy use of the linearity properties of the \( z \)-sets. We can view the conjunction

\[
\Lambda_k (\sum_{i \in k} z_i \sim \sum_j d_{k} \{ t_j \})
\]
as a linear system of equations and inequalities. We could write it in matrix form as

\[ \mathbf{C} \mathbf{Z} = \mathbf{D} \mathbf{T} \]

where \( \mathbf{C} \) is a \( p \times m \) matrix of I-terms, \( \mathbf{Z} \) is a column vector of \( m \) Z-variables, \( \mathbf{D} \) is a \( p \times n \) matrix of I-terms and \( \mathbf{T} \) is a column vector of \( n \) terms of the form \( \{ t_i \} \). The terms \( \{ t_i \} \) are all distinct basis elements.

Now consider some \( w, \Omega \) in which neither \( \min(z_1) \) nor \( \max(z_1) \) occurs, and assume for the moment that Type-T is an infinite type, i.e. that there are infinite number of elements in the set which underlies the interpretation for Type-T. Then we can prove the following lemma.

**Lemma 27:** If under the conditions above an assignment \( \varphi \) satisfies \( w \), then there is another assignment \( \varphi' \) such that

1. \( \varphi' \) satisfies \( w \).
2. \( \varphi' \) agrees with \( \varphi \) on the non-Z-variables.
3. On each Z-variable \( z \), \( \varphi \) and \( \varphi' \) agree on all of the terms \( z \prec t_1 \prec \ldots \prec t_n \prec \text{TX}(w) \).
4. There is at most one element \( s \) in the set underlying Type-T such that \( s \) is distinct from \( V_{\varphi'}(t_1) \ldots V_{\varphi'}(t_n) \) and such that for some Z-variable \( z \) occurring in \( w \), \( \varphi'(z) \prec s \neq 0 \). In other words, for all values \( u = \{ V_{\varphi'}(t_1), \ldots, V_{\varphi'}(t_n), s \} \) and for all Z-variables \( z \) occurring in \( w \), \( \varphi'(z) \prec u = 0 \).

**Proof:** Omitted; implicit in the proof in Section 7.1.4.6.

The point of this lemma is that for any \( \varphi \) satisfying \( w \) there is another \( \varphi' \) similar to \( \varphi \) except that whereas \( \varphi(z) \prec u \) may behave chaotically when \( u = \{ V_{\varphi}(t_1), \ldots, V_{\varphi}(t_n) \} \), by contrast \( \varphi'(z) \prec u \) is zero for all but at most one \( u = \{ V_{\varphi'}(t_1), \ldots, V_{\varphi'}(t_n) \} \) (the exception being \( u = s \) of the lemma). Because of this, we can introduce a new T-variable \( t_{n+1}' \) into the formula to stand for \( s \), and new I-variables \( i_1', \ldots, i_{n+1}' \) to stand for \( z_1 \prec t_1', \ldots, z_{n+1} \prec t_{n+1}' \) and we can then eliminate all other occurrences of \( z_1 \) by substituting \( i_1', \ldots, i_{n+1}' \) for them. Of course we must prove that this is sound.

That is basically what happens in this step. There are two major complications, however.
The first involves the possible occurrence of the terms \( \min(z_1) \) and \( \max(z_1) \) in the formula. The second complication involves the possibility that Type-T may be a finite type, with a cardinality equal to or smaller than \( n \) (the number of terms in \( TX(w) \), all of which must take distinct values if \( w \) is to be satisfied.) These complications will be dealt with and explained in detail in Subsections 7.14.5 - 7.14.6.

7.14.4 A note on the Strength of the Result in this Stage

There is a semantic point to be clarified regarding this step. We will be proving statements of the form \( \models w \Rightarrow \models w' \) where \( \models \) means satisfiable in BITZV (or BITZ). Let us denote by BITZ(T) a model of BITZ constructed from the totally ordered set T. We are entitled to assert \( \models w \) if there is a totally ordered set T and an assignment \( \varphi \) such that \( \varphi \models w \). And we can prove \( \models w \Rightarrow \models w' \) if, for any T, \( \varphi \) such that BITZ(T), \( \varphi \models w \), we can construct a T', \( \varphi' \) such that BITZ(T'), \( \varphi' \models w' \). The point is that we are permitted to change the totally ordered set T to T' in the proof, from one total ordering to an extension of it.

We will not, however, take advantage of this possibility. We will prove \( \models w \Rightarrow \models w' \) the "hard way" by showing that if BITZ(T), \( \varphi \models w \), then there is a \( \varphi' \) such that BITZ(T), \( \varphi' \models w' \). Note that T is unchanged.

The significance of this is that at the end of Chapter 7 we will have proved not just that

\[ \models w_0 \iff \not \models \Omega \]

which means

\[ (\text{For all } T, \varphi: \text{BITZ}(T), \varphi \models w_0) \iff (\text{For all } T, \varphi, \text{For all } w \in \Omega: \text{BITZ}(T), \varphi \models w). \]

but instead we will have proved the stronger result that

\[ \text{For all } T: (\text{For all } \varphi: \text{BITZ}(T), \varphi \models w_0) \iff (\text{For all } \varphi, \text{For all } w \in \Omega: \text{BITZ}(T), \varphi \models w). \]

Hence, if we begin our decision procedures with \( w \in \text{BITZV} \) and then perform two Type Reductions and a Type Separation to get formulae in BI and BT, the total ordering at the end of the process is the same as the total ordering at the beginning of the process, and not some enlargement or other modification of it.
7.14.5 The Algorithm

The algorithm in this section takes a wff \( w \in \Omega \) containing \( Z \)-variables \( z_1 \ldots z_n \), and transforms it into two wffs, \( w'_1 \) and \( w'_2 \), having the same form but containing no occurrences of \( z_1 \). The procedure, then, is to, repeatedly replace \( w \in \Omega \) by \( w'_1 \) and \( w'_2 \) i.e.

\[ \Omega \left\{ \right\} \left( \Omega \left\{ w \right\} \right) \left\{ w'_1 \right\} \left\{ w'_2 \right\} \]

until no \( w \in \Omega \) contains any \( Z \)-variables. The remainder of this subsection is devoted to describing how to transform \( w \) to \( w'_1 \), \( w'_2 \).

According to the Precondition (which we must prove to hold in the next subsection) each wff \( w \in \Omega \) in which the variable \( z_1 \) occurs can be viewed as being in one of the following forms (in which all occurrences of the variable \( z_1 \) are made explicit):

(a) \[ \text{DISTINCT}(t_1 \ldots t_n) \land \left( \text{size}(z_1) \land z_1 < t_1 \ldots z_1 < t_n \right) \]
\[ \land \left( \sum_{1 \leq i \leq m \leq n} c_k \cdot z_1 \sim \sum_{1 \leq j \leq n} d_k \cdot \{ t_j \} \right) \]

(b) \[ \text{DISTINCT}(t_1 \ldots t_n) \land \left( \text{size}(z_1) \land z_1 < t_1 \ldots z_1 < t_n \right) \land \min(z_1) \land \max(z_1) \]
\[ \land \left( \sum_{1 \leq i \leq m \leq n} c_k \cdot z_1 \sim \sum_{1 \leq j \leq n} d_k \cdot \{ t_j \} \right) \]

(c) \[ \text{DISTINCT}(t_1 \ldots t_n) \land \left( \text{size}(z_1) \land z_1 < t_1 \ldots z_1 < t_n \right) \land \max(z_1) \]
\[ \land \left( \sum_{1 \leq i \leq m \leq n} c_k \cdot z_1 \sim \sum_{1 \leq j \leq n} d_k \cdot \{ t_j \} \right) \]

(d) \[ \text{DISTINCT}(t_1 \ldots t_n) \land \left( \text{size}(z_1) \land z_1 < t_1 \ldots z_1 < t_n \right) \land \min(z_1), \max(z_1) \]
\[ \land \left( \sum_{1 \leq i \leq m \leq n} c_k \cdot z_1 \sim \sum_{1 \leq j \leq n} d_k \cdot \{ t_j \} \right) \]

Case (a) corresponds to wffs in which no occurrence of \( \min(z_1) \) or \( \max(z_1) \) occurs (although \( \min \) and \( \max \) may occur with different \( Z \)-variables as arguments). Case (b) corresponds to a wff in which only \( \min(z_1) \) but not \( \max(z_1) \) occurs, and case (c) represents the converse situation. Case (d), of course, represents the case where both occur.

The transformation treats each case separately, but there are strong similarities. In each case we introduce between \( n+2 \) and \( n+4 \) new variables.
1. L-variables \(i'_1 \ldots i'_n\) (not to be confused with the meta-variable \(i\) used as an index in the summations) are introduced to take the place of the terms \(z_1^\langle t_1 \rangle \ldots z_1^\langle t_n \rangle\).

2. A new T-variable \(t_{n+1}'\) is introduced to represent a value distinct from \(t_1 \ldots t_n\).

3. Another L-variable, \(i_{n+1}'\), represents the term \(z_1^\langle t_{n+1}' \rangle\).

4. If \(\min(z_1)\) occurs in \(w\), or if \(\max(z_1)\) occurs in \(w\) and \(\min(z_1)\) does not, then a new variable \(t_1'\) is created to take the place of the term \(\min(z_1)\) (or \(\max(z_1)\)). [Recall the convention on the numbering of the T-terms \(t_1 \ldots t_n\)].

5. If both \(\min(z_1)\) and \(\max(z_1)\) occur in \(w\), then two new variables, \(t_1'\) and \(t_2'\) are created to take their places.

The algorithm treats each case separately, substituting for \(w \in \Omega\) a pair of formulae according to the following table.

**Case (a)** Substitute for \(w \in \Omega\) the following pair of wffs:

1. \(\text{DISTINCT}(t_1 \ldots t_n) \land U(\sum_{1 \leq j \leq n} (i'_j) i_1'' \ldots i_n'') \land \\
\Lambda_{1 \leq k \leq p} (\sum_{1 \leq j \leq m} c_{ik} \cdot z_i) \sim \\
\sum_{1 \leq j \leq m} (d_{j} - c_{ik}' i'_j) \ast \{t_j\})

2. \(\text{DISTINCT}(t_1 \ldots t_{n+1}) \land U(\sum_{1 \leq j \leq n+1} (i'_j) i_1'' \ldots i_n'') \land \\
\Lambda_{1 \leq k \leq p} (\sum_{1 \leq j \leq m} c_{ik} \cdot z_i) \sim \\
\sum_{1 \leq j \leq m} (d_{j} - c_{ik}' i'_j) \ast \{t_j\} + (-c_{ik}' i_{n+1}'') \ast \{t_{n+1}'\})
Case (b) Substitute for $w \in \Omega$ the following pair of wffs:

1. $$\text{DISTINCT}(t'_1, t'_2, \ldots, t'_n) \land U\left(\sum_{1 \leq j \leq n}(i'_j), i'_1, \ldots, i'_n, t'_j\right) \land$$

   $$i'_1 \neq 0 \land A_{2S,j \leq n}(t'_1 < t'_j \Rightarrow i'_j = 0) \land$$

   $$\Lambda_{1S,ksp}\left(\sum_{2S,i \leq n}c_{ik}z_i \sim \left(d_{ik} - c_{ik}x_{i'_1}\right)\{t'_1\} + \sum_{2S,j \leq n}(d_{jk} - c_{jk}x_{i'_j})\{t'_j\}\right)$$

2. $$\text{DISTINCT}(t'_1, t'_2, \ldots, t'_n, t'_{n+1}) \land U\left(\sum_{1 \leq j \leq n+1}(i'_j), i'_1, \ldots, i'_{n+1}, t'_j\right) \land$$

   $$i'_1 \neq 0 \land A_{2S,j \leq n}(t'_1 < t'_j \Rightarrow i'_j = 0) \land t'_{n+1} < t'_1 \land$$

   $$\Lambda_{1S,ksp}\left(\sum_{2S,i \leq n}c_{ik}z_i \sim \left(d_{ik} - c_{ik}x_{i'_1}\right)\{t'_1\} + \sum_{2S,j \leq n}(d_{jk} - c_{jk}x_{i'_j})\{t'_j\}\right)$$

Case (c) Substitute for $w \in \Omega$ the following pair of wffs:

1. $$\text{DISTINCT}(t'_1, t'_2, \ldots, t'_n) \land U\left(\sum_{1 \leq j \leq n}(i'_j), i'_1, \ldots, i'_n, t'_j\right) \land$$

   $$i'_1 \neq 0 \land A_{2S,j \leq n}(t'_1 < t'_j \Rightarrow i'_j = 0) \land t'_{n+1} < t'_1 \land$$

   $$\Lambda_{1S,ksp}\left(\sum_{2S,i \leq n}c_{ik}z_i \sim \left(d_{ik} - c_{ik}x_{i'_1}\right)\{t'_1\} + \sum_{2S,j \leq n}(d_{jk} - c_{jk}x_{i'_j})\{t'_j\}\right)$$

2. $$\text{DISTINCT}(t'_1, t'_2, \ldots, t'_n, t'_{n+1}) \land U\left(\sum_{1 \leq j \leq n+1}(i'_j), i'_1, \ldots, i'_{n+1}, t'_j\right) \land$$

   $$i'_1 \neq 0 \land A_{2S,j \leq n}(t'_1 < t'_j \Rightarrow i'_j = 0) \land t'_{n+1} < t'_1 \land$$

   $$\Lambda_{1S,ksp}\left(\sum_{2S,i \leq n}c_{ik}z_i \sim \left(d_{ik} - c_{ik}x_{i'_1}\right)\{t'_1\} + \sum_{2S,j \leq n}(d_{jk} - c_{jk}x_{i'_j})\{t'_j\}\right)$$
Case (d) Substitute for \( w \Omega \) the following pair of wffs:

1. \( \text{DISTINCT}(t'_1, t'_2, t'_3, \ldots, t'_n) \land \bigcup \{ \sum_{i \leq j} s_i(t'_i), i'_1 \ldots i'_n, t'_1, t'_n \} \land \\
   i'_p \neq 0 \land \Lambda_{3 \leq j \leq n}(t'_j < t'_i) \land \\
   i'_2 \neq 0 \land \Lambda_{3 \leq j \leq n}(t'_2 < t'_j) \land \\
   \Lambda_{1 \leq k \leq n}(\sum_{z_i \neq 0} c_{a_i} z_i \sim (d_{i,k} - c_{i,k})^* \{t'_i\}) \\
   + (d_{i,k} - c_{i,k})^* \{t'_i\} + \sum_{3 \leq j \leq n} (d_{j,k} - c_{j,k})^* \{t'_j\}) \\

2. \( \text{DISTINCT}(t'_1, t'_2, t'_3, \ldots, t'_n) \land \bigcup \{ \sum_{i \leq j} s_i(t'_i), i'_1 \ldots i'_n, t'_1, t'_n \} \land \\
   i'_p \neq 0 \land \Lambda_{3 \leq j \leq n}(t'_j < t'_i) \land \\
   i'_2 \neq 0 \land \Lambda_{3 \leq j \leq n}(t'_2 < t'_j) \land \\
   \Lambda_{1 \leq k \leq n}(\sum_{z_i \neq 0} c_{a_i} z_i \sim (d_{i,k} - c_{i,k})^* \{t'_i\}) \\
   + (d_{i,k} - c_{i,k})^* \{t'_i\} + \sum_{3 \leq j \leq n} (d_{j,k} - c_{j,k})^* \{t'_j\} + \\
   (-c_{i,k} + t'_i)^* \{t'_i\}) \\

7.14.6 Proof of Correctness

The proof of correctness for this step must be divided into several sub-proofs to be comprehensible; so we will divide it into numbered paragraphs.

Paragraph 7.14.6.1 is a proof that the Precondition holds at the start of this step. This is not obvious because unlike for all previous steps, the Precondition is significantly different in form from the Postcondition of the previous step.

In Paragraph 7.14.6.2 we will prove the invariance of all conjuncts of the Invariant except Conjunct 2.

Paragraph 7.14.6.3 will be devoted to proving the invariance of Conjunct 2. This is the heart of the proof.
Paragraph 7.14.6.4 will show that the Postcondition holds when the algorithm terminates.

Paragraph 7.14.6.5 contains a trivial proof of termination.

7.14.6.1 The Precondition Holds

Here we must show that the Postcondition for Step 13 implies that the Precondition for Step 14 holds.

Conjuncts (1)-(5) of the respective assertions are identical. Hence, our job is to show that Conjuncts (6)-(10) of the Precondition hold.

In the case of Conjunct 6 we assume that no Z-variables occur in the formula $w$. Since Conjunct 3 requires that the Z-arguments to $\min$, $\max$, size and $\circ<\circ$ be simple Z-variables, we conclude that none of those functions can occur in $w$. Using Conjunct 5 we thus conclude that the only Z-relevant function symbols that can occur in $w$ are $z_1, z_2, \phi, +, *$ and $\{o\}$.

Because the function $\{o\}$ may occur in $w$, $TX(w)$ might not be empty. If not, however, the $Y$-part of $w$ must be of the form $\Lambda_{1\leq j \leq n}(t \neq t_j) \land \Lambda_{1\leq i \leq n}(t_i = t_j)$.

This follows from the definition of $TCF(w)$, of which the $Y$-part of $w$ is a member, and from Conjuncts 7 and 8 of the Postcondition of Step 13 which prevent any conjuncts of the form $\neg u t$ or $t_1 = t_2$ from occurring in $w$ (unless $t_1$ and $t_2$ are the same term.) But we can rewrite the $Y$-part of $w$ as

\[
\text{DISTINCT}(t_1 \ldots t_n) \land \Lambda_{1\leq i \leq n}(t_i = t_j).
\]

[Recall the definition of the DISTINCT "predicate" that was given in Section 7.14.1.] We have thus established that each $w \in \Omega$ has Conjuncts which are of the form DISTINCT$(t_1 \ldots t_n)$ where $t_1 \ldots t_n$ are the elements of $TX(w)$.

To establish the rest of Conjunct 6, we must show that those conjuncts of $w$ which are not included in DISTINCT$(t_1 \ldots t_n)$ can be viewed as being of the form

\[
U \land \Lambda_{1\leq k \leq n}(\phi \sim \sum_{1\leq j \leq n} o_k + \{t_j\}).
\]
where \( U \) contains no \( Z \)-terms. This is fairly straightforward. We let \( U \) represent the conjuncts in the \( U \)-part of the formula (as that has been defined up through Section 7.13) plus the conjuncts left over from the \( Y \)-part, namely \( \bigwedge_{1 \leq i \leq n} (t_i = t_i') \), which are not included in \( \text{DISTINCT}(t_1, \ldots, t_n) \). We argue that \( U \) contains no \( Z \)-terms because it contains no occurrences of \( =_Z \) or \( \leq_Z \) (part of the definition of \( U \)-part.) But with only the function symbols \( =_Z, \leq_Z, \phi, +, \ast \) and \( \{\circ\} \) to work with, there is no way to construct a Boolean term \( U \) containing \( Z \)-terms that does not contain \( =_Z \) or \( \leq_Z \). Hence \( U \) contains no \( Z \)-terms.

Up to this point we have shown that the \( Y \)-part and the \( U \)-part of the formulae as they stand at the end of Step 13 can be regrouped and viewed as \( \text{DISTINCT}(t_1, \ldots, t_n) \land U \) as required by the Precondition for Step 14. All that remains to establish Conjunct 6 of the Precondition is to show how the EQ- and LEQ-parts of a wff \( w \) can be viewed as being of the form

\[
\bigwedge_{1 \leq k \leq p} \left( \phi \sim \sum_{1 \leq j \leq n} d_k \cdot t_j \right)
\]

This observation is trivial, however, because \( w \) is assumed to contain no \( Z \)-variables, and the equations and inequalities of the EQ- and LEQ-parts of \( w \) are in Polynomial Normal Form. If we imagine that perhaps some of the \( d_k \) are zero, and we make appropriate allowances for empty summations and empty conjunctions, Conjunct 6 is established.

The arguments that Conjuncts 7, 8, 9 and 10 of the Precondition hold follow closely the lines of the argument that Conjunct 6 holds. We will omit arguments for 7, 8 and 9, and just outline the argument for Conjunct 10, which subsumes the others.

For Conjunct 10 we assume that at least one \( Z \)-variable occurs in \( w \), denoted by \( z_1 \), and we assume further that both of the terms \( \min(z_1) \) and \( \max(z_1) \) occur in \( w \). We must show that \( w \) can be viewed as being of the form

\[
\text{DISTINCT}(t_1, \ldots, t_n) \land \bigwedge \left( \text{size}(z_1), z_1 < t_1 < \ldots, z_1 < t_n, \text{min}(z_1), \text{max}(z_1) \right) \land \\
\bigwedge_{1 \leq k \leq p} \left( \sum_{1 \leq j \leq n} c_k \cdot t_j = \sum_{1 \leq j \leq n} d_k \cdot t_j \right)
\]
The argument that the Y-part and U-parts of the formula \( w \) can be viewed as

\[
\text{DISTINCT}(t_1, \ldots, t_n) \land U(\text{size}(z_1), z_1 < t_1 >, \ldots, z_1 < t_n >, \min(z_1), \max(z_1))
\]

is similar to the one presented before. We need only amend that argument with one explaining why the contexts \( \text{size}(z_1), z_1 < t_1 >, \ldots, z_1 < t_n >, \min(z_1) \) and \( \max(z_1) \) are the only contexts in which the variable \( z_1 \) can occur in the U-part of \( w \).

If a Z-term occurs at all in the U-part of \( w \), then some "outermost" Z-term must occur as an argument to one of the following six function symbols: \( =, \leq, \text{size}, \circ \circ, \min \) or \( \max \). (We exclude \( + \) and \( * \) because they return Type-z values and thus their arguments can never be "outermost" Z-terms). Of these six, \( = \) and \( \leq \) cannot occur in U by the definition of U. The other four, namely size, \( \circ \circ \), \( \min \) and \( \max \) all must take Z-arguments that are simple variables. Thus, all occurrences of \( z_1 \) in U must be in one of the contexts \( \text{size}(z_1), z_1 < t_1 >, \ldots, z_1 < t_n >, \min(z_1) \) or \( \max(z_1) \).

We should perhaps point out that the notation \( U(\text{size}(z_1), z_1 < t_1 >, \ldots, z_1 < t_n >, \min(z_1), \max(z_1)) \) does not imply that all of those \( n+3 \) terms actually occur in U; only that all occurrences of the variable \( z_1 \) in U occur in one of those contexts. Recall also our notational convention that \( t_1 \) is the term \( \min(z_1) \) and \( t_2 \) is the term \( \max(z_1) \).

The fact that the EQ- and LEQ-part of \( w \) can be viewed as being of the form

\[
\Lambda_{1 \leq k \leq p} \left( \sum_{1 \leq i \leq m} c_k \cdot z_i \right) \sim \sum_{1 \leq j \leq n} d_j \cdot \{t_j\}
\]

follows the same reasoning as before, and uses the fact that the equations and inequalities of EQ and LEQ are in PNF. Again, allowances must be made for empty conjunctions and summations; the \( c_k \) and \( d_j \) coefficients for any missing terms are considered to be the term 0.

This concludes the proof that the Precondition holds at the beginning of Step 14.

7.14.6.2 Proof of Invariance for all except Conjunct 2

The invariance of Conjunct 1 is trivial since the algorithm obviously introduces no quantifiers, nor any function symbols outside the language of BITZ. \( \Omega \) remains finite because
each iteration of Step 14 removes one wff from \( \Omega \) and inserts two.

We show that Conjunct 3 is invariant by inspecting all of the wff-forms introduced as a result of Cases (a), (b), (c) and (d) in the algorithm. Any occurrences of the terms \( \text{size}(z_1), z_1 < t_1, ..., z_1 < t_n, \text{min}(z_1) \) or \( \text{max}(z_1) \) are removed, and all other occurrences of size, \( o < o \), \( \text{min} \) or \( \text{max} \) are left undisturbed by the transformation. Hence, the fact that size, \( o < o \), \( \text{min} \) and \( \text{max} \) take only \( Z \)-variables as arguments remains invariant.

[mumble about Clause 4; bug must be fixed.]

Clause 5 is clearly invariant, as inspection reveals that no occurrences of any \( Z \)-relevant functions outside of the list \( \neg, \leq, o < o, \text{size}, \text{min}, \text{max}, \phi, +, * \) or \{\} are introduced.

The invariance of Conjuncts 6-10 is basically easy but extremely tedious to establish. Essentially, one must prove 40 small lemmas, one lemma for each combination of the five Conjuncts 6-10 and the eight wff-forms, case (a.1) to case (d.2). We will give a proof sketch here for only the last case, i.e. we will show that the wff \( w' \) under case (d.2) must satisfy Conjunct 10 if the wff \( w \) from which it was transformed satisfied the entire invariant.

The wff \( w \), because it is case (d) by hypothesis, is of the form

\[
\text{DISTINCT}(t_1, t_2, ..., t_n) \land U(\text{size}(z_1), z_1, < t_1, ..., z_1, < t_n, \text{min}(z_1), \text{max}(z_1)) \land \\
\Lambda_{1 \leq k \leq p}(\sum_{1 \leq i \leq m} c_{ik} \cdot z_i \sim \sum_{1 \leq j \leq n} d_{jk} \cdot \{t_j\}).
\]

The wff \( w' \) which we are considering is the following variant of \( w \):

\[
\text{DISTINCT}(t', t'_2, t'_3, ..., t'_n, t'_{n+1}) \land U(\sum_{1 \leq j \leq n+1} (i'_j, i'_1, ..., i'_{n+1}) \land \\
i'_1 \neq 0 \land \Lambda_{3 \leq j \leq n}(t'_j < t'_j' > 0) \land t'_1 < t'_{n+1} \land \\
i'_2 \neq 0 \land \Lambda_{3 \leq j \leq n}(t'_2 < t'_j > 0) \land t'_{n+1} < t'_2 \land \\
\Lambda_{1 \leq k \leq p}(\sum_{2 \leq i \leq m} c_{ik} \cdot z_i \sim (d_{1k} - c_{1k} \cdot i'_1) \{t'_j\} + \\
(d_{2k} - c_{2k} \cdot i'_2) \{t'_2\} + \sum_{3 \leq j \leq n}(d_{jk} - c_{jk} \cdot i'_j) \{t'_j\} + \\
(-c_{1k} \cdot i'_{n+1}) \{t'_{n+1}\}))
\]
Now, if \( w' \) contains no occurrences of Z-variables (because \( z_1 \) was the only Z-variable occurring in \( w \), i.e. \( m=1 \) in \( w \)) then Conjuncts 7-10 hold trivially for \( w' \) because those conjuncts hypothesize the occurrence of at least one Z-variable. Thus, in this case the only issue is with Conjunct 6.

To show that Conjunct 6 holds for \( w' \) we must show that it is of the form

\[
\text{DISTINCT}(t_1, \ldots, t_{nn}) \land UU \land \\
\Lambda_{1 \leq \alpha < pp} (\phi \sim \sum_{1 \leq j < nn} \sum_{1 \leq k < nn-1} \sum_{1 \leq l < nn-1} \sum_{1 \leq m < nn-1} \sum_{1 \leq n < nn-1} \sum_{1 \leq \gamma < \gamma_\gamma} \delta_{j\gamma} \cdot (t_{j\gamma}))
\]

Note that in copying this form from Conjunct 6 of the invariant we have taken the liberty of changing the metavariables consistently from i to ii, j to jj, etc. This is to avoid a metavariable name-clash with i, j, etc. in the schema \( w' \).

To show that \( w' \) is of this form we identify parts of \( w' \) with parts of the form. Obviously the "conjunct" \( \text{DISTINCT}(t_1', \ldots, t_{nn}') \) from \( w' \) matches the "conjunct" \( \text{DISTINCT}(t_1, \ldots, t_{nn}) \) of the form, where \( nn=n+1 \). But to properly establish this identification we must also establish that the terms \( t_1', \ldots, t_{nn}' \) together form \( TX(w') \). They do, in fact, but we omit the proof.

The zsel equation and inequality "conjuncts" of \( w' \) (there are \( p \) of them) obviously match the equation and inequality "conjuncts" of the schema given in the Invariant, where \( pp=p \).

Because \( w' \) contains no Z-variables (by hypothesis), \( w \) must contain exactly one. Hence, \( m=1 \), and the summation \( \sum_{2 \leq i \leq m} c_i \) is empty, and is thus term \( \phi \) by convention. Hence, the left-hand sides of the equations and inequalities of \( w' \) match those of the form. The right-hand-sides also match, if we identify the coefficient terms appropriately, i.e. if we identify \( d_{j\gamma} \) of the form with the term \( (d_{j\gamma} - c_{i_\gamma}) \) for \( 1 \leq k \leq pp \) and \( 1 \leq j \leq nn-1 \) of \( w' \), and identify \( d_{nn,\gamma} \) with the term \(-c_{i_\gamma}(n+1)\) for \( nn=nn+1 \) and \( 1 \leq k \leq pp \).

The UU "conjunct" of the form matches the U-part of \( w' \) and all of the remaining parts of \( w' \) not yet accounted for. We can without trouble verify all of the properties that UU should have. For example \( =z \) and \( \leq z \) cannot occur in UU, and indeed \( =z \) and \( \leq z \) do not occur, either
in the U-part of \( w' \) (by hypothesis) or in the other conjuncts (by inspection). This concludes the proof sketch that Conjuncts 6-10 are invariant in the case that \( w' \) contains no Z-variables.

The case where \( w' \) does contain other Z-variables is only a little more complicated. Depending on which Z-variable is (re)numbered to be \( z_1 \), \( w' \) might fall under any of cases (a)-(d). Each of those cases, however, can be treated in much the same way as the above case was treated (in which \( w' \) was assumed to contain no Z-variables.) We will omit detailed proofs in the interest of brevity.

This completes the proof sketch of the invariance of Conjuncts 6-10.

### 7.14.6.3 Proof of Invariance for Conjunct 2

The heart of the correctness proof for Step 14 is the proof of Invariance for Conjunct 2.

In each case (a), (b), (c) or (d) the algorithm substitutes for \( w_\in \Omega \) a pair of wffs \( w_1' \) and \( w_2' \). To show the invariance of Conjunct 2 we must show that in each case

\[ \# w \iff \# w_1' \text{ and } \# w_2'. \]

We actually show the contrapositive, namely

\[ \# w \iff \# w_1' \text{ or } \# w_2'. \]

To be complete we would have to give a proof for each of the four cases (a)-(d). However, case (d) is more complicated than the other three and the most general. All arguments needed for cases (a)-(c) are included in those for case (d), and in fact cases (a)-(c) could be viewed as subcases of (d) except that it would cause severe notational problems. (We would have to invent notation for conditionally including certain subformulae in the wffs \( w_1' \) and \( w_2' \). As our notation is already overburdened, we choose not to do that.) Because of these considerations we will prove the invariance of Conjunct 2 only for case (d).

We will prove, then, the following three things for Case (d) wffs \( w_\in \Omega \):

1. \( \# w_1' \Rightarrow \# w \);
Here are the proofs.

**Proof that \( \not\models w'_1 \Rightarrow \not\models w \)**

Suppose that for some totally ordered type \( T \) and some assignment \( \varphi' \), we have

\[
\text{BITZ}(T), \varphi \models w'
\]

We will construct a new assignment \( \varphi \) for the variables in \( w \) such that

\[
\text{BITZ}(T), \varphi \not\models w.
\]

Notice that we leave \( T \) fixed as we discussed in Subsection 7.14.4.

Assignment \( \varphi' \) satisfies \( w'_1 \) which is of the form

\[
\text{DISTINCT}(t'_1, t'_2, \ldots, t'_n, t'_{n+1}) \land U(\sum_{1 \leq j \leq n+1} (t'_j) \land i'_1 \land \ldots \land i'_{n+1} t'_2)
\]

\[
i'_1 \neq 0 \land \Lambda_{3 \leq j \leq n+1} (t'_j < t'_1 \Rightarrow i'_j = 0) \land t'_1 < t'_{n+1} \land
\]

\[
i'_2 \neq 0 \land \Lambda_{3 \leq j \leq n+1} (t'_j < t'_1 \Rightarrow i'_j = 0) \land t'_1 < t'_{n+1} \land
\]

\[
\Lambda_{1 \leq k \leq p} (\sum_{2 \leq j \leq n} c_k x_j) = \{1\} + (d_k - c_k x'_1) + \sum_{3 \leq j \leq n+1} (-c_k x'_{n+1}) + \{1, \ldots, n\}
\]

The primed symbols \( t'_1, t'_2, \ldots, t'_{n+1} \) are all **new** variables which thus do not occur in \( w \). Because they are variables, \( \varphi' \) assigns non-error values to them.

We now construct \( \varphi \), which is to satisfy \( w \) of the form
A careful inspection of \( w \) and \( w' \) reveals the following syntactic relationships:

1. \( w \) contains the variable \( z_1 \), while \( w' \) does not; conversely, \( w' \) contains the variables \( t'_1, t'_2, t'_3 \ldots t'_n \) while \( w \) does not.

2. \( w' \) contains a number of conjuncts not corresponding to any parts of \( w \).

3. Among the parts of \( w' \) that do correspond to parts of \( w \), the relationship is expressed by the following replacements:

   a. size(\( z_1 \)) in \( w \) is replaced by \( \sum_{1 \leq j \leq n} i'_j \) in \( w' \);
   b. \( t_1 \) and \( \min(z_1) \) in \( w \) are replaced by \( t'_1 \) in \( w' \);
   c. \( t_2 \) and \( \max(z_1) \) in \( w \) are replaced by \( t'_2 \) in \( w' \);
   d. \( z_1 \) in \( w \) is replaced by \( t'_1 \{ t'_1 \} + t'_2 \{ t'_2 \} + \sum_{3 \leq j \leq n} i'_j \{ t_j \} \) in \( w' \).

4. Finally, in \( w' \) terms arising from the substitutions for \( z_1 \) have been transposed and collected to the right-hand-sides of the equations and inequalities.

We define the assignment \( \varphi \) as follows:

\[
\varphi(x) = \varphi'(x) \quad \text{for all variables } x \text{ except } z_1 \text{ occurring in } w
\]

\[
\varphi(z_1) = \varphi'(i'_1) + \varphi'(i'_2) + \varphi'(t'_1) + \varphi'(t'_2) + \sum_{3 \leq j \leq n} \varphi'(i'_j) + \{ V\varphi'(t_j) \}
\]

The proof that \( \varphi \) satisfies \( w \) is based on the following identities.

1. \( V\varphi(c_k) = V\varphi'(c_k) \) for all \( i, k \)
2. \( V\varphi(d_k) = V\varphi'(d_k) \) for all \( j, k \)
3. \( V\varphi(t_j) = V\varphi'(t_j) \) for \( 3 \leq j \leq n \)
4. \( V\varphi(t_1) = V\varphi(\min(z_1)) = V\varphi'(t'_1) \)
5. \( V\varphi(t_2) = V\varphi(\max(z_1)) = V\varphi'(t'_2) \)
6. $V_{\varphi}(z < t_j>) = V_{\varphi}(t_j')$ for all $j$
We will first establish each of these.

Identities 1, 2 and 3 all hold because the $c_w$, $d_w$ and $t_j$ terms do not contain any occurrences of the variables $z_1, t_1', t_2', t_3', \ldots t_n'$ (else Conject 4 would be violated.) Since $\varphi$ and $\varphi'$ agree on all other variables, $V_{\varphi}$ and $V_{\varphi'}$ must agree on the values assigned to these terms.

Identities 4 and 5 are similar, so we will only prove 4. Recall that our notational convention is that the term $t_1$ is the same term as $\min(z_1)$. Therefore we need only show that

$$V_{\varphi}(\min(z_1)) = V_{\varphi'}(t_1').$$

Let us expand $V_{\varphi}(\min(z_1))$ using the definition of $\varphi$.

$$V_{\varphi}(\min(z_1)) = \min(V_{\varphi}(z_1))$$

$$= \min(\varphi(z_1))$$

$$= \min(\varphi'(t_1') + \varphi'(t_1') + \varphi'(t_2') + \sum_{3 \leq j < n} \varphi'(t_j') V_{\varphi'}(t_j))$$

This last expression shows that $V_{\varphi}(\min(z_1))$ must be among the values $\varphi'(t_1'), \varphi'(t_2'), V_{\varphi'}(t_3) \ldots V_{\varphi'}(t_n)$ (unless it is undefined.) We show that it is defined, and the value is in fact $\varphi'(t_1').$

First note that $\varphi'(t_1') \ldots \varphi'(t_n')$ cannot be error values since no assignment $\varphi'$ ever assigns an error value to a variable. Second, $V_{\varphi'}(t_j)$ for $j=3..n$ cannot be an error value either, because $\varphi'$ satisfies $w'_1$ and thus

$$V_{\varphi}(\text{DISTINCT}(t_1', t_2', t_3 \ldots t_n)) = \text{true}.$$ 

This would be impossible if any of the $t_j$ took error values. (Recall that the DISTINCT-pseudopredicate is defined using $\neq$, not $\neq$.)

Since none of the terms $\varphi'(t_1') \ldots \varphi'(t_n')$, $V_{\varphi'}(t_3) \ldots V_{\varphi'}(t_n)$ are error values, the only way that $V_{\varphi}(\min(z_1))$ could be an error value is if

$$\varphi'(t_1') + \varphi'(t_1') + \varphi'(t_2') + \sum_{3 \leq j < n} \varphi'(t_j') V_{\varphi'}(t_j) = \varphi.$$
Now, because \( \varphi' \) satisfies \( w' \), we know that \( \varphi'(t'_1), \varphi'(t'_2), \ldots \varphi'(t'_n) \) are all distinct, and thus the preceding condition is equivalent to

\[
\varphi'(i'_1) = \varphi'(i'_2) = \ldots = \varphi'(i'_n) = 0.
\]

This, however, is impossible, because \( \varphi' \) satisfies the conjunct \( i'_i \neq 0 \). Therefore we now know that \( \varphi'(\min(z_1)) \) is not the error value, and is among the values \( \varphi'(t'_1), \varphi'(t'_2), \varphi'(t'_3) \ldots \varphi'(t'_n) \).

But which value is it? The answer is easy. Since \( \varphi' \) satisfies

\[
\lambda_{3 \leq j \leq n} (i'_j > i'_j = 0)
\]

it is not any of the \( \varphi'(t'_3) \ldots \varphi'(t'_n) \) which happen to be less than \( \varphi'(t'_1) \). Furthermore, since \( \varphi' \) also satisfies the conjunct \( i'_i \neq 0 \), the answer cannot be any of the \( \varphi'(t'_3) \ldots \varphi'(t'_n) \) that happen to be greater than \( \varphi'(t'_1) \) either. Hence all of the values \( \varphi'(t'_j) \) for \( j = 3 \ldots n \) are ruled out. The only other possible terms are \( \varphi'(t'_1) \) and \( \varphi'(t'_2) \). But \( \varphi' \) satisfies \( t'_1 < t'_2 \), so \( \varphi'(t'_2) \) cannot be the minimum either. We conclude that

\[
\varphi'(\min(z_1)) = \varphi'(t'_1)
\]

as required.

Finally, we must show the sixth identity, namely

\[
\varphi(z_1 < t_j >) = \varphi'(t'_j)
\]

for all \( j \).

By definition we know that

\[
\varphi(z_1) = \varphi'(t'_1) + \varphi'(t'_2) + \varphi'(t'_3) \ldots \varphi'(t'_n) + \sum_{3 \leq j \leq n} \varphi'(t'_j)\{V_{\varphi}(t'_j)\}.
\]

Since \( \varphi'(t'_1), \varphi'(t'_2), \varphi'(t'_3) \ldots \varphi'(t'_n) \) are all distinct, and since \( \varphi \) and \( \varphi' \) agree on \( t_3 \ldots t_n \), and since

\[
\varphi'(t_1) = \varphi'(t'_1) \text{ and } \varphi'(t_2) = \varphi'(t'_2),
\]

the result follows immediately.

With these identities in hand, we can now show that \( \varphi \) satisfies \( w \). We show in turn that \( \varphi \) satisfies each "conjunct" of \( w \). First since \( \varphi' \) satisfies \( \text{DISTINCT}(t'_1, t'_2, t_3 \ldots t_n) \) our identities immediately establish that \( \varphi' \) satisfies \( \text{DISTINCT}(t_1, t_2, t_3 \ldots t_n) \).

Second, we know that \( \varphi' \) satisfies
\[ U(\sum_{1 \leq j \leq n} (i'_1, i'_2, \ldots, i'_n, i'_2)) \]

From the definition of \( \varphi(z_1) \) it is direct that
\[ \forall \varphi(\text{size}(z_1)) = \sum_{1 \leq j \leq n} \varphi(i'_j) \]

From the other six identities we immediately establish that \( \varphi \) satisfies
\[ U(\text{size}(z_1), z_1<\ell_1> \ldots z_1<\ell_n>, \min(z_1), \max(z_1)) \]
as required.

Finally, we need to establish that \( \varphi \) satisfies
\[ \Lambda_{\text{siksp}} \left( \sum_{1 \leq i \leq m} c_{ik} * z_i \right) \sim \sum_{1 \leq j \leq n} d_{jk} * \{t_j\} \]
We start from the fact that \( \varphi' \) satisfies
\[ \Lambda_{\text{siksp}} \left( \sum_{2 \leq i \leq m} c_{ik} * z_i \right) \sim \left( d_{ik} - c_{ik} * i'_j \right) * \{t'_j\} + \left( d_{ik} - c_{ik} * i'_j \right) * \{t'_j\} + \sum_{3 \leq j \leq m} (d_{jk} - c_{jk} * i'_j) * \{t_j\} \].

We transpose some terms from the RHS to the LHS (an operation which is permissible whether \( \sim \) represents \( = \) or \( \leq \)). Thus, \( \varphi' \) satisfies
\[ \Lambda_{\text{siksp}} \left( c_{ik} * i'_j * \{t'_j\} + c_{ik} * i'_j * \{t'_j\} + \sum_{3 \leq j \leq m} c_{ik} * z_i \right) \sim \sum_{3 \leq j \leq m} c_{ik} * z_i \sim \sum_{3 \leq j \leq m} c_{ik} * z_i \sim \sum_{3 \leq j \leq m} d_{jk} * \{t_j\} \] + \sum_{3 \leq j \leq m} d_{jk} * \{t_j\} + \sum_{3 \leq j \leq m} d_{jk} * \{t_j\}.

Then we factor out the term \( c_{ik} \):
\[ \Lambda_{\text{siksp}} \left( c_{ik} * i'_j * \{t'_j\} + i'_j * \{t'_j\} + \sum_{3 \leq j \leq m} c_{ik} * \{t_j\} \right) + \sum_{2 \leq i \leq m} c_{ik} * z_i \sim d_{jk} * \{t_j\} + d_{jk} * \{t_j\} + \sum_{3 \leq j \leq m} d_{jk} * \{t_j\} \].

Now, since \( \varphi \) and \( \varphi' \) agree on all \( c_{ik}, d_{jk}, z_2 \ldots z_n \) and all variables not explicit in this display we can use the definition of \( \varphi(z_1) \) and the six identities to conclude that \( \varphi \) satisfies
\[ \Lambda_{\text{siksp}} \left( c_{ik} * z_1 + \sum_{2 \leq i \leq m} c_{ik} * z_i \right) \sim \sum_{3 \leq j \leq m} d_{jk} * \{t_j\} \].

When we rewrite the summations to include the separated summands we conclude that \( \varphi \)
satisfies
\[ \Lambda_{1 \leq k \leq p} \left( \sum_{1 \leq i \leq m} c_{ik} z_i \sim \sum_{1 \leq j \leq n} d_{ik} \{ t_j \} \right) \]
as required.

This concludes the proof that \( \vdash w'_1 \Rightarrow \vdash w \).

Proof that \( \vdash w'_2 \Rightarrow \vdash w \)

This proof is almost exactly the same as the previous case. We must show that if \( \varphi' \)

satisfies \( w'_2 \), which is of the form

\[ \text{DISTINCT}(t'_1, t'_2, t_3, \ldots, t_n, t'_{n+1}) \land \bigcup \left( \sum_{1 \leq j \leq n} (i'_j, j'_i) \land \delta^n \right) \land
\]

\[ \left( \delta_{i'_j} = 0 \land \sum_{1 \leq j' \leq n} (i'_j, j'_i) \land \delta^n \right) \land t'_i < t'_{n+1} \land
\]

\[ \left( \delta_{i'_j} = 0 \land \sum_{1 \leq j' \leq n} (i'_j, j'_i) \land \delta^n \right) \land t'_{n+1} < t'_2 \land
\]

\[ \Lambda_{1 \leq k \leq p} \left( \sum_{1 \leq i \leq m} c_{ik} z_i \sim \left( d_{ik} - c_{ik} i'_i \right) \{ t'_i \} \right) +
\]

\[ \left( d_{ik} - c_{ik} i'_i \right) \{ t'_i \} + \sum_{1 \leq j \leq n} (d_{ik} - c_{ik} i'_i) \{ t_j \} +
\]

\[ \left( -c_{ik} i'_i + t'_{n+1} \right) \{ t'_{n+1} \} \right)
\]
them there is some \( \varphi \) which satisfies \( w \), which is of the form

\[ \text{DISTINCT}(t_1, \ldots, t_n) \land \bigcup \left( \text{size}(z_1, z_1 < t_1, \ldots, z_1 < t_n), \min(z_1), \max(z_1) \right) \land
\]

\[ \Lambda_{1 \leq k \leq p} \left( \sum_{1 \leq i \leq m} c_{ik} z_i \sim \sum_{1 \leq j \leq n} d_{ik} \{ t_j \} \right) .
\]

Let \( \varphi' \) satisfy \( w'_2 \) and define \( \varphi \) as follows:

\( \varphi(x) = \varphi'(x) \) for all variables \( x \) except \( z_1 \), and

\[ \varphi(z_1) = \varphi'(i'_1) + \varphi'(i'_2) + \varphi'(i'_3) + \cdots + \varphi'(i'_n) + \varphi'(i'_{n+1}) + \sum_{1 \leq j \leq n} \varphi'(i'_j) \{ \varphi'(t'_j) \} +
\]

\[ + \varphi'(i'_n+1) + \varphi'(i'_{n+1}) \{ \varphi'(t'_n+1) \} .\]
The proof proceeds by establishing the following identities:

1. \( V_{\varphi}(c_{i,k}) = V_{\varphi'}(c_{i,k}) \) for all \( i, k \)
2. \( V_{\varphi}(d_{j,k}) = V_{\varphi'}(d_{j,k}) \) for all \( j, k \)
3. \( V_{\varphi}(t_j) = V_{\varphi'}(t_j) \) for all \( 3 \leq j \leq n \)
4. \( V_{\varphi}(t_1) = V_{\varphi}(\min(z_1)) = \varphi'(t_1) \)
5. \( V_{\varphi}(t_2) = V_{\varphi}(\max(z_1)) = \varphi'(t_2) \)
6. \( V_{\varphi}(z_1 < t_j >) = \varphi'(i_j') \) for all \( 1 \leq j \leq n + 1 \)
7. \( V_{\varphi}(\text{size}(z_1)) = \sum_{1 \leq j \leq n} \varphi'(i_j') \)

These identities are established in essentially the same way that the corresponding identities were established in the proof that \( \models w' \Rightarrow \models w \). It is then a simple (but tedious) matter to establish that \( \varphi \models w \), again in a way similar to that in the previous proof.

Proof that \( \models w \Rightarrow \models w_1' \) or \( \models w_2' \)

To complete the proof of invariance for Conject 2 we must show that if \( \varphi \models w \) then we can either construct \( \varphi' \) such that \( \varphi' \models w_1' \) or we can construct a (possibly different) \( \varphi' \) such that \( \varphi' \models w_2' \) (without changing the interpretation of Type-T). Recall that we are still considering \( w \) to be a class \( (d) \) wff in which both \( \min(z_1) \) and \( \max(z_1) \) occur.

The proof is divided into two cases according to which of the following two conditions applies.

1. In the totally ordered domain of interpretation for Type T there exists an elements \( \tau \) such that \( V_{\varphi}(\min(z_1)) < \tau < V_{\varphi}(\max(z_1)) \) and \( \tau \) is distinct from \( V_{\varphi}(t_1), \ldots, V_{\varphi}(t_n) \).
2. There is no such \( \tau \), i.e. the elements \( V_{\varphi}(t_1) \ldots V_{\varphi}(t_n) \) exhaust all of the elements of Type T between \( V_{\varphi}(\min(z_1)) \) and \( V_{\varphi}(\max(z_1)) \) inclusive.
We will show that if the first of these two cases arises we can form \( \varphi' \) such that \( \varphi' \models w'_2 \). We will omit most of the proof that if the second case arises then we can find \( \varphi' \) such that \( \varphi' \models w'_1 \).

We can omit this proof for several reasons. First, it bears a strong resemblance to the proof we will give (but it is simpler.) Second, in many cases Type-T might be known to be a densely ordered type such as the rationals, in which case the second of the two cases cannot arise. Finally, if we are not interested in proving the "strong form" of the preceding result, but were to allow ourselves to vary the interpretation of Type-T, the second case would not be necessary, and we would never have to use the formula \( w'_1 \) in the preceding algorithm.

For each of these reasons the proof of the second of these cases seems less important, so we will only sketch it.

Now we present the full proof for the first case. Suppose \( \varphi \models w \) where \( w \) is of the form

\[
\text{DISTINCT}(t_1 \ldots t_n) \land \bigcup \{ \text{size}(z_1), z_1 < t_1 > \ldots z_1 < t_n >, \min(z_1), \max(z_1) \} \land \\
\Lambda_{1 \leq k \leq n} \left( \sum_{1 \leq i \leq m} c_k * z_i \sim \sum_{1 \leq j \leq m} d_j * \{ t_j \} \right)
\]

We must construct \( \varphi' \) such that \( \varphi' \models w' \) where \( w' \) is

\[
\text{DISTINCT}(t'_1, t'_2, t'_3 \ldots t'_n, t'_{n+1}) \land \bigcup \{ \sum_{1 \leq j \leq n+1} (i'_j), t'_1 \ldots t'_{n+1} \land \\
i'_1 \neq 0 \land \Lambda_{1 \leq j \leq n} (t'_j < t'_j = i'_j)'_j = 0 \land t'_j < t'_{j+1} \land \\
i'_2 \neq 0 \land \Lambda_{1 \leq j \leq n} (t'_j < t'_j = i'_j)'_j = 0 \land t'_j < t'_{j+1} \land \\
\Lambda_{1 \leq k \leq n} \left( \sum_{2 \leq i \leq m} c_k * z_i \sim (d_{l_k} - c_{l_k} * i'_l) * \{ t'_l \} \right) + \\
(d_{2_k} - c_{2_k} * i'_2) * \{ t'_2 \} + \sum_{3 \leq j \leq n} (d_{l_k} - c_{l_k} * i'_j) + \\
(-c_{1_k} * i'_{n+1}) * \{ t'_{n+1} \}
\]

The formula \( w'_2 \) contains the following variables not occurring in \( w \): \( t'_1, t'_2, t'_{n+1}, i'_1 \ldots i'_{n+1} \). Let \( \tau \) be the Type-T element assumed to exist such that \( V_\varphi(\min(z_1)) < \tau < V_\varphi(\max(z_1)) \) and \( \tau \) is distinct from \( V_\varphi(t'_1) \ldots V_\varphi(t'_n) \). Then define \( \varphi' \) as follows:
\[ \phi'(t'_1) \equiv V_{\phi'}(\min(z_i)) \left(= V_{\phi}(t_1) \right) \]
\[ \phi'(t'_2) \equiv V_{\phi'}(\max(z_i)) \left(= V_{\phi}(t_2) \right) \]
\[ \phi'(t'_{n+1}) = \tau \]
\[ \phi'(i'_1) \equiv V_{\phi}(z_1 < t'_1) \]
\[ \vdots \]
\[ \phi'(i'_n) \equiv V_{\phi}(z_1 < t'_n) \]
\[ \phi'(i'_{n+1}) \equiv V_{\phi}(\text{size}(z_i) - \sum_{1 \leq j \leq n} z_i < t'_j) \]
\[ \phi'(z) \equiv V_{\phi}(\sum_{1 \leq j \leq n} z_i < t'_j) + V_{\phi}(\text{size}(z_i) - \sum_{1 \leq j \leq n} z_i < t'_j) \times \{ \tau \} \quad \text{for } 2 \leq i \leq m \]
\[ \phi'(x) \equiv \phi(x) \quad \text{for all other variables } x. \]

Notice that by this definition \( \phi \) and \( \phi' \) do not generally agree on the values they assign to the \( Z \)-variables \( z_2 \ldots z_m \). However, they are very closely related in the following ways:

Lemma 28:
\[ V_{\phi}(\text{size}(z_i)) = V_{\phi'}(\text{size}(z_i)) \quad \text{for } i=2..m \]
\[ V_{\phi}(\min(z_i)) = V_{\phi'}(\min(z_i)) \quad \text{for } i=2..m \]
\[ V_{\phi}(\max(z_i)) = V_{\phi'}(\max(z_i)) \quad \text{for } i=2..m \]
\[ V_{\phi}(z_i < t'_j) = V_{\phi'}(z_i < t'_j) \quad \text{for } i=2..m, j=1..n \]

Proof:
These four properties follow directly from the definition of \( \phi' \) and from the defining properties of \( \tau \). □

Also, since \( \phi \) and \( \phi' \) agree on all variables except \( z_1 \ldots z_m, t'_1, t'_2, t'_{n+1}, l'_1 \ldots l'_{n+1} \) and since none of these variables occur in the terms \( t_3 \ldots t_n \) or any of the \( c_{ik} \) or \( d_{ik} \), we can immediately conclude
\[ V_\varphi(t_j) = V_\varphi(t'_j) \quad \text{for } j=3..n, \]
\[ V_\varphi(c_{ik}) = V_\varphi(c'_{ik}) \quad \text{for } i=1..m, k=1..p, \]
\[ V_\varphi(d_{jk}) = V_\varphi(d'_{jk}) \quad \text{for } j=1..n, k=1..p. \]

The method of the proof is to show, one by one, that \( \varphi' \) satisfies all of the conjuncts of \( w'_2 \).

For the first "conjunct" we note that we have established (or defined) these identities:
\[ V_\varphi(t_1) = V_\varphi(t'_1), \]
\[ V_\varphi(t'_2) = V_\varphi(t_2), \]
\[ V_\varphi(t_j) = V_\varphi(t'_j) \quad \text{for } j=3..n. \]

Therefore, since
\[ \varphi = \text{DISTINCT}(t_1, t_2), \]
we can conclude that
\[ \varphi' = \text{DISTINCT}(t'_1, t'_2, t_3 \ldots t_n). \]
Furthermore, by definition \( \varphi'(t'_{n+1}) = \tau \) and \( \tau \) is distinct from \( V_\varphi(t_1) \ldots V_\varphi(t_n) \). Hence, we can conclude
\[ \varphi' = \text{DISTINCT}(t'_1, t'_2, t_3 \ldots t_n, t'_{n+1}) \] as desired.

To show that \( \varphi' \) satisfies the second "conjunct" of \( w'_2 \):
\[ U(\sum_{1 \leq j \leq n+1} \langle t'_1, t'_2 \ldots t'_n, t'_{n+1} \rangle) \]
we use, of course, the fact that \( \varphi \) satisfies the second "conjunct" of \( w \), namely
\[ U(\text{size}(z_1), z_1, t_1 \ldots z_{n+1}, \min(z_1), \max(z_1)). \]

We already know, from previous definitions or lemmas, that
\[ V_\varphi(z_1, t_j) = V_\varphi(t'_j) \quad \text{for } j=1..n; \]
\[ V_\varphi(\min(z_1)) = V_\varphi(t'_1); \]
\[ V_\varphi(\max(z_1)) = V_\varphi(t'_2). \]
From the definition of \( \varphi' \) it is simple to show as well that
\[ V_\varphi(\text{size}(z_1)) = \sum_{1 \leq j \leq n+1} \langle t'_j \rangle. \]
Hence, where new terms have been substituted for old in the U-part of $w_i'$, the new terms have the same value under $\varphi'$ as the old had under $\varphi$. This would complete the proof that $\varphi'$ satisfies the U-part of $w_i'$ except for the circumstance that $\varphi'$ assigns different values (in general) to $z_1...z_m$ than those assigned by $\varphi$, and $z_1...z_m$ may indeed occur in $U$. We have arranged things, however, so that $z_1...z_m$ can only occur in $U$ as arguments to the size, $o<o>$, min and max functions, and further that in the case of the $o<o>$-function all of the $T$-arguments are among the terms $t_1...t_n$. As we pointed out already in Lemma 28, the value of each of the terms size($z_1$)...size($z_m$), min($z_1$)...min($z_m$), max($z_1$)...max($z_m$) and $z_i^{<t_i>}$ for i=2...m and j=1...n is the same under $V_{\varphi}$ as under $V_{\varphi'}$. Since all occurrences of $z_1...z_m$ in $U$ are in the contexts of these terms, and since $V_{\varphi}$ and $V_{\varphi'}$ agree on all other corresponding parts of $U$, $\varphi'$ must satisfy the U-part of $w_i'$ as required.

The argument that $\varphi'$ satisfies the conjuncts $i_1'$#0 and $i_2'$#0 is slightly round-about. Since $\varphi$ satisfies $w$ it satisfies DISTINCT$(t_1...t_n)$, and in particular satisfies $t_1'$#t_2'. But $t_1$ and $t_2$ are the terms min($z_1$) and max($z_1$), and for $\varphi$ to satisfy min($z_1$)#max($z_1$) it is necessary for $\varphi(z_1)$ to be defined (which it must since $z_1$ is a variable) and nonempty (else $V_{\varphi}(\text{min}(z_1) # \text{max}(z_1))$ would be error rather than true.)

Since $\varphi(z_1)$ is nonempty, $V_{\varphi}(\text{min}(z_1))$ and $V_{\varphi}(\text{max}(z_1))$ are defined, and by definition of min and max for $z$-sets we know that $V_{\varphi}(z_1^{<\text{min}(z_1)>})$ and $V_{\varphi}(z_1^{<\text{max}(z_1)>})$ are defined and nonzero. But

$$\varphi'(i_1') = V_{\varphi}(z_1^{<t_1>}) = V_{\varphi}(z_1^{<\text{min}(z_1)>})$$

and

$$\varphi'(i_2') = V_{\varphi}(z_1^{<t_2>}) = V_{\varphi}(z_1^{<\text{max}(z_1)>})$$

and thus $\varphi'$ satisfies $i_1' # 0$ and $i_2' # 0$.

To show that $\varphi'$ satisfies the next "conjunct" of $w_i'$, namely

$$\Lambda_{35}(t_1^{<t_1>} = i_1' = 0),$$

we need only show (by the equivalences we have already proved) that $\varphi$ satisfies

$$\Lambda_{35}(t_1^{<\text{min}(z_1)} > z_1^{<t_1>} = 0).$$
Because under $\varphi$ all of the $t_j$ are defined this assertion is obvious; it follows immediately from the definition of the minimum of a zset. Similar considerations can be used to show that $\varphi'$ satisfies

$$\Lambda_{3\leq j \leq n} \left( t'_j < t_j \Rightarrow i'_j = 0 \right)$$

Showing that $\varphi'$ satisfies the conjuncts $t'_j < t'_{n+1}$ and $t'_{n+1} < t'_2$ amounts to showing that $\tau < V_{\varphi}(\min(z_i))$ and $\tau < V_{\varphi}(\max(z_i))$.

Both of these, however, are part of the definition of $\tau$.

Our final task is to show that $\varphi'$ satisfies the last "conject" of $w'_2$, namely

$$\Lambda_{1 \leq k \leq m} \left( \sum_{2 \leq i \leq m} c_{ik} z_i \sim (d_{ik} - c_{ik} i'_j) \{ t_j \} + \sum_{i \leq n+1} (c_{ik} + i'_{n+1}) \{ t'_{n+1} \} \right)$$

given that $\varphi$ satisfies

$$\Lambda_{1 \leq k \leq m} \left( \sum_{1 \leq i \leq m} c_{ik} z_i \sim \sum_{1 \leq j \leq n} d_{ik} \{ t_j \} \right).$$

The proof is almost entirely algebraic. We will dispense with the conjunction sign $\Lambda_{1 \leq k \leq m}$ in our manipulations, and work with a single equation (or inequality) at a time.

We start with the fact that $\varphi$ satisfies

$$\sum_{1 \leq i \leq m} c_{ik} z_i \sim \sum_{1 \leq j \leq n} d_{ik} \{ t_j \}.$$  

Each $z_i$ is a sum of the following form:

$$z_i = \sum_{1 \leq j \leq n} z_j < t'_j \rangle \{ t_j \} + \sum_{1 \leq j \leq n} t'_j \langle TX(w_2) z_j < t'_j \rangle \{ t_j \}$$

In other words, any zset $z_i$ can be decomposed into two summands, one summand representing the part of $z_i$ containing all of its copies of the elements $t_1 \ldots t_n$ and zero copies of other elements, and the other summand containing all of its copies of elements of Type-T not belonging to the set $t_1 \ldots t_n$ and zero copies each of $t_1 \ldots t_n$. We will express each $z_i$ in such a form and substitute it into our equation (or inequality), so that $\varphi$ must satisfy the following:
\[ \sum_{1 \leq i \leq m} c_k \sum_{1 \leq j \leq n} z_{i}^{t_j > t} \{ t_j \} + \sum_{t \notin TX(w)} z_{i}^{t > t} \{ t \} \]
\[ \sim \sum_{1 \leq j \leq n} d_{k} ^{t} \{ t_j \} \]

Notice that we are committing a notational abuse here, since the bound variable "t" is supposed to vary over the (non-error) elements of Type-T. Thus, t denotes a bare element, not an object language term or variable which may take a value under an assignment as for example the "variables" \( z, t, c_k \) and \( d_k \) do. We trust this will cause no problems. Note also that even if Type-T is infinite, sums of the form
\[ \sum_{t \notin TX(w)} z_{i}^{t > t} \{ t \} \]
are always defined since each zset is finite, and thus all but a finite number of the \( z_i^{t} \) must take the value 0.

We can distribute the \( \sum_{1 \leq i \leq m} c_k \) operation to get
\[ \sum_{1 \leq i \leq m} \sum_{1 \leq j \leq n} c_k z_{i}^{t > t} \{ t_j \} \]
\[ \sum_{1 \leq i \leq m} \sum_{j \notin TX(w)} c_k z_{i}^{t > t} \{ t \} \sim \sum_{1 \leq j \leq n} d_{k} ^{t} \{ t_j \} \]
Here we make use of the linearity properties of zset operations (which would not apply if we used multisets instead of zsets). Some of the \( * \)-functions represent integer multiplication, \( *_I \), and others represent zset scalar multiplication, \( *_o \). The fact that we hardly need distinguish them, nor worry about associativity between them is again due to the linearity properties of the zset operations.

The next step is to separate from the two sums on the left hand sides the case for \( i=1 \) from the cases for \( i=2..m \). Hence, \( \varphi \) satisfies
\[ \sum_{1 \leq j \leq n} c_k z_1^{t > t} \{ t_j \} + \sum_{2 \leq i \leq m} \sum_{1 \leq j \leq n} c_k z_i^{t > t} \{ t_j \} + \sum_{t \notin TX(w)} c_k z_1^{t > t} \{ t \} + \sum_{2 \leq i \leq m} \sum_{t \notin TX(w)} c_k z_i^{t > t} \{ t \}
\[ \sim \sum_{1 \leq j \leq n} d_{k} ^{t} \{ t_j \}. \]

Whether or not \( \sim \) represents \( =_z \) or \( \leq_z \) we are permitted (by the linearity properties of zsets) to transpose terms from one side of the equation (or inequality) to the other, if we
negate them in the process. The next step in the proof is to transpose the terms involving $z_1$ to the right-hand-side, and to combine similar terms (i.e., terms based on the same $\{t\}$-term). Thus, $\varphi$ satisfies

$$
\sum_{1 \leq j \leq m} \left( \sum_{2 \leq i \leq m} c_{ik} z_i <t> \right) r\{t\} \sim \\
\sum_{t \notin TX(w)} \left( \sum_{2 \leq i \leq m} c_{ik} z_i <t> \right) r\{t\}
$$

Notice also that we have commuted the $\sum$-signs and grouped the operations so that both sides of the equation (or inequality) are in the form of a "polynomial" in the elements $\{t_1\}, ..., \{t_n\}$ and $\{t\}$ for $t \notin TX(w)$.

Now we use the fact that the sets form a module over the integers, and that the singleton sets form an orthogonal set of basis "vectors" for the space. In general, for any integers $a$, $b$, and $c$, and any nonerror $t_1, t_2 \in T$ with $t_1 = t_2$, we have the following identities:

$$(a \ast \{t_1\} + b \ast \{t_2\}) = c \ast \{t_1\} + d \ast \{t_2\} \equiv (a - c \land b - d), \
(a \ast \{t_1\} + b \ast \{t_2\}) \leq c \ast \{t_1\} + d \ast \{t_2\} \equiv (a \leq c \land b \leq d).$$

Because we know that under $\varphi$ all of the $t_1, ..., t_n$ are defined and distinct, we know that we can apply the above relations (suitably generalized) and deduce that $\varphi$ satisfies both of the following equations (or inequalities):

$$
\sum_{1 \leq j \leq m} \left( \sum_{2 \leq i \leq m} c_{ik} z_i <t> \right) r\{t\} \sim \\
\sum_{1 \leq j \leq m} (d_{ik} - c_{ik} z_i <t>) r\{t\} \\
\sum_{t \notin TX(w)} \left( \sum_{2 \leq i \leq m} c_{ik} z_i <t> \right) r\{t\} \sim \\
\sum_{t \notin TX(w)} (-c_{ik} z_i <t>) r\{t\}
$$

Again, because all of the $\{t\}$ are basis sets, the second equation is equivalent to the following conjunction:

$$
\Lambda_{t \notin TX(w)} \left( \sum_{2 \leq i \leq m} c_{ik} z_i <t> \sim -c_{ik} z_i <t> \right)
$$
Note that the \( \sim \) predicate now represents \( \leq \) or \( \equiv \), according to whether before it represented \( \geq \) or \( \equiv \). We will hold the first of the two equations for a while and work with the integer conjunction.

We can scalar-multiply each of the equations (or inequalities) in our conjunction by the term \( \{\tau\} \). This is the same \( \tau \) as the one used in the definition of \( \phi' \). We thus deduce that \( \phi \) satisfies

\[
\lambda_{1_1}^{\mathbb{I}} R \left( \sum_{2 \leq m \leq n} c_k x^m \right) \left[ z_{1}^{<t>} + \{\tau\} \right] = -c_k x^m z_{1}^{<t>} + \{\tau\}.
\]

Now we can sum all of these equations (or inequalities). This proves that \( \phi \) satisfies

\[
\sum_{1 \leq m \leq n} c_k x^m \left[ z_{1}^{<t>} + \{\tau\} \right] = \sum_{1 \leq m \leq n} c_k x^m z_{1}^{<t>} + \{\tau\}.
\]

Adding this to the other equation (or inequality) that we left a few paragraphs ago, we deduce that \( \phi \) satisfies:

\[
\sum_{1 \leq m \leq n} \left( \sum_{1 \leq j \leq m} \left[ z_{1}^{<t>} + \{\tau\} \right] + \sum_{1 \leq m \leq n} c_k x^m \right) = \sum_{1 \leq m \leq n} c_k x^m z_{1}^{<t>} + \{\tau\}.
\]

We now rearrange and combine the summations. Thus, \( \phi \) satisfies

\[
\sum_{1 \leq m \leq n} c_k x^m \left( \sum_{1 \leq j \leq m} \left[ z_{1}^{<t>} + \{\tau\} \right] + \sum_{1 \leq m \leq n} c_k x^m \right) = \sum_{1 \leq m \leq n} c_k x^m z_{1}^{<t>} + \{\tau\}.
\]

Next, we notice that

\[
\sum_{1 \leq m \leq n} c_k x^m \left[ z_{1}^{<t>} + \{\tau\} \right] = \left( \text{size}(z) - \sum_{1 \leq m \leq n} z_{1}^{<t>} \right)
\]

so we can make the appropriate substitutions:

\[
\sum_{1 \leq m \leq n} c_k x^m \left( \sum_{1 \leq j \leq m} \left[ z_{1}^{<t>} + \{\tau\} \right] + \left( \text{size}(z) - \sum_{1 \leq m \leq n} z_{1}^{<t>} \right) \right) = \sum_{1 \leq m \leq n} c_k x^m z_{1}^{<t>} + \{\tau\}.
\]

Now, finally, we can see the reason for all of this algebraic manipulation. Recall that our final goal is to show that \( \phi' \) satisfies the EQ and LEQ parts of \( w'_2 \). Everything will now fold
into place. Because $\phi$ satisfies the above equation (or inequality) we know that the truth value of the following expression is true:

$$
\sum_{2 \leq i \leq m} V_{\phi}(c_{ik}) \cdot (V_{\phi}(\sum_{1 \leq j \leq n} z_{i}^{<t_{j}^{*}>}) + V_{\phi}(\text{size}(z_{i}^{<t_{j}^{*}>}) \cdot \{\tau})
$$

\sim \sum_{1 \leq j \leq n} (V_{\phi}(d_{jk}) \cdot V_{\phi}(c_{ik}) + V_{\phi}(z_{1}^{<t_{j}^{*}>})) \cdot \{V_{\phi}(t_{j})\} + V_{\phi}(-c_{ik}) + V_{\phi}(\text{size}(z_{i}^{<t_{j}^{*}>}) \cdot \{\tau}).

From the definition of $\phi'(z_{i})$ we can simplify the above expression to this:

$$
\sum_{2 \leq i \leq m} V_{\phi}(c_{ik}) \cdot \phi'(z_{i}) \sim \sum_{1 \leq j \leq n} (V_{\phi}(d_{jk}) \cdot V_{\phi}(c_{ik}) + V_{\phi}(z_{1}^{<t_{j}^{*}>})) \cdot \{V_{\phi}(t_{j})\} + V_{\phi}(-c_{ik}) \cdot \phi'(i_{n+1}' \cdot \{\tau}).
$$

From the definition of $\phi'(i_{n+1}')$ we simplify further:

$$
\sum_{2 \leq i \leq m} V_{\phi}(c_{ik}) \cdot \phi'(z_{i}) \sim \sum_{1 \leq j \leq n} (V_{\phi}(d_{jk}) \cdot V_{\phi}(c_{ik}) + V_{\phi}(z_{1}^{<t_{j}^{*}>})) \cdot \{V_{\phi}(t_{j})\} + V_{\phi}(-c_{ik}) \cdot \phi'(i_{n+1}') \cdot \{\tau}.
$$

We recall the following identities, either previously established or definitions:

- $V_{\phi}(c_{ik}) = V_{\phi}(c_{ik})$ for all $i, k$
- $V_{\phi}(d_{jk}) = V_{\phi}(d_{jk})$ for all $j, k$
- $V_{\phi}(z_{1}^{<t_{j}^{*}>}) = \phi'(i_{j})$ for all $1 \leq j \leq n$
- $\tau = \phi'(i_{n+1})$.

When we make the appropriate substitutions we prove that the following is true:

$$
\sum_{2 \leq i \leq m} V_{\phi}(c_{ik}) \cdot \phi'(z_{i}) \sim \sum_{1 \leq j \leq n} V_{\phi}(\phi'(c_{ik}) \cdot \phi'(i_{j}) \cdot \{V_{\phi}(t_{j})\} + V_{\phi}(-c_{ik}) \cdot \phi'(i_{n+1}') \cdot \{i_{n+1}'\}).$$
Notice that this entire equation (or inequality) is written in terms of \( \varphi' \) except for the occurrences of \( V_\varphi(t_j) \).

By separating the cases \( j=1 \) and \( j=2 \) from the summation and using the facts that

\[
V_\varphi(t_1) = \varphi'(t_1),
\]
\[
V_\varphi(t_2) = \varphi'(t_2),
\]
\[
V_\varphi(t_j) = V_\varphi(t_j) \quad \text{for } j \neq 1, 2
\]

we deduce the truth of this schema:

\[
\sum_{2 \leq i \leq m} V_\varphi(c_i + x_i) \sim V_\varphi'(d_{ik} - c_{ik} + i_i') + V_\varphi'(d_{ik} - c_{ik} + i_i') \{ \varphi'(t_1) \} + V_\varphi'(d_{ik} - c_{ik} + i_i') \{ \varphi'(t_2) \} + V_\varphi'(d_{ik} - c_{ik} + i_i') \{ \varphi'(t_j) \} + \sum_{3 \leq i \leq n} V_\varphi'(d_{ik} - c_{ik} + i_i') \{ \varphi'(t_j') \} + V_\varphi'(d_{ik} - c_{ik} + i_i') \{ t_{n+1} \}
\]

But if the truth value of this is true, that is the same as saying that \( \varphi' \) satisfies

\[
\sum_{2 \leq i \leq m} c_i + x_i \sim (d_{ik} - c_{ik} + i_i') \{ t_{i_1} \} + (d_{ik} - c_{ik} + i_i') \{ t_{i_2} \} + \sum_{3 \leq i \leq n} (d_{ik} - c_{ik} + i_i') \{ t_{i_1} \} + c_{ik} + i_i' \{ t_{n+1} \}
\]

Taking the conjunction \( \bigwedge_{1 \leq i \leq p} \) over the above equations (or inequalities) yields the result desired, that \( \varphi' \) does indeed satisfy the EQ and LEQ parts of \( w' \).

This concludes the proof of invariance of Conjunct 2, and of the entire Invariant.

7.14.6.4 The Postcondition

At termination Conjuncts 1, 2 and 3 of the Postcondition hold because they are part of the Invariant.

The algorithm does not terminate with any occurrences of \( \varphi \)-variables in any \( w<\Omega \). But by Conjunct 3 of the Invariant the min, max, size and \( \circ<\circ \) take only \( \varphi \)-variables as \( \varphi \)-arguments. As a result, we conclude that these four functions do not occur in the formulae at termination, and that the only \( \varphi \)-relevant function symbols left are =, ≤, \varphi, +, * and \( \{ \circ \} \).
This establishes Conjunct 4 of the Postcondition.

Conjunct 5 of the Postcondition follows directly from Conjunct 5 of the Invariant and from the fact that no Z-variables occur in any \( w \in \Omega \) at termination.

This completes the proof of weak correctness for the algorithm.

7.14.6.5 Termination

The algorithm terminates because each step in the algorithm substitutes for \( w \in \Omega \) a pair of wffs \( w'_1 \) and \( w'_2 \) each of which contains one fewer Z-variables than \( w \). (Notice: we do not say "one fewer occurrences of Z-variables", but rather "one fewer Z-variables".)

7.14.7 The Introduction of Integer Multiplication

All through Chapter 6 and until Step 14 of Chapter 7 it was the case that our algorithm never introduced any of the three functions \( *_1, /_1 \) or \( *_2 \) into any formula. If those symbols were present in the original wff \( w_0 \) they were inherited by some or all of the wffs \( w \in \Omega \), but no multiplication or division symbols (nor any function symbols from outside BITZV other than elem) were ever introduced.

In this step, however, we see the "introduction" of integer multiplication for the first time. More precisely what happens is that the scalar multiplication operator \( *_2 \) in the terms of the form \( c_{1k} *_2 z_1 \) occurring in the wff \( w \) (using our standard notational conventions) is removed and replaced by the \( n \) or \( n+1 \) occurrences of the integer multiplication operator \( *_1 \) in the terms of the form \( c_{1k} *_1 z'_1 \) occurring in \( w'_1 \) and \( w'_2 \). In other words, the scalar multiplications, if they occur, are "translated" to integer multiplications. But it is still the case that if none of the three functions \( *_2, *_1 \) or \( /_1 \) occur in \( w_0 \), then none of them are ever introduced, even in this step. [Note: we do not consider such terms as \( 0 *_1 i \) or \( 1 *_1 i \) as true occurrences of multiplication; such terms do arise technically in our algorithm because of the uniform notation we have used in expressing the algorithm, but the reader can surely see that such terms are not necessary.]
One important point to realize is that if the scalar term $c_{ik}$ in the term $c_{ik}z_j$ is an integer constant, then all of the multiplications introduced to replace it, which are of the form $c_{ik}z_j^t$, are multiplications by a constant. We can thus strengthen our statement in the following way. Let us say that a wff $w$ contains only constant multiplications wff (1) every occurrence of $z_1$ has one argument which contains no variables, and (2) in every occurrence of $z_2$ the integer argument contains no variables. Then we can say that if the original formula $w_0$ contains only constant multiplications, this property will be true of all $w^\Omega$ all through the algorithms in this thesis.

The importance of containing only constant multiplication is that constant multiplication can be reduced to addition, and thus decision procedures such as integer programming or Presburger's algorithm can be used to decide the formula, but if nonconstant multiplication is permitted in a formula (language) the decision problem for validity is recursively unsolvable. Notice that in none of the example programs presented in this thesis have we needed to use nonconstant multiplication or (division) in any of our inductive assertions or in the program text, and that the process of VC-construction does not introduce nonconstant multiplication if it is not present to begin with. Hence, the VCs for those programs all contain only constant multiplication, and are decidable.
7.15 Step 15: Remove all remaining Z-terms and Z-related function symbols: =,
≤, 0, +, * and \( \{t\} \)

In this step we remove the last remaining Z-related function symbols from the wffs \( w(\Omega) \).
We thus complete the Type Reduction of BITZ to BIT.

The basic idea is to take advantage of the linearity properties of Type-Z. We take a zset
equation or inequality of the form
\[
\phi = \sum_{1 \leq j \leq n} d_j \{t_j\}, \quad \text{or}
\]
\[
\phi \leq \sum_{1 \leq j \leq n} d_j \{t_j\}
\]
and replace it by the corresponding integer equation or inequality
\[
0 = \sum_{1 \leq j \leq n} d_j t_j, \quad \text{or}
\]
\[
0 \leq \sum_{1 \leq j \leq n} d_j t_j.
\]
We can do this because all of the \( t_j \)-terms take distinct values under any \( \psi \) satisfying \( w(\Omega) \),
and because the singleton zsets form a linearly independent basis over the integers for the space of all zsets.

7.15.1 Specifications

**Precondition:** Same as Postcondition for Step 14

**Invariant:** Same as Precondition

**Postcondition**

1. \( \Omega \subseteq \text{BITZ, unquantified, finite} \)
2. \( \not\forall w_0 \iff \not\Omega \)
3. No Z-relevant function symbols occur in any \( w(\Omega) \)
7.15.2 The Algorithm

For each \( w \in \Omega \) perform the following transformation:

\[
\begin{align*}
\text{do} & \quad \phi = \sum_{1 \leq j \leq n} d_j \cdot \{t_j\} : \iff 0 = \sum_{1 \leq j \leq n} d_j \\
\text{od} & \quad \phi \leq \sum_{1 \leq j \leq n} d_j \cdot \{t_j\} : \iff 0 \leq \sum_{1 \leq j \leq n} d_j
\end{align*}
\]

7.15.3 Proof of Weak Correctness

The Precondition holds because it is the Postcondition of the previous step.

The invariance of Conjuncts 1, 3 and 4 of the Invariant is trivial.

The invariance of Conjunct 5 is quite simple. Each execution of a production transforms an \( r_2 \)-atom into an \( r_1 \)-atom or a \( r_2 \)-atom to a \( r_1 \)-atom. If the original formula \( w \) is of the form

\[
\text{DISTINCT}(t_1, \ldots, t_n) \land U \land \\
\lambda_{1 \leq k \leq p} (\phi \sim \sum_{1 \leq j \leq n} d_j \cdot \{t_j\})
\]

then the formula transformed by either production, which we will call \( w' \), is of the form

\[
\text{DISTINCT}(t_1, \ldots, t_n) \land U \land \\
\lambda_{1 \leq k \leq p-1} (\phi \sim \sum_{1 \leq j \leq n} d_j \cdot \{t_j\}) \land 0 \sim \sum_{1 \leq j \leq n} d_j
\]

If we include the last atomic-formula conjunct of \( w' \), \( 0 \leq \sum_{1 \leq j \leq n} d_j \) in the \( U \)-part of \( w' \), then \( w' \) clearly fits the required form (where \( p \) in \( w' \) is \( p-1 \) from \( w \), i.e. there is one fewer \( U \)-equation or \( U \)-inequality in \( w' \)).

The invariance of Conjunct 2 follows from the following valid schemas:

\[
\begin{align*}
\text{DISTINCT}(t_1, \ldots, t_n) & \Rightarrow ( (\phi = \sum_{1 \leq j \leq n} d_j \cdot \{t_j\}) = (0 = \sum_{1 \leq j \leq n} d_j) ) \\
\text{DISTINCT}(t_1, \ldots, t_n) & \Rightarrow ( (\phi \leq \sum_{1 \leq j \leq n} d_j \cdot \{t_j\}) = (0 \leq \sum_{1 \leq j \leq n} d_j) )
\end{align*}
\]

Since each \( w \in \Omega \) contains the "conjunct" \( \text{DISTINCT}(t_1, \ldots, t_n) \), the replacements specified in the
two productions are identity substitutions. This concludes the proof of invariance.

When the PS terminates there are no occurrences of \( =_Z \) or \( \leq_Z \) in any \( w \in \Omega \). The only remaining \( Z \)-relevant function symbols are \( \phi, \ast, \cdot \) and \( \{ o \} \). But as none of these function symbols are of the kind that take a \( Z \)-argument but return a non-\( Z \)-value, none of them can occur in \( w \) either at termination. This establishes Conjunct 3 of the Postcondition. Since Conjuncts 1 and 2 of the Postcondition are also part of the Invariant, the entire Postcondition is established.

7.15.4 Termination

Each production reduces the total number of occurrences of \( =_Z \) and \( \leq_Z \) in \( \Omega \).
8. An Example

Any algorithm of the length and complexity of the one given in Chapters 6 and 7 is extremely difficult to understand without an example. In this Chapter we will apply the Type Reduction procedure of Chapter 6 to a BITZV-formula which has been selected to exercise most of the interesting parts of the algorithms. This will help the reader understand just which parts of the algorithm are critical, and will also perhaps clarify any ambiguous notation used in the algorithm.

As we trace the algorithm's action we will be forced to make a number of compromises with formality. Naturally, since the algorithm is highly nondeterministic, we cannot follow all legal execution paths. Hence we will select one, usually the one which leads to the shortest formula with which to continue our trace. Secondly, the algorithm as written follows the course of an entire set of wffs Ω; we, however, will always be looking at a few wffs, elements of Ω, usually those having the most interesting subsequent course through the Type Reduction procedure. The size of Ω grows too large for anything else to be practical. Finally, in order to keep the size of the formulae down to reasonable limits we will feel free to make certain simplifications from time to time, especially propositional simplifications. These simplifications should be part of any implementation of our algorithms, but they were not explicitly included in the algorithm as written so as not to complicate the proof of its correctness.

The wff we will use as an example is as follows:

\[
\text{i} \leq \text{j} \land j \leq \text{k} \land k \leq \text{l} \land \text{m} \land \text{ms} \land \text{ord}(v_1[i..n]) \land \\
v_1[i..n] = v_2 \land v_1[k] = v_2[m] \land \text{l}b(v_2) = \text{l}b(v_3) \land \{v_2\} = \{v_3\} \land \\
\text{ord}(v_3) \Rightarrow \\
\min(v_1[k..m]) = v_3[l] \land \text{len}(v_3[k..m]) \leq \{v_1\} < v_2[k] \land v_2 = v_3
\]

This wff will be known as \(w_0\) all through the execution of the algorithm. It is in fact valid in BITZV, and the reader might wish to spend a few minutes convincing himself of that fact.
before proceeding. The variables \( v_1, v_2 \) and \( v_3 \) are, of course, Type-V, and the variables \( i, j, k, l, m, n \) are all Type-I. Although there are no variables of Types \( B, T \) or \( Z \), there obviously are terms of all three of those types. This wff is not a verification condition for any interesting program; it was invented only for the purpose of using as many interesting operations as possible in a short formula.

We will now begin tracing the execution of the Type Reduction procedure in Chapter 6. Each Section of this chapter will correspond to a "step" or Section of that chapter.

8.1 Step 6.1: Simplify the formula, canonicalize V-terms

8.1.1 Stage 1: Remove all occurrences of \( \text{len}, \text{min}_v, \text{max}_v, v \) and the one-way and two-way conditional functions.

There are no occurrences in \( w_0 \) of the \( \text{max}_v \) function or of the conditional functions. However, there are occurrences of \( \text{len}, \text{min}_v \) and \( v \). When we apply the productions of Step 1, Stage 1 we transform \( w_0 \) to this:

\[
\begin{align*}
\text{lb}(v_1) & \leq i \land i \leq j \land j \leq k \land k \leq l \land m \land n \land \text{sub}(v_1) \land \text{ord}(v_1[i..n]) \\
& \quad \quad \left( \#v_1[i..n] \land \#v_2 \mid v_1[i..n] \equiv v_2 \right) \land v_1[k] = v_2[m] \\
\text{lb}(v_2) & = \text{lb}(v_3) \land \{v_2\} = \{v_3\} \land \text{ord}(v_3) \geq 0. \\
\text{min}\{v_1[k..m]\} & = v_3[k] \\
\text{ub}(v_3[k..m]) - \text{lb}(v_3[k..m]) + 1 & \leq \{v_1\} < v_2[k] \\
\left( \#v_2 \land \#v_3 \mid v_1 \equiv v_3 \right)
\end{align*}
\]

8.1.2 Stage 2: Remove all occurrences of \( \#_v \)

There are four occurrences of \( \#_v \) in \( w \), and three of them apply to simple variables. Production (a) of the algorithm will replace the terms \( \#v_2 \) and \( \#v_3 \) by \( \text{true} \). However, since they are parts of conjunctions we will drop the occurrences of \( \text{true} \) from the wff in the interests of simplicity.
Notice that when production (c) is applied to the term \#v_1[j..n] it will create a term \#v_1 which production (a) will then simplify to true.

After Stage 2, wff w is as follows:
\[
ib(v_1)i \land isj \land jsk \land ksl \land ism \land msn \land nsub(v_1) \land ord(v_1[i..n]) \land \\
\left( uj \land \mu n \land lb(v)sj \land jsn+1 \land nsub(v) \mid v_1[j..n] \equiv v_2 \right) \land \\
\min\left\{ v_1[k..m] \right\} = v_3[l] \land \\
ub(v_3[k..m]) - lb(v_3[k..m]) + 1 \leq \left\{ v_1, v_2, v_3 \right\} \land \\
v_1 \equiv v_3
\]

The very last conjunct, \( v_1 \equiv v_3 \), we have simplified from \( \left( \text{true} \land \text{true} \mid v_1 \equiv v_3 \right) \).

At this point, because Step 1 is a Markov production system, all of Stage 1 will be re-executed, followed by a re-execution of Stage 2. These have no effect, however, since no productions are eligible to fire.

8.1.3 Stage 3: Remove occurrences of the three-way conditional function

There are no occurrences of \( \left( o \mid o \mid o \mid o \right)_v \) in w, so this stage does nothing.

8.1.4 Stage 4: Remove occurrences of the Assignment function

As there are no occurrences of \( <o, o, o> \) in w this Stage has no effect either.

8.1.5 Stage 5: Replaces occurrences of \( v[i] \) with occurrences of the elem function

In this stage we replace occurrences of terms of the form \( v[i] \) by terms of the form \( v[i..i] \). After this stage wff w is this:
\[ \begin{align*}
\text{lb}(v_1) & \leq i \land i \leq j \land j \leq k \land k \leq l \land m \land n \land n \text{sub}(v_1) \land \\
\text{ord}(v_1[1..n]) & \land (u \land u \land \text{lb}(v_1) \leq j \land j \leq n \land n \text{sub}(v_1) \mid v_1[1..n] = v_2) \land \\
\text{elem}(v_1[k..k]) & = \text{elem}(v_2[m..m]) \land \text{lb}(v_2) = \text{lb}(v_3) \land \{ v_2 \} = \{ v_3 \} \land \\
\text{ord}(v_3) & >. \\
\text{min}(\{ v_1[k..m] \}) & = \text{elem}(v_3[1..1]) \land \\
\text{ub}(v_3[k..m]) - \text{lb}(v_3[k..m]) + 1 \leq v_1 & \land \text{elem}(v_2[k..k]) > \land \\
v_1 & = v_3
\end{align*} \]

The re-execution of Stages 1-5 causes no further change.

8.1.6 Stage 6: Remove nested occurrences of \( v[1..2] \)

As there are no nested occurrences of the \( \circ[\circ, \circ] \)-function, this stage makes no change in \( w \).

8.1.7 Stage 7: Rewrite certain miscellaneous expressions

Of the eight productions of this stage, only productions (d)-(i) apply. All of these except productions (d) and (e) have the effect of replacing a bare \( V \)-variable \( v \) by the term \( v[\text{lb}(v)..<\text{ub}(v)] \). Production (d) replaces a term of the form \( \text{lb}(v[1..j]) \) by the term \( \text{ub}(v[1..j]) \); production (e) is similar, substituting for \( \text{ub}(v[1..j]) \) the term \( \text{ub}(v[1..j]) \). After these transformations the wff \( w \) looks like this:
After the productions of Stage 7 are executed, Stages 1-7 are re-executed (because this is still a Markov-production system.) During this re-execution the only changes that occur are that the two occurrences of \( * \) are removed by productions (e) and then (a) of Stage 2. We are then left with the following formula:

\[
\begin{align*}
\text{lb}(v_1) & \land \text{isj} \land jsk \land ksl \land lsm \land msn \land nsub(v_1) \land \\
\text{ord}(v_1[i..n]) & \land (ui \land unm \land \text{lb}(v_2)isj \land jsn \land nsub(v_1) \mid v_1[i..n]=v_2[\text{lb}(v_2)\ldots\text{ub}(v_2)]) \land \\
\text{elem}(v_1[k..k]) & = \text{elem}(v_2[m..m]) \land \text{lb}(v_2)=\text{lb}(v_3) \land \{v_2[\text{lb}(v_2)\ldots\text{ub}(v_2)]\} \{b_3[\text{lb}(v_3)\ldots\text{ub}(v_3)]\} \land \\
\text{ord}(v_3[\text{lb}(v_3)\ldots\text{ub}(v_3)]) & \Rightarrow \quad \\
\min(\{v_1[k..m]\}) & = \text{elem}(v_3[i..l]) \land \\
(\#v_3[k..m] \mid k) - (\#v_3[k..m] \mid l) + 1 \leq \{v_3[\text{lb}(v_3)\ldots\text{ub}(v_3)]\} \text{ <elem}(v_2[k..k]) \} \land \\
v_1[\text{lb}(v_1)\ldots\text{ub}(v_1)] & = v_3[\text{lb}(v_3)\ldots\text{ub}(v_3)]
\end{align*}
\]

At this point we will again make a slight simplification that is not in the algorithm. Because \( j, k, m \) and \( n \) are all variables, \( u_i, u_k, u_m \) and \( u_n \) are all identically true. We will remove these terms from the formula to avoid clutter; if they are not removed, they will remain untouched by any transformation from now until the end of the algorithm. After this cosmetic change the formula looks like this:
This is the formula \( w \) at the completion of all seven stages of Step 1. The reader can readily verify that it satisfies all of the Conjectures of the Postcondition for Step 1 except Conjecture 2. The verification that it satisfies Conjecture 2, that \( Fw \iff Fw_0 \), is not immediate. The best thing to do is to study the correctness proof as given in Chapter 6.

Step 2: Put formula into DNF; separate the disjuncts

In this step we convert a problem of deciding validity to one of deciding unsatisfiability. We use the relation \( Fw \iff \not\exists T(w) \), so that deciding whether \( \not\exists T(w) \) is unsatisfiable is the same as deciding whether \( w \) is valid. We then put \( \not\exists T(w) \) into three-valued Disjunctive Normal Form, and let \( \Omega \) be the set of disjuncts. We can then assert that \( Fw_0 \iff \not\exists \Omega \).

In the present case we can view \( w \) as being of the form

\[ \Lambda_{P_i} \Rightarrow \Lambda_{Q_j} \]

We will assume for the moment that the occurrence of \( (o \mid o)_B \) in the antecedent is replaced by \( \land \). This is not always sound, but among the conjuncts in the antecedent of \( \Rightarrow \) it is sound because \( T((b_1 \mid b_2)) = T(b_1 \land b_2) \) for any B-terms \( b_1 \) and \( b_2 \). We make the substitution because it makes the computation of DNF easier. With these conventions a DNF-equivalent for \( \not\exists T(w) \) is
\[ V_j(\Lambda(p \land p) \land Q \land \neg Q_j) \lor V_j(\Lambda(p \land p) \land \neg Q_j). \]

Because the consequent of \( w \) contains three conjuncts, there are three \( Q \)'s, and hence six disjuncts in the DNF and six initial members of \( \Omega \). The original wff \( w_0 \) is valid only if each member of \( \Omega \) is unsatisfiable.

Of the six disjuncts, the last three, containing the conjuncts \( \neg Q_j \), are less interesting than the first three, so we will only retain the first three in this trace. Those three wffs, derived from \( w \), all have the same initial conjuncts; so to avoid lengthy formulae we will abbreviate and call those conjuncts \( H \) (for hypotheses). \( H \) is

\[ \#(lb(v_1)\leq i) \land lb(v_1)\leq i \land \#(i\leq j) \land (i\leq j) \land \#(j< k) \land (j< k) \land \]

\[ \#(kl) \land (kl) \land \#(lm) \land (lm) \land \#(ms) \land (ms) \land \#(m< n) \land (m< n) \land \]

\[ \#(l< m) \land (l< m) \land \#(n< u) \land (n< u) \land \#(u< v) \land (u< v) \land \]

\[ \#(v< w) \land (v< w) \land \#(w< x) \land (w< x) \land \#(x< y) \land (x< y) \land \]

\[ \#(y< z) \land (y< z) \land \#(z< w) \land (z< w) \land \#(w< x) \land (w< x) \land \]

\[ \#(x< y) \land (x< y) \land \#(y< z) \land (y< z). \]

The hypothesis \( H \) is paired with three different conclusion-parts which we will call \( C_1 \), \( C_2 \) and \( C_3 \) (which are derived from \( Q_1 \), \( Q_2 \) and \( Q_3 \) respectively).

\( C_1 \) is this:

\[ \#(\text{min}\{v_1[k..m]\}) = \text{elem}(v_3[l..l]) \land \]

\[ \neg \text{min}\{v_1[k..m]\} = \text{elem}(v_3[l..l]). \]

\( C_2 \) is this:
\[ ((\text{lb}(v_2) \leq k \land k \leq m+1 \land m \leq \text{ub}(v_3) \mid k) - (\text{lb}(v_3) \leq k \land k \leq m+1 \land m \leq \text{ub}(v_3) \mid k)) + 1 \leq \{ v_1[\text{lb}(v_1) \text{..ub}(v_1)] \text{<elem}(v_2[k..k]) \} \land -((\text{lb}(v_2) \leq k \land k \leq m+1 \land m \leq \text{ub}(v_3) \mid k) - (\text{lb}(v_2) \leq k \land k \leq m+1 \land m \leq \text{ub}(v_3) \mid k)) + 1 \leq \{ v_1[\text{lb}(v_1) \text{..ub}(v_1)] \text{<elem}(v_2[k..k]) \}
\]

\[ C_3 \text{ is this:} \]
\[ ((v_1[\text{lb}(v_1) \text{..ub}(v_1)] \equiv v_3[\text{lb}(v_3) \text{..ub}(v_3)]) \land -((v_1[\text{lb}(v_1) \text{..ub}(v_1)] \equiv v_3[\text{lb}(v_3) \text{..ub}(v_3)])
\]

\[ \Omega, \text{ then, contains the following three wffs at the end of Step 2, along with others:} \]
\[ H \land C_1, \; H \land C_2, \; H \land C_3. \]

8.3 Step 3: Remove EQ\#, EQ⁻\#, ORD\#, ORD⁻\#

This step removes \( \varepsilon \)-atoms and ord-atoms which are preceded by a \( \# \) (or \(-\#\)). This has no effect on \( C_1 \) or \( C_2 \), but it does affect \( H \) and \( C_3 \).

When we apply the productions of Step 3, terms of the form \( \#(v_1=v_2) \) are changed to \text{true} (because the result of \( \# \) is never \text{ERROR}). We will delete those new occurrences of \text{true} since they each occur as arguments to \( \land \).

At the same time we will make a few other simplifications. Since the values assigned to variables are always defined, such expressions as \( \#(\text{lb}(v_1)\leq i) \) and \( \#(i\leq j) \) and \( \#(\text{lb}(v_2)\leq \text{lb}(v_2)) \) are always defined. We will convert all such expressions to \text{true} (and then delete the occurrences of \text{true}) in order to shorten the formulae. If we did not do this, the subformulae in question would remain unchanged until Step 15 when new integer variables will be substituted for \( \text{lb-} \) and \( \text{ub-} \) terms. After that they would not be altered again until after the Type Reductions are complete, after the Type Separations are complete and until the decision method for BI is invoked. At that point such subformulae would be converted \text{true}; we are just doing it early now for expository convenience. For similar reasons we will delete
conjuncts of the form $\bar{a}_i$ where $i$ is an $I$-variable.

Here, then, are the new $H$ and $C_3$. ($C_1$ and $C_2$ are unchanged in Step 3.)

$H$: 
\[
\begin{align*}
&\text{lb}(v_1)i \land isj \land jsk \land k< i \land \text{lsn} \land \text{nsub}(v_1) \land \\
&\text{lb}(v_1)i \land isn+1 \land n+1\text{sub}(v_1)+1 \land \text{ord}(v_1[i..n]) \land \text{lb}(v_1)isj \land \\
&\text{jsn} \land n\text{sub}(v_1) \land v_1[i..n]=v_2[\text{lb}(v_2)..<\text{ub}(v_2)] \land \\
&\text{uelem}(v_1[k..k])=\text{ulem}(v_2[m..m]) \land \text{ulem}(v_1[k..k])=\text{ulem}(v_2[m..m]) \land \\
&\text{lb}(v_2)=\text{lb}(v_3) \land \{\{v_2[\text{lb}(v_2)..<\text{ub}(v_2)]\}=\{v_3[\text{lb}(v_3)..<\text{ub}(v_3)]\}\} \land \\
&\{v_2[\text{lb}(v_2)..<\text{ub}(v_2)]\}=\{v_3[\text{lb}(v_3)..<\text{ub}(v_3)]\} \land \\
&\text{lb}(v_3)\leq\text{lb}(v_3) \land \text{lb}(v_3)\leq\text{ub}(v_3)+1 \land \text{ub}(v_3)\leq\text{ub}(v_3) \land \text{ord}(v_3[\text{lb}(v_3)..<\text{ub}(v_3)])
\end{align*}
\]

$C_3$: 
\[
\begin{align*}
&\neg v_1[\text{lb}(v_1)..<\text{ub}(v_1)] = v_3[\text{lb}(v_3)..<\text{ub}(v_3)]
\end{align*}
\]

8.4 Step 4: Remove $\text{EQ}^-$, $\text{ORD}^-$

In this step we remove negated occurrences of $\bar{a}_v$- and $\text{ord}$-atoms. The only such occurrence among $H$, $C_1$, $C_2$ and $C_3$ is the one in $C_3$, so we consider only it.

Step 4 is a sequence of three production systems. The first, a two-stage Markov production system, removes negated ord- and $\bar{a}_v$-atoms, but introduces quantifiers. The second and third production systems remove the quantifier(s).

The last production of the second stage of the first PS is the primary production. It introduces a new bound $I$-variable which we will call $p$. The effect of that production on $C_3$ is to produce the following:

$C_3$: 
\( (\forall v_1[\text{lb}(v_1)\ldots\text{ub}(v_1)] \land \forall v_2[\text{lb}(v_2)\ldots(v_2)] \land \text{lb}(v_1)\neq \text{lb}(v_2) \lor \text{ub}(v_1)\neq \text{ub}(v_2) \lor \\
(\exists p. \text{lb}(v_1)\leq p \land p\leq \text{ub}(v_1) \land v_1[p]\neq v_2[p]))) \land \forall v[\text{lb}(v_1)\ldots\text{ub}(v_1)] \lor \forall v_2[\text{lb}(v_2)\ldots\text{ub}(v_2)] 
\)

As this transformation has introduced occurrences of \( \forall v, o[o] \) and \( o | o | o \) which were previously removed, the first stage of the Markov production system removes them, producing this:

\text{C}_3:\n
\text{T}(\forall \text{lb}(v_1) \land \forall \text{ub}(v_1) \land \text{lb}(v_1)\leq \text{ub}(v_1) \land \text{lb}(v_1)\leq \text{ub}(v_1)+1 \land \text{ub}(v_1)\leq \text{ub}(v_1) \land \\
\forall \text{lb}(v_2) \land \forall \text{ub}(v_2) \land \text{lb}(v_2)\leq \text{ub}(v_2) \land \text{lb}(v_2)\leq \text{ub}(v_2)+1 \land \text{ub}(v_2)\leq \text{ub}(v_2) \land \\
(\text{lb}(v_1)\neq \text{lb}(v_2) \lor \text{ub}(v_1)\neq \text{ub}(v_2) \lor (\exists p. \text{lb}(v_1)\leq p \land p\leq \text{ub}(v_1) \land \\
\text{elem}(v_1[p..p])\neq \text{elem}(v_2[p..p]))) \lor \\
\text{F}(\forall \text{lb}(v_1) \land \forall \text{ub}(v_1) \land \text{lb}(v_1)\leq \text{ub}(v_1) \land \text{lb}(v_1)\leq \text{ub}(v_1)+1 \land \text{ub}(v_1) \land \\
\forall \text{lb}(v_2) \land \forall \text{ub}(v_2) \land \text{lb}(v_2)\leq \text{ub}(v_2) \land \text{lb}(v_2)\leq \text{ub}(v_2)+1 \land \text{ub}(v_2)\leq \text{ub}(v_2) \land \\
(\forall \text{lb}(v_1) \land \forall \text{ub}(v_1) \land \text{lb}(v_1)\leq \text{ub}(v_1) \land \text{lb}(v_1)\leq \text{ub}(v_1)+1 \land \text{ub}(v_1)\leq \text{ub}(v_1) \land \\
\forall \text{lb}(v_2) \land \forall \text{ub}(v_2) \land \text{lb}(v_2)\leq \text{ub}(v_2) \land \text{lb}(v_2)\leq \text{ub}(v_2)+1 \land \text{ub}(v_2)\leq \text{ub}(v_2) \land \\
\text{E}(\forall \text{lb}(v_1) \land \forall \text{ub}(v_1) \land \text{lb}(v_1)\leq \text{ub}(v_1) \land \text{lb}(v_1)\leq \text{ub}(v_1)+1 \land \text{ub}(v_1)\leq \text{ub}(v_1) \land \\
\forall \text{lb}(v_2) \land \forall \text{ub}(v_2) \land \text{lb}(v_2)\leq \text{ub}(v_2) \land \text{lb}(v_2)\leq \text{ub}(v_2)+1 \land \text{ub}(v_2)\leq \text{ub}(v_2) \land \\
\text{error})$

Though this is literally what is produced, in order to continue with our exposition we must simplify it now. We will remove such expressions as \( \forall \text{lb}(v) \) and \( \text{lb}(v)\leq \text{lb}(v) \). We will also remove identities such as \( \text{lb}(v)\leq \text{ub}(v)+1 \), and make Boolean simplifications as well. All of these simplifications would occur eventually anyway, but not as part of the Type Reduction
After applying these simplifications $C_3$ collapses to

$$lb(v_1) \neq lb(v_2) \lor ub(v_1) \neq ub(v_2) \lor (\exists p. lb(v_1) \leq p \land p < sub(v_1) \land elem(v_1[p..p]) \neq elem(v_2[p..p]))$$

No further productions from the first PS apply, so we proceed to the second PS of Step 4. This PS simply brings the quantifier $\exists p$ out to the front of the formula - not just to the front of $C_3$, but to the front of the entire formula $H \land C_3$. The last PS simply drops the quantifier.

We can thus view the wff after Step 4 as being $H \land C_3$ where $H$ is as before and $C_3$ is

$$lb(v_1) \neq lb(v_2) \lor ub(v_1) \neq ub(v_2) \lor (lb(v_1) \leq p \land p < ub(v_1) \land elem(v_1[p..p]) \neq elem(v_2[p..p]))$$

8.5 Step 5: Hypothesize all possible index order configurations

Let us review at this point exactly what the formulae look like that we are following. They are the wffs $H \land C_1$, $H \land C_2$ and $H \land C_3$. In transcribing $H$ this time we will drop the one occurrence each of the following conjuncts: $lb(v_1) \leq i$, $nsub(v_1)$, $lb(v_1) \leq j$, $jsn$, $nsub(v_1)$, $lb(v_3) \leq lb(v_3)$, $lb(v_3) \leq ub(v_3) + 1$, $ub(v_3) \leq ub(v_3)$. These are dropped because they are either repeated elsewhere in the wff or can be inferred by transitivity or are identically true. In each case it is reasonable to suppose that a simplifier working in conjunction with the Type Reduction algorithms would do the same. We apply no simplifications to $C_1$, $C_2$ or $C_3$.

Here is $H$ (after simplification):

$$lb(v_1) \leq i \land i \leq j \land j \leq k \land k \leq l \land l \leq m \land m \leq n \land n + 1 \leq ub(v_1) + 1 \land$$

$$i \leq n + 1 \land ord(v_1[i..n]) \land v_1[j..n] \leq v_2[lb(v_2)..ub(v_2)] \land$$

$$\{elem(v_1[k..k]) = elem(v_2[m..m])\} \land elem(v_1[k..k]) = elem(v_2[m..m]) \land$$

$$lb(v_2) = lb(v_3) \land \{v_2[lb(v_2)..ub(v_2)] = v_3[lb(v_3)..ub(v_3)]\} \land$$

$$\{v_2[lb(v_2)..ub(v_2)] = v_3[lb(v_3)..ub(v_3)]\} \land ord(v_3[lb(v_3)..ub(v_3)])$$

This is $C_1$:

$$\{min(\{v_1[k..m]\}) = elem(v_2[l..l])\} \land$$

$$\neg min(\{v_1[k..m]\}) = elem(v_2[l..l]).$$
This is $C_2$:

$$ u \left( (lb(v_3) \leq k \land k \leq m+1 \land m \leq sub(v_3)[k]) \right) \left( (lb(v_3) \leq k \land k \leq m+1 \land m \leq sub(v_3)[k]) \right) + 1 $$

$$ \leq \{ v_1[lb(v_1)..ub(v_1)] \}_{\text{elem}(v_2[k..k])} \land $$

$$ \left( (lb(v_3) \leq k \land k \leq m+1 \land m \leq sub(v_3)[k]) \right) \left( (lb(v_3) \leq k \land k \leq m+1 \land m \leq sub(v_3)[k]) \right) + 1 $$

$$ \leq \{ v_1[lb(v_1)..ub(v_1)] \}_{\text{elem}(v_2[k..k])} \}.$$

This is $C_3$:

$$ lb(v_1) \neq lb(v_2) \lor ub(v_1) \neq ub(v_2) \lor $$

$$ (lb(v_1) \leq p \land p \leq ub(v_1) \land \text{elem}(v_1[p..p]) \neq \text{elem}(v_2[p..p])).$$

Let us refer to the three wffs as $w_1 = H \land C_1$, $w_2 = H \land C_2$ and $w_3 = H \land C_3$. The first task in Step 1 is to identify the elements of the sets $IX(w_1, v_1)$, $IX(w_2, v_2)$ ... $IX(w_3, v_3)$. Recall that the set $IX(w_1, v_1)$ is the set of terms including $lb(v_1)$, $ub(v_1)+1$ and the simplifications of all terms occurring in the left-endpoint position of a term such as $v[i..j]$ (namely $i_0$) and the successors of the right-endpoints in such terms (namely $(j+1)_0$). Here, then, are the nine sets.

$$ IX(w_1, v_1) = \{ lb(v_1), ub(v_1)+1, i, n+1, j, k, k+1, m, m+1 \} $$

$$ IX(w_1, v_2) = \{ lb(v_2), ub(v_2)+1, m, m+1 \} $$

$$ IX(w_1, v_3) = \{ lb(v_3), ub(v_3)+1, i, i+1 \} $$

$$ IX(w_2, v_1) = \{ lb(v_1), ub(v_1)+1, i, n+1, j, k, k+1, m, m+1 \} $$

$$ IX(w_2, v_2) = \{ lb(v_2), ub(v_2)+1, m, m+1, k, k+1 \} $$

$$ IX(w_2, v_3) = \{ lb(v_3), ub(v_3)+1 \} $$

$$ IX(w_3, v_1) = \{ lb(v_1), ub(v_1)+1, i, n+1, j, k, k+1, m, m+1, p, p+1 \} $$

$$ IX(w_3, v_2) = \{ lb(v_2), ub(v_2)+1, m, m+1, p, p+1 \} $$

$$ IX(w_3, v_3) = \{ lb(v_3), ub(v_3)+1 \} $$

Notice that the terms in these sets are exactly as they appear in the formulae. This is simply
because our simplification operator, $\circ$, maps each of the terms to themselves in this case.

If we were to follow the algorithm as written, the next step would be to calculate the sets $\text{ICF}(w_1, v_1), \text{ICF}(w_1, v_2), \ldots, \text{ICF}(w_1, v_3)$ and then to break e.g. $w_1$ into many cases. The number of cases into which $w_1$ would be divided is the product of the cardinalities of $\text{ICF}(w_1, v_1), \text{ICF}(w_1, v_2)$ and $\text{ICF}(w_1, v_3)$ since the Y-parts of the formulae constructed in this step are formed by conjunctions of the elements of those three sets.

We cannot follow this procedure exactly, and in fact no computer implementation can either. Without making an accurate combinatorial calculation it appears to me that formula $w_1$ would be divided into about $10^{20}$ cases. Fortunately, it is not necessary to enumerate anywhere near that number of cases, and any implementation of our Type Reduction procedures should consider the following ways to reduce the set of cases.

Consider the following element of $\text{ICS}(w_1, v_2)$.

$$\{-\#(m+1), \text{lb}(v_2) \leq m, \text{lb}(v_2) \leq \text{ub}(v_2)+1, \# \text{sub}(v_2)+1 \}$$

which corresponds to the following formula of $\text{ICF}(w_1, v_2)$ (which is unique up to the order of the conjuncts):

$$\neg \#(m+1) \land \text{lb}(v_2) \leq m \land \text{lb}(v_2) \leq \text{ub}(v_2)+1 \land \# \text{sub}(v_2)+1.$$  

This conjunction is unsatisfiable because its first conjunct $\neg \#(m+1)$ is unsatisfiable (since $m$ is a variable and cannot be assigned an error value.) Hence, any formula to which this formula is conjoined would also be immediately unsatisfiable. Because every step remaining in the two Type-Reduction algorithms preserves this unsatisfiability, it is a waste of effort to process further any formula which can be determined early to be unsatisfiable. We therefore do not need to consider any $y \in \text{ICS}(w_1, v_2)$ containing such unsatisfiable conjuncts as $\#(m+1)$.

By similar reasoning we need never consider any $y \in \text{ICS}(w_1, v_2)$ containing the unsatisfiable conjuncts $m+1 \leq m$. And in the case of wff $w_1$ and variable $v_1$ we need not consider any $y \in \text{ICS}(w_1, v_1)$ containing e.g. the unsatisfiable pair of conjuncts $k < j$ and $j < k+1$.

In addition, there is another class of cases we need not consider. After Step 5 the wff $w_1$
will be replaced by wffs of the form

\[ Y_1 \land Y_2 \land Y_3 \land w_\downarrow \]

where \( Y_i \in \text{ICF}(w_\downarrow, \uparrow_i) \) for \( i = 1, 2, 3 \). Even though each \( Y_i \) may be satisfiable, \( Y_1 \) and \( Y_2 \) for example might contain the jointly unsatisfiable conjuncts \( lb(\uparrow_1) = m \) and \( lb(\uparrow_2) < m \) respectively. We need never consider such cases any further.

Finally, by the same reasoning as before, we need not consider any \( Y_i \in \text{ICF}(w_\downarrow, \uparrow_i) \) which contains conjuncts "obviously" inconsistent with conjuncts of \( w_\downarrow \). Since \( w_\downarrow \) contains the conjuncts

\[ lb(\uparrow_1) \leq i \land isj \land jsk \land ksl \land ism \land msn \land n_{sub}(v) \]

this is a tremendously strong constraint. The vast majority of wffs \( Y_i \in \text{ICF}(w_\downarrow, \uparrow_i) \) will be clearly inconsistent with this part of \( w_\downarrow \), and in fact only about 500 members of \( \text{ICF}(w_\downarrow, \uparrow_1) \) are consistent with it. Since only 11 members of \( \text{ICF}(w_\downarrow, \uparrow_2) \) and 11 members of \( \text{ICF}(w_\downarrow, \uparrow_3) \) are "obviously" consistent (according to the above discussed criteria) we see that \( w_\downarrow \) must be broken into at most \( 500 \times 11 \times 11 = 60500 \) cases. This is still far too many for humans (or computers) to handle practically, but it is a vast improvement over the previous estimate of \( 10^{20} \) cases. Further improvement can only be achieved by a creative reorganization of the Type Reduction algorithm between Steps 5 and 9. (I believe that such a reorganization could reduce the number of necessary cases to around ten.)

We will choose, for each of our wffs \( w_\downarrow, w_\downarrow, w_\downarrow \), one case to follow in this example. Rather than choose "random" cases, we will choose some which are in a sense "most general", i.e. which assume all index terms are defined and which do not assume any equalities between index terms (which would make certain steps easier later). We thus make the following selections to follow in this example. Each \( Y_i \) is a member of \( \text{ICF}(w_\downarrow, \uparrow_i) \) for \( 1 \leq i, j \leq 3 \).
\[ Y_{11} = \text{lb}(v_1) < i \land i < j \land j < k \land k < k+1 \land k+1 < m \land m < m+1 \land m+1 < n+1 \land n+1 < \text{ub}(v_1) + 1 \]

\[ Y_{12} = \text{lb}(v_2) < m \land m < m+1 \land m+1 < \text{ub}(v_2) + 1 \]

\[ Y_{13} = \text{lb}(v_3) < l \land l < l+1 \land l+1 < \text{ub}(v) + 1 \]

\[ Y_{21} = \text{lb}(v_1) < i \land i < j \land j < k \land k < k+1 \land k+1 < m \land m < m+1 \land m+1 < n+1 \land n+1 < \text{ub}(v) + 1 \]

\[ Y_{22} = \text{lb}(v_2) < k \land k < k+1 \land k+1 < m \land m < m+1 \land m+1 < \text{ub}(v_2) + 1 \]

\[ Y_{23} = \text{lb}(v_3) < \text{ub}(v_3) + 1 \]

\[ Y_{31} = \text{lb}(v_1) < i \land i < j \land j < k \land k < k+1 \land k+1 < m \land m < m+1 \land m+1 < n+1 \land n+1 < p \land p < p+1 \land p+1 < \text{ub}(v_1) + 1 \]

\[ Y_{32} = \text{lb}(v_2) < m \land m < m+1 \land m+1 < p \land p < p+1 \land p+1 < \text{ub}(v_3) + 1 \]

\[ Y_{33} = \text{lb}(v_3) < \text{ub}(v_3) + 1 \]

Our formulae now are

\[ Y_{11} \land Y_{12} \land Y_{13} \land H \land C_1, \]

\[ Y_{21} \land Y_{22} \land Y_{23} \land H \land C_2, \text{ and} \]

\[ Y_{31} \land Y_{32} \land Y_{33} \land H \land C_3. \]

We will call these three formulae \( w_1, w_2 \) and \( w_3 \) respectively, redefining our use of those three symbols. We further define these notations for the \( Y \)-parts of \( w_1, w_2 \) and \( w_3 \):

\[ Y_1 \triangleq Y_{11} \land Y_{12} \land Y_{13} \]

\[ Y_2 \triangleq Y_{21} \land Y_{22} \land Y_{23} \]

\[ Y_3 \triangleq Y_{31} \land Y_{32} \land Y_{33} \]

These replace earlier uses of the notations \( Y_1, Y_2 \) and \( Y_3 \).

Let us summarize now the effect of Step 5. Before Step 5 there were six wffs in \( \Omega \), and \( \Omega \) had the property that all wffs in \( \Omega \) were unsatisfiable if and only if the original \( w_0 \) was valid.
We were following the algorithm's behavior on three of those six wffs.

After Step 5, each of those wffs has been replaced in \( \Omega \) by a (possibly large) set of wffs. Each of the wffs has a Y-part constructed according to the method we have illustrated. \( \Omega \) still has the property that \( \not\models \Omega \iff \exists w_0 \).

We are following only three wffs in \( \Omega \), namely \( w_1 \), \( w_2 \) and \( w_3 \), each derived respectively from (the former) \( w_1 \), \( w_2 \) and \( w_3 \). Since \( w_0 \) is valid we expect to show eventually that our new \( w_1 \), \( w_2 \) and \( w_3 \) are each unsatisfiable. However, they are not trivially unsatisfiable. \( Y_1 \), \( Y_2 \) and \( Y_3 \) are each satisfiable, and they are also consistent respectively with the rest of \( w_1 \), \( w_2 \) and \( w_3 \), at least up to possible refutation using propositional manipulation and the general properties of \( =_1 \) and \( <_1 \) such as transitivity.

8.6 Step 6: Collapse undefined V-terms to \( E_v \)

In this step we look in each \( w \in \Omega \) for terms of the form \( v[i..j] \) which are forced by the Y-part of \( w \) to take the value \( E_v \) under any assignment satisfying \( w \). If the Y-part contains any of the conjuncts \( \neg w^o, \neg w(j+1)^o, (j+1)^o < lb(v), (j+1)^o < i^o \) or \( ub(v)+1 < i^o \), then we can replace the term \( v[i..j] \) by the term \( E_v \).

In the case of our formulae \( w_1 \), \( w_2 \) and \( w_3 \), however, this step has no effect. In Step 5 we deliberately chose Y's so that no such simplifications would occur. If we had chosen different Y's to use in this example, Step 5 would have an effect. For example, if for \( Y_{12} \) we had chosen the wff \( m < m+1 \land m+1 < lb(v) \land lb(v) < ub(v)+1 \) then in this step we would convert the term \( \text{elem}(v_2[m..m]) \) to \( \text{elem}(E_v) \).

8.7 Step 7: Remove the Constant \( E_v \)

If there were any occurrences of the constant \( E_v \) in any of our three wffs this step would remove them, but as there are not, this step makes no changes.

To follow up on the hypothetical example from last Section, if we had introduced the term
elem(E_v) into w, this step would substitute the term E_1 for elem(E_v), thus removing the occurrence of E_v.

8.8 Step 8: Remove zero-length vectors; canonicalize expressions based on \( \equiv_v \)

This step does two fairly unrelated things. The first three productions remove all occurrences of V-terms which, because of constraints in the Y-part of the formula, can only be satisfied by a value which is an empty vector. In our current example we have no such terms in w_1, w_2 or w_3. The reason for this is that we chose to follow in our example only those formulae whose Y-parts did not assume any equalities between index terms. For example, where the term v[i..j] occurs in one of our wffs, we have chosen to follow the path in which the Y-part of that wff contains the constraint i\( \leq \)j+1, meaning the vector is non-empty. As a result, the first three productions of this step have no effect on our current examples.

The fourth production, however, does have an effect. Its purpose is to make sure that the upper and lower bound index terms on either side of a vector equivalence are the same. We have only one vector equivalence in our examples, namely the atom

\[ v_1[j..n] \equiv v_2[\text{lb}(v_2), \text{ub}(v_2)] \]

in the formula \( \Pi \) (which is part of all three wffs w_1, w_2 and w_3). In this case the respective endpoint terms are not the same on both sides of the equivalence. Production (d) has the effect of replacing this equivalence by

\[ v_1[j..n] \equiv v_2[j..n] \]

and also appends to the beginning of the formula the integer equalities

\[ j=\text{lb}(v_2) \land n=\text{ub}(v_2) \] .

These equations become part of all three of our formulae now.
8.9 Step 9: Remove vector overlap

Let us review the state of the computation. We are now following three formulae:

\[ w_1 \equiv Y_1 \land j=lb(v_2) \land n=ub(v_2) \land H \land C_1 \]
\[ w_2 \equiv Y_2 \land j=lb(v_2) \land n=ub(v_2) \land H \land C_2 \]
\[ w_3 \equiv Y_3 \land j=lb(v_2) \land n=ub(v_2) \land H \land C_3 \]

These are fairly large formulae, so let me display them to remind the reader of what we are dealing with. We have rearranged the \( Y \)-parts of them formulae and eliminated duplicate \#-atoms from them (which we assume the most rudimentary of simplifiers would do).

\[ w_1 \equiv \]
\[ lb(v_1)<i \land i<j \land j<k \land k<k+1 \land k+1<m \land \]
\[ m<m+1 \land m+1<n+1 \land n+1<ub(v_1)+1 \land \]
\[ lb(v_2)<m \land m<m+1 \land m+1<ub(v_2)+1 \land \]
\[ lb(v_3)<i \land i<l+1 \land l+1<ub(v)+1 \land \]
\[ j=lb(v_2) \land n=ub(v_2) \land \]
\[ lb(v_1)<i \land i<j \land j<k \land k<k+1 \land k+1<m \land \]
\[ m<m+1 \land m+1<n+1 \land n+1<ub(v_1)+1 \land \]
\[ isn+1 \land ord(v_1[i..n]) \land v_1[j..n] \land \]
\[ \#(elem(v_1[k..m])=elem(v_2[m..m])) \land elem(v_1[k..k])=elem(v_2[m..m]) \land \]
\[ lb(v_2)=lb(v_3) \land \ni\{v_2[\{lb(v_2),ub(v_2)\}]\}=\{v_3[\{lb(v_3),ub(v_3)\}]\} \land \]
\[ \{v_2[\{lb(v_2),ub(v_2)\}]\}=\{v_3[\{lb(v_3),ub(v_3)\}]\} \land ord(v_3[lb(v_3),ub(v_3)]) \land \]
\[ \#(\min(\{v_[k..m]\})=elem(v_3[l..l])) \land \]
\[ \neg\min(\{v_[k..m]\})=elem(v_3[l..l]). \]
\[ w_2 \triangleq \]

\[
\begin{align*}
&\text{lb}(v_1) < i \land i < j \land j < k \land k < l + 1 \land l < m \land m < n + 1 \land n + 1 < \text{ub}(v) + 1 \land \\
&\text{lb}(v_2) < k \land k < l + 1 \land l < m \land m < n + 1 \land n + 1 < \text{ub}(v_2) + 1 \land \\
&\text{lb}(v_3) < \text{ub}(v_3) + 1 \\
&\land \\
&j = \text{lb}(v_2) \land n = \text{ub}(v_2) \\
&\land \\
&\text{lb}(v_1) < i \land i < j \land j < k \land k < l + 1 \land l < m \land m < n + 1 \land n + 1 < \text{ub}(v_1) + 1 \land \\
&i < n + 1 \land \text{ord}(v_1[i..n]) \land v_1[i..n] = v_2[i..n] \land \\
&\text{#}(\text{elem}(v_1[k..k]) = \text{elem}(v_2[m..m])) \land \text{elem}(v_1[k..k]) \neq \text{elem}(v_2[m..m]) \land \\
&\text{lb}(v_2) = \text{lb}(v_3) \land u(\{v_2[\text{lb}(v_2)..<\text{ub}(v_2)]\} = \{v_3[\text{lb}(v_3)..<\text{ub}(v_3)]\}) \land \\
&\{v_2[\text{lb}(v_2)..<\text{ub}(v_2)]\} = \{v_3[\text{lb}(v_3)..<\text{ub}(v_3)]\} \land \text{ord}(v_3[\text{lb}(v_3)..<\text{ub}(v_3)]) \\
&\land \\
&u(\{(\text{lb}(v_2) < k \land k \leq m + 1 \land m < \text{ub}(v_3)\}) - (\text{lb}(v_2) < k \land k \leq m + 1 \land m < \text{ub}(v_3)) + 1 \\
\leq \{v_1[\text{lb}(v_1)..<\text{ub}(v_1)]\} < \text{elem}(v_2[k..k]) \} \land \\
- (\text{lb}(v_2) < k \land k \leq m + 1 \land m \leq \text{ub}(v_3)) \cdot (\text{lb}(v_2) < k \land k \leq m + 1 \land m \leq \text{ub}(v_3)) + 1 \\
\leq \{v_1[\text{lb}(v_1)..<\text{ub}(v_1)]\} < \text{elem}(v_2[k..k]) \
\end{align*}
\]
\[ w_3 \triangleq \]

\[ \begin{align*}
\text{lb}(v_1) &< i \land i < j \land j < k \land k < k + 1 \land k + 1 < m \land m < m + 1 \land m + 1 < n + 1 \land n + 1 < p \land \\
& p < p + 1 \land p + 1 < \text{ub}(v_1) + 1 \land \\
\text{lb}(v_2) &< m \land m < m + 1 \land m + 1 < p \land p < p + 1 \land p + 1 < \text{ub}(v_2) + 1 \land \\
\text{lb}(v_3) &< \text{ub}(v_3) + 1 \\
& \land \\
& j = \text{lb}(v_2) \land n = \text{ub}(v_2) \\
& \land \\
\text{lb}(v_1) &< i \land i < j \land j < k \land k < k + 1 \land k + 1 < l \land l < l + 1 \land l + 1 < \text{ub}(v_1) + 1 \land \\
& n + 1 < l + 1 \land \text{ord}(v_1[i..n]) \land v_1[j..n] = v_2[j..n] \land \\
& \#(\text{elem}(v_1[k..k]) = \text{elem}(v_2[m..m])) \land \text{elem}(v_1[k..k]) = \text{elem}(v_2[m..m]) \land \\
\text{lb}(v_2) &< \text{lb}(v_3) \land u \left( \left\{ v_2[\text{lb}(v_2) .. \text{ub}(v_2)] \right\} = \left\{ v_3[\text{lb}(v_3) .. \text{ub}(v_3)] \right\} \right) \land \\
\left\{ v_2[\text{lb}(v_2) .. \text{ub}(v_2)] \right\} &< \left\{ v_3[\text{lb}(v_3) .. \text{ub}(v_3)] \right\} \land \text{ord}(v_3[\text{lb}(v_3) .. \text{ub}(v_3)]) \\
& \land \\
\text{lb}(v_1) &< \text{lb}(v_2) \lor \text{ub}(v_1) \neq \text{ub}(v_2) \lor \\
& \left( \text{lb}(v_1) \leq p \land p \neq \text{ub}(v_1) \land \text{elem}(v_1[p..p]) = \text{elem}(v_2[p..p]) \right). 
\end{align*} \]

To be complete in this example we should continue following all three of these formulae through the rest of the algorithm. But it should be apparent by now that we cannot. The wffs are simply growing too large to handle by hand in a document like this. So we will abandon the last two formulae and continue with only \( w_1 \), which we will rename simply \( w \). Formula \( w \) exhibits most of the features that the other wffs exhibit anyway, so little would be gained by continuing with them.

In this critical Step 9 we will transform \( w \) (i.e. \( w_1 \)) into a form in which any pair of vector terms occurring in it refer either to the same segment of the same vector, or to disjoint segments of the same vector, or to segments of different vectors. By "same" or "disjoint"
parts of a vector we mean same or disjoint according to the integer inequalities in the Y-part of the formula.

If we examine the formula \( w \) we find four vector terms referring to \( v_1 \). They are

\[ v_1[i..n], \ v_1[j..n], \ v_1[k..k] \text{ and } v_1[k..m]. \]

The index terms used in these \( V \)-terms are

\[ i, j, k, k+1, m+1, n+1. \]

In addition, the index term \( m \) gets introduced by the interaction of \( v_1 \) and \( v_2 \) as a result of the \( = \)-atom relating the two variables. Notice that we use the successors of the right-endpoint terms, and that the integer term simplification operator \( o \) has been applied to each of the terms (although it has no effect since the terms are already simplified).

From the \( Y \)-part of \( w \) (i.e. what we were calling \( Y_1 \)) we see that \( w \) is only satisfied if

\[ i<j, \ j<k, \ k<k+1, \ k+1<m, \ m<m+1, \text{ and } m+1<n+1. \]

From these inequalities we can divide \( v_1 \) into six disjoint intervals of interest, namely

\[ v_1[i..j-1], \ v_1[j..k-1], \ v_1[k..k], \ v_1[k+1..m-1], \ v_1[m..m] \text{ and } v[m+1..n] \]

which exhaust all of the parts of vector \( v_1 \) which are covered by any of the terms occurring in \( w \). The goal of Step 9, then, is to rewrite \( w \) to an equivalent form using only these last six terms. This is not really apparent in the way step 9 is written, but this is the effect.

We cannot apply the productions of Step 9 directly in this example; it would take too much space to show each step. But the net effect will be to perform the following substitutions on \( w \).
\[\{v_1[k..m]\} \mapsto\]
\[\{v_1[k..k]\} + \{v_1[k+1..m-1]\} + \{v_1[m..m]\}\]

\[v_1[j..n] \equiv v_2[j..n] \mapsto\]
\[v_1[j..k-1] \equiv v_2[j..k-1] \wedge\]
\[v_1[k..k] \equiv v_2[k..k] \wedge\]
\[v_1[k+1..m-1] \equiv v_2[k+1..m-1] \wedge\]
\[v_1[m..m] \equiv v_2[m..m] \wedge\]
\[v_1[m+1..n] \equiv v_2[m+1..n]\]

\[\text{ord}(v_1[i..n]) \mapsto\]
\[\text{ord}(v_1[i..j-1]) \wedge \max\{\{v_1[i..j-1]\}\} \leq \min\{\{v_1[i..k-1]\}\} \wedge\]
\[\text{ord}(v_1[j..k-1]) \wedge \max\{\{v_1[j..k-1]\}\} \leq \min\{\{v_1[k..k]\}\} \wedge\]
\[\text{ord}(v_1[k..k]) \wedge \max\{\{v_1[k..k]\}\} \leq \min\{\{v_1[k+1..m-1]\}\} \wedge\]
\[\text{ord}(v_1[k+1..m-1]) \wedge \max\{\{v_1[k+1..m-1]\}\} \leq \min\{\{v_1[m..m]\}\} \wedge\]
\[\text{ord}(v_1[m..m]) \wedge \max\{\{v_1[m..m]\}\} \leq \min\{\{v_1[m+1..n]\}\} \wedge\]
\[\text{ord}(v_1[m+1..n])\]

The thing to notice about these substitutions is that they are equivalence or equality substitutions (in the context of the inequalities in the index terms that we highlighted) and that they have the same set of index terms. No new index terms have been added. Furthermore, in the current context, all of the substituted terms are either identical or refer to disjoint parts of \(v_1\).

Step 9 will cause similar substitutions to be performed for the terms based on vector variables \(v_2\) and \(v_3\). Thus, we make the following summary of the effects. The terms based on \(v_2\) which occur in \(w\) are
\[ v_2[.,k+1..m], v_3[.,m+1..n] \] and \[ v_2[\text{lb}(v_2).\text{ub}(v_2)] \].

The first four terms were not in \( w \) at the beginning, but were introduced during the substitutions for the vector \( v_1 \) at the same time that the term \( v_2[j..n] \) was removed from \( w \). These \( V \)-terms are subject to the following conditions on the index terms from the \( Y \)-part of \( w \):

\[ \text{lb}(v_2)<j, j<k, k<k+1, k+1<m, m+1<n+1, n+1<\text{ub}(v_2)+1. \]

We thus break the vector \( v_2 \) into the following seven disjoint parts,

\[ v_2[\text{lb}(v_2).j-1], v_2[j..k+1..m-1], v_2[k..k+1..m], v_2[m..n], v_2[n+1..\text{ub}(v_2)] \]

and re-express \( w \) using only these seven terms based on the variable \( v_2 \). We do this by applying the rules of Step 9 which have the net effect of the following additional substitution:

\[ \{v_2[\text{lb}(v_2).\text{ub}(v_2)]\} = \]

\[ \{v_2[\text{lb}(v_2).j-1]\} + \{v_2[j..k-1]\} + \{v_2[k..k]\} + \{v_2[k+1..m-1]\} + \]

\[ \{v_2[m..m]\} + \{v_2[m+1..n]\} + \{v_2[n+1..\text{ub}(v_2)]\}. \]

Once again we see that in the context of the integer inequalities in the \( Y \)-part of the formula this substitution is an equality substitution. Furthermore, it does not increase the set of index terms, and it includes only terms representing disjoint parts of the vector \( v_2 \).

Finally, for the variable \( v_3 \) we have the following terms occurring in \( w \) (after the previous substitutions):

\[ v_3[.,1] \text{ and } v_3[\text{lb}(v_3).\text{ub}(v_3)]. \]

The index terms mentioned are

\( \text{lb}(v_3), 1, 1+1 \) and \( \text{ub}(v_3)+1 \)

which are subject to the following constraints from \( w \):

\[ \text{lb}(v_3)<1, 1<l+1, l+1<\text{ub}(v_3)+1. \]

Hence, the effect of Step 9 will be to divide \( v_3 \) into three disjoint regions,

\[ v_3[\text{lb}(v_3).l-1], v_3[l..l] \text{ and } v_3[l+1..\text{ub}(v_3)]. \]
and each term mentioning $v_3$ will be re-expressed in using only these three segments. Step 9 gives rise to the following substitutions (in addition to the previous ones):

\[
\{v_3[\text{lb}(v_3)\ldots\text{ub}(v_3)]\} \rightarrow \\
\{v_3[\text{lb}(v_3),1-1]\} + \{v_3[1,1]\} + \{v_3[1+1,\text{ub}(v_3)]\}; \text{ and}
\]

\[
\text{ord}(v_3[\text{lb}(v_3)\ldots\text{ub}(v_3)]) \rightarrow \\
\text{ord}(v_3[\text{lb}(v_3),1-1]) \land \max\left(\{v_3[\text{lb}(v_3),1-1]\}\right) \leq \min\left(\{v_3[1,1]\}\right) \land \\
\text{ord}(v_3[1,1]) \land \max\left(\{v_3[1,1]\}\right) \leq \min\left(\{v_3[1+1,\text{ub}(v_3)]\}\right) \land \\
\text{ord}(v_3[1+1,\text{ub}(v_3)])
\]

We can now display the result of these substitutions on $w$:

\[
lb(v_1)<i \land i<j \land j<k \land k+1 < k+1 \land k+1 < m \land \\
m < m+1 \land m+1 < n+1 \land n+1 < \text{ub}(v_1)+1 \land \\
lb(v_2)<m \land m < m+1 \land m+1 < \text{ub}(v_2)+1 \land \\
lb(v_3)<l \land l+1 < l+1 < \text{ub}(v_3)+1 \\
\land \\
j=\text{lb}(v_2) \land n=\text{ub}(v_2) \\
\land \\
lb(v_1)<i \land i=j \land j=k \land k+1 < m \land m \leq m \land m+1 < \text{ub}(v_1)+1 \land \\
i < n+1 \land \\
\text{ord}(v_1[i..j-1]) \land \max\left(\{v_1[i..j-1]\}\right) \leq \min\left(\{v_1[j..k-1]\}\right) \land \\
\text{ord}(v_1[j..k-1]) \land \max\left(\{v_1[j..k-1]\}\right) \leq \min\left(\{v_1[k..k]\}\right) \land \\
\text{ord}(v_1[k..k]) \land \max\left(\{v_1[k..k]\}\right) \leq \min\left(\{v_1[k+1..m-1]\}\right) \land \\
\text{ord}(v_1[k+1..m-1]) \land \max\left(\{v_1[k+1..m-1]\}\right) \leq \min\left(\{v_1[m..m]\}\right) \land \\
\text{ord}(v_1[m..m]) \land \max\left(\{v_1[m..m]\}\right) \leq \min\left(\{v_1[m+1..n]\}\right) \land
\]
ord(v₃[m+1..n]) ∧
v₃[i..k-1] = v₂[i..k-1] ∧
v₃[k..k] = v₂[k..k] ∧
v₃[k+1..m-1] = v₂[k+1..m-1] ∧
v₃[m..m] = v₂[m..m] ∧
v₃[m+1..n] = v₂[m+1..n] ∧
\#(elem(v₃[k..k]) - elem(v₂[m..m])) ∧ elem(v₃[k..k]) - elem(v₂[m..m]) ∧

lb(v₂) = lb(v₃) ∧
\#(\{v₂[lb(v₂)..j-1]\} + \{v₂[i..k-1]\} + \{v₂[k..k]\} + \{v₂[k+1..m-1]\} +
\{v₂[m..m]\} + \{v₂[m+1..n]\} + \{v₂[n+1..ub(v₂)]\} =
\{v₃[lb(v₃)..j-1]\} + \{v₃[i..l]\} + \{v₃[l+1..ub(v₃)]\}) ∧

\{v₂[lb(v₂)..j-1]\} + \{v₂[i..k-1]\} + \{v₂[k..k]\} + \{v₂[k+1..m-1]\} +
\{v₂[m..m]\} + \{v₂[m+1..n]\} + \{v₂[n+1..ub(v₂)]\} =
\{v₃[lb(v₃)..j-1]\} + \{v₃[i..l]\} + \{v₃[l+1..ub(v₃)]\}) ∧

ord(v₃[lb(v₃)..l-1]) ∧ max(\{v₃[lb(v₃)..l-1]\}) ≥ min(\{v₃[i..l]\}) ∧

ord(v₃[i..l]) ∧ max(\{v₃[i..l]\}) ≤ min(\{v₃[l+1..ub(v₃)]\}) ∧

ord(v₃[l+1..ub(v₃)]) ∧

\#(min(\{v₁[k..k]\} + \{v₁[k+1..m-1]\} + \{v₁[m..m]\} = elem(v₃[i..l])) ∧

min(\{v₁[k..k]\} + \{v₁[k+1..m-1]\} + \{v₁[m..m]\} = elem(v₃[i..l]))

If the reader will compare this formula phrase by phrase with w (w1) as given on page 360 he will find that the indicated substitutions have been made. This new w is satisfiable if
and only if the old \( w \) was satisfiable.

### 8.10 Step 10: Remove occurrences of the restriction function

In this step we remove all occurrences of the \( o[0..0] \) function that occur in the formula. We can do this only because we have guaranteed that any two vector terms are either identical, or refer to disjoint parts of the same vector, or refer to different vectors. We will be substituting new vector variables for the old ones and conjoining side conditions according to the following table.

<table>
<thead>
<tr>
<th>( V )-term ( v_i[...j-1] )</th>
<th>( \text{New } V )-var ( v_j )</th>
<th>( \text{Side-condition} ) ( lb(v_j) = i \land j-1 = ub(v_j) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v_i[j..k-1] )</td>
<td>( v_k )</td>
<td>( lb(v_k) = j \land k-1 = ub(v_k) )</td>
</tr>
<tr>
<td>( v_i[k..k] )</td>
<td>( v_k )</td>
<td>( lb(v_k) = k \land k = ub(v_k) )</td>
</tr>
<tr>
<td>( v_i[k+1..m-1] )</td>
<td>( v_m )</td>
<td>( lb(v_m) = k+1 \land m-1 = ub(v_m) )</td>
</tr>
<tr>
<td>( v_i[m..m] )</td>
<td>( v_m )</td>
<td>( lb(v_m) = m \land m = ub(v_m) )</td>
</tr>
<tr>
<td>( v_i[m+1..n] )</td>
<td>( v_n )</td>
<td>( lb(v_n) = m+1 \land n = ub(v_n) )</td>
</tr>
<tr>
<td>( v_2[lb(v_2)..j-1] )</td>
<td>( v_{10} )</td>
<td>( lb(v_{10}) = lb(v_2) \land j-1 = lb(v_{10}) )</td>
</tr>
<tr>
<td>( v_2[j..k-1] )</td>
<td>( v_{11} )</td>
<td>( lb(v_{11}) = j \land k-1 = ub(v_{11}) )</td>
</tr>
<tr>
<td>( v_2[k..k] )</td>
<td>( v_{12} )</td>
<td>( lb(v_{12}) = k \land k = ub(v_{12}) )</td>
</tr>
<tr>
<td>( v_2[k+1..m-1] )</td>
<td>( v_{13} )</td>
<td>( lb(v_{13}) = k+1 \land m-1 = ub(v_{13}) )</td>
</tr>
<tr>
<td>( v_2[m..m] )</td>
<td>( v_{14} )</td>
<td>( lb(v_{14}) = m \land m = ub(v_{14}) )</td>
</tr>
<tr>
<td>( v_2[m+1..n] )</td>
<td>( v_{15} )</td>
<td>( lb(v_{15}) = m+1 \land n = ub(v_{15}) )</td>
</tr>
<tr>
<td>( v_2[n+1..ub(v_2)] )</td>
<td>( v_{16} )</td>
<td>( lb(v_{16}) = n+1 \land ub(v_2) = ub(v_{16}) )</td>
</tr>
<tr>
<td>( v_3[lb(v_3)..l-1] )</td>
<td>( v_{17} )</td>
<td>( lb(v_{17}) = lb(v_3) \land l-1 = ub(v_{17}) )</td>
</tr>
<tr>
<td>( v_3[l..l] )</td>
<td>( v_{18} )</td>
<td>( lb(v_{18}) = l \land l = ub(v_{18}) )</td>
</tr>
<tr>
<td>( v_3[l+1..ub(v_3)] )</td>
<td>( v_{19} )</td>
<td>( lb(v_{19}) = l+1 \land ub(v_3) = ub(v_{19}) )</td>
</tr>
</tbody>
</table>
The result of making these variable substitutions and conjoining these side conditions to the
beginning of the formula \( w \) is this:

\[
\begin{align*}
\text{lb}(v_4) &= i \land j-1 = \text{ub}(v_4) \land \text{lb}(v_5) = j \land k-1 = \text{ub}(v_5) \land \\
\text{lb}(v_6) &= k \land k = \text{ub}(v_6) \land \text{lb}(v_7) = k+1 \land m-1 = \text{ub}(v_7) \land \\
\text{lb}(v_8) &= m \land m = \text{ub}(v_8) \land \text{lb}(v_9) = m+1 \land n = \text{ub}(v_9) \land \\
\text{lb}(v_{10}) &= \text{lb}(v_2) \land j-1 = \text{lb}(v_{10}) \land \text{lb}(v_{11}) = j \land k-1 = \text{ub}(v_{11}) \land \\
\text{lb}(v_{12}) &= k \land k = \text{ub}(v_{12}) \land \text{lb}(v_{13}) = k+1 \land m-1 = \text{ub}(v_{13}) \land \\
\text{lb}(v_{14}) &= m \land m = \text{ub}(v_{14}) \land \text{lb}(v_{15}) = m+1 \land n = \text{ub}(v_{15}) \land \\
\text{lb}(v_{16}) &= n+1 \land \text{ub}(v_2) = \text{ub}(v_{16}) \land \text{lb}(v_{17}) = \text{lb}(v_3) \land 1-1 = \text{ub}(v_{17}) \land \\
\text{lb}(v_{18}) &= 1 \land 1 = \text{ub}(v_{18}) \land \text{lb}(v_{19}) = l+1 \land \text{ub}(v_2) = \text{ub}(v_{19}) \land \\
\text{lb}(v_2) &= i \land i < j \land j < k \land k < k+1 \land k+1 < m \land \\
m < m+1 \land m+1 < n+1 \land n+1 < \text{ub}(v_4) + 1 \land \\
\text{lb}(v_2) &= m \land m < m+1 \land m+1 < \text{ub}(v_2) + 1 \land \\
\text{lb}(v_3) &= l \land l < l+1 \land l+1 < \text{ub}(v_3) + 1 \\
\land \\
j = \text{lb}(v_2) \land n = \text{ub}(v_2) \\
\land \\
\text{lb}(v_1) &= i \land i < j \land j < k \land k < l \land l < m \land m < n \land n + 1 < \text{ub}(v_4) + 1 \land \\
isn+1 \land \\
\text{ord}(v_4) \land \max (\{v_4\}) \leq \min (\{v_5\}) \land \\
\text{ord}(v_5) \land \max (\{v_5\}) \leq \min (\{v_6\}) \land \\
\text{ord}(v_6) \land \max (\{v_6\}) \leq \min (\{v_7\}) \land \\
\text{ord}(v_7) \land \max (\{v_7\}) \leq \min (\{v_8\}) \land \\
\text{ord}(v_8) \land \max (\{v_8\}) \leq \min (\{v_9\}) \land \\
\end{align*}
\]
\[ \text{ord}(v_9) \wedge \]
\[ v_5 \equiv v_{11} \wedge v_6 \equiv v_{12} \wedge v_7 \equiv v_{13} \wedge v_8 \equiv v_{14} \wedge v_9 \equiv v_{15} \wedge \]
\[ u(\text{elem}(v_6)) = \text{elem}(v_{14}) \wedge \text{elem}(v_6) = \text{elem}(v_{14}) \wedge \]
\[ \text{lb}(v_2) = \text{lb}(v_3) \wedge \]
\[ u(\{v_{10}\} + \{v_{11}\} + \{v_{12}\} + \{v_{13}\} + \{v_{14}\} + \{v_{15}\} + \{v_{16}\} = \]
\[ \{v_{17}\} + \{v_{18}\} + \{v_{19}\}) \wedge \]
\[ \{v_{10}\} + \{v_{11}\} + \{v_{12}\} + \{v_{13}\} + \{v_{14}\} + \{v_{15}\} + \{v_{16}\} = \]
\[ \{v_{17}\} + \{v_{18}\} + \{v_{19}\} \wedge \]
\[ \text{ord}(v_{17}) \wedge \text{max}(\{v_{17}\}) \leq \text{min}(\{v_{18}\}) \wedge \]
\[ \text{ord}(v_{18}) \wedge \text{max}(\{v_{18}\}) \leq \text{min}(\{v_{19}\}) \wedge \]
\[ \text{ord}(v_{19}) \wedge \]
\[ u(\text{min}(\{v_6\} + \{v_7\} + \{v_8\}) = \text{elem}(v_{18}) \wedge \]
\[ -\text{min}(\{v_6\} + \{v_7\} + \{v_8\}) = \text{elem}(v_{18}) \]

The crucial thing about the formula in its new form is that the only \( V \)-terms are simple \( V \)-variables. We no longer have to worry about the vector restriction function which has been plagueing us for so long. Major simplifications will come quickly in the remaining steps.

8.11 Step 11: Remove \( \equiv_V \)

Now we can remove all occurrences of the \( \equiv_V \) operator. All \( \equiv_V \)-atoms are at top level conjunctive positions, and they all assert identities between variables, so the removal is trivial. If we had any atoms of the form \( v \equiv v \) we would simply delete them (or replace them by \text{true} ), but we do not have any such occurrences. The \( \equiv_V \)-atoms that we have in our formula are each identities between two different variables, e.g. \( v_5 \equiv v_{11} \). We will simply
substitute $v_{11}$ for all occurrences of $v_5$ in $w$ and then remove the $\varepsilon_v$-atom itself, making similar changes for the other vector identities.

This procedure is so simple that we will combine it with the next step before reprinting the formula.

8.12 Step 12: Remove ord

In this step we simply remove all occurrences of ord-atoms (or, strictly speaking, we replace them with $\text{true}$). We do this because, as we prove in section 6.12.3, the formula in its current state is satisfiable if and only if the formula with its ord-atoms deleted is satisfiable. This is not ordinarily the case, but it is the case for formulae satisfying the Precondition to Step 12.

After Steps 11 and 12 the formula looks like this:

\[
\begin{align*}
\text{lb}(v_4) &= i \land j - 1 = \text{ub}(v_4) \land \text{lb}(v_5) = j \land k - 1 = \text{ub}(v_5) \land \\
\text{lb}(v_6) &= k \land k = \text{ub}(v_6) \land \text{lb}(v_7) = k + 1 \land m - 1 = \text{ub}(v_7) \land \\
\text{lb}(v_8) &= m \land m = \text{ub}(v_8) \land \text{lb}(v_9) = m + 1 \land n = \text{ub}(v_9) \land \\
\text{lb}(v_{10}) &= \text{lb}(v_2) \land j - 1 = \text{lb}(v_{10}) \land \text{lb}(v_5) = j \land k - 1 = \text{ub}(v_5) \land \\
\text{lb}(v_6) &= k \land k = \text{ub}(v_6) \land \text{lb}(v_7) = k + 1 \land m - 1 = \text{ub}(v_7) \land \\
\text{lb}(v_8) &= m \land m = \text{ub}(v_8) \land \text{lb}(v_9) = m + 1 \land n = \text{ub}(v_9) \land \\
\text{lb}(v_{10}) &= n + 1 \land \text{ub}(v_2) = \text{ub}(v_{10}) \land \text{lb}(v_{17}) = \text{lb}(v_3) + 1 = \text{ub}(v_{17}) \land \\
\text{lb}(v_{18}) &= i \land i = \text{ub}(v_{18}) \land \text{lb}(v_{19}) = i + 1 \land \text{ub}(v_3) = \text{ub}(v_{19}) \land \\
\text{lb}(v_1) &= i \land i < j \land j < k \land k < k + 1 \land k + 1 < m \land \\
&m < m + 1 \land m + 1 < n + 1 \land n + 1 < \text{ub}(v_1) + 1 \land \\
\text{lb}(v_2) &= m \land m < m + 1 \land m + 1 < \text{ub}(v_2) + 1 \land \\
\text{lb}(v_3) &= i \land i < l + 1 \land l + 1 < \text{ub}(v_3) + 1 \land \\
\end{align*}
\]
8.13 Step 13: Remove elem

In this stage we remove all occurrences of the elem function. If a term elem(v) occurs in \( w \) then we create a new \( T \)-variable \( t \) to replace all occurrences of that term. If, in addition, the term \( \{ v \} \) occurs in the formula, then we replace it by \( \{ t \} \). We also append to the front of the formula the side-condition that \( \text{lb}(v) = \text{ub}(v) \).

There are three terms involving the elem function in our formula: \( \text{elem}(v_6) \), \( \text{elem}(v_8) \), and \( \text{elem}(v_{18}) \). Each of them occurs more than once. We will create the new variables \( t_6 \), \( t_8 \) and \( t_{18} \) and make the appropriate substitutions. After making the changes the formula now looks like this.
\[ \begin{align*}
&\text{lb}(v_6) = \text{ub}(v_6) \land \text{lb}(v_9) = \text{ub}(v_9) \land \text{lb}(v_{18}) = \text{ub}(v_{18}) \land \\
&\text{lb}(v_4) = i \land j - 1 = \text{ub}(v_4) \land \text{lb}(v_5) = j \land k - 1 = \text{ub}(v_5) \land \\
&\text{lb}(v_6) = k \land k = \text{ub}(v_6) \land \text{lb}(v_7) = k + 1 \land m - 1 = \text{ub}(v_7) \land \\
&\text{lb}(v_8) = m \land m = \text{ub}(v_8) \land \text{lb}(v_9) = m + 1 \land n = \text{ub}(v_9) \land \\
&\text{lb}(v_{10}) = \text{lb}(v_2) \land j - 1 = \text{lb}(v_{10}) \land \text{lb}(v_5) = j \land k - 1 = \text{ub}(v_5) \land \\
&\text{lb}(v_6) = k \land k = \text{ub}(v_6) \land \text{lb}(v_7) = k + 1 \land m - 1 = \text{ub}(v_7) \land \\
&\text{lb}(v_8) = m \land m = \text{ub}(v_8) \land \text{lb}(v_9) = m + 1 \land n = \text{ub}(v_9) \land \\
&\text{lb}(v_{16}) = n + 1 \land \text{ub}(v_2) = \text{ub}(v_{16}) \land \text{lb}(v_{17}) = \text{lb}(v_3) \land l - 1 = \text{ub}(v_{17}) \land \\
&\text{lb}(v_1) = i \land i < j \land j < k \land k < k + 1 \land k + 1 < m \land \\
&m < m + 1 \land m + 1 < n + 1 \land n + 1 < \text{ub}(v_4) + 1 \land \\
&\text{lb}(v_2) < m \land m < m + 1 \land m + 1 < \text{ub}(v_2) + 1 \land \\
&\text{lb}(v_3) < l \land l < l + 1 \land l + 1 < \text{ub}(v_3) + 1 \land \\
&j = \text{lb}(v_2) \land n = \text{ub}(v_2) \land \\
&\text{lb}(v_1) \leq i \land i \leq j \land j \leq k \land k \leq l \leq m \land m \leq n \land n + 1 \leq \text{ub}(v_4) + 1 \land i \leq n + 1 \land \\
&\max(\{v_4\}) \leq \min(\{v_5\}) \land \max(\{v_5\}) \leq \min(\{v_6\}) \land \\
&\max(\{t_6\}) \leq \min(\{v_7\}) \land \max(\{v_7\}) \leq \min(\{t_8\}) \land \\
&\max(\{t_8\}) \leq \min(\{v_9\}) \land \\
&t_6 = t_8 \land t_6 = t_8 \land \\
&\text{lb}(v_2) = \text{ub}(v_3) \land \\
&\text{lb}(v_1) = \text{lb}(v_{10}) + \{v_5\} + \{t_6\} + \{v_7\} + \{t_8\} + \{v_9\} + \{v_{16}\} = \{v_{17}\} + \{t_{18}\} + \{v_{19}\} \land \\
&\text{lb}(v_{10}) + \{v_5\} + \{t_6\} + \{v_7\} + \{t_8\} + \{v_9\} + \{v_{16}\} = \\
\end{align*} \]
\{v_17\} + \{t_18\} + \{v_19\} \land \\
\max(\{v_17\} \land \min(\{t_18\}) \land \max(\{t_18\}) \land \min(\{v_19\}) \land \\
\mu(\min(\{t_5\} + \{v_7\} + \{t_8\}) = t_1 \land \min(\{t_5\} + \{v_7\} + \{t_8\}) = t_1)

8.14 Step 14: Remove the \{v\} function

Here we take all occurrences of terms of the form \{v\} and create new Z-variables to replace them. Whenever we replace all occurrences of \{v\} by a new variable z we must also conjoin to the formula the side conditions

\text{size}(z) = \text{ub}(v) - \text{lb}(v) + 1 \land \phi \leq z

In our formula we have the following terms

\{v_4\}, \{v_5\}, \{v_7\}, \{v_9\}, \{v_{10}\}, \{v_{16}\}, \{v_{17}\}, \text{ and } \{v_{19}\}

which will be replaced respectively by the new Z-variables

\text{\(z_4, z_5, z_7, z_9, z_{10}, z_{16}, z_{17}\) and } \text{\(z_{19}\).}

The resulting formula, with the side conditions conjoined, is this:

\text{size}(z_4) = \text{ub}(v_4) - \text{lb}(v_4) + 1 \land \phi \leq z_4 \land \\
\text{size}(z_5) = \text{ub}(v_5) - \text{lb}(v_5) + 1 \land \phi \leq z_5 \land \\
\text{size}(z_7) = \text{ub}(v_7) - \text{lb}(v_7) + 1 \land \phi \leq z_7 \land \\
\text{size}(z_9) = \text{ub}(v_9) - \text{lb}(v_9) + 1 \land \phi \leq z_9 \land \\
\text{size}(z_{10}) = \text{ub}(v_{10}) - \text{lb}(v_{10}) + 1 \land \phi \leq z_{10} \land \\
\text{size}(z_{16}) = \text{ub}(v_{16}) - \text{lb}(v_{16}) + 1 \land \phi \leq z_{16} \land \\
\text{size}(z_{17}) = \text{ub}(v_{17}) - \text{lb}(v_{17}) + 1 \land \phi \leq z_{17} \land \\
\text{size}(z_{19}) = \text{ub}(v_{19}) - \text{lb}(v_{19}) + 1 \land \phi \leq z_{19} \land \\
\text{lb}(v_6) = \text{ub}(v_6) \land \text{lb}(v_6) = \text{ub}(v_6) \land \text{lb}(v_{18}) = \text{ub}(v_{18}) \land \\
\text{lb}(v_4) = i \land j - 1 = \text{ub}(v_4) \land \text{lb}(v_5) = j \land k - 1 = \text{ub}(v_5) \land
8.15 Step 15: Remove lb and ub

In this final step we remove all occurrences of the lb and ub functions, and with them the last V-terms. A quick scan of our formula in its current state reveals that all occurrences of V-terms are as simple variables which are arguments to lb or ub.

We have 14 V-variables left: \( v_1 \ldots v_{10} \) and \( v_{16} \ldots v_{19} \). Each of the occurs both as argument to lb and to ub. Hence, Step 15 requires that we invent 28 new integer variables, which we will designate \( i_1 \ldots i_{10} \), \( i_{16} \ldots i_{19} \), and \( i_{16} \ldots i_{19} \), and substitute them systematically for the lb and ub terms. In addition we must add the side conditions

\[
i_j \leq j_1 + 1 \land i_i \leq j_2 + 1 \land i_3 \leq j_3 + 1 \land i_4 \leq j_4 + 1 \land i_5 \leq j_5 + 1 \land i_6 \leq j_6 + 1 \land i_7 \leq j_7 + 1 \land i_8 \leq j_8 + 1 \land i_9 \leq j_9 + 1 \land i_{10} \leq j_{10} + 1 \land i_{16} \leq j_{16} + 1 \land i_{17} \leq j_{17} + 1 \land i_{18} \leq j_{18} + 1 \land i_{19} \leq j_{19} + 1 \land
\]

one condition for each V-variable being replaced. After making these systematic substitutions and additions of side-conditions we arrive at the following final formula.

\[
i_1 \leq j_1 + 1 \land i_2 \leq j_2 + 1 \land i_3 \leq j_3 + 1 \land i_4 \leq j_4 + 1 \land i_5 \leq j_5 + 1 \land i_6 \leq j_6 + 1 \land i_7 \leq j_7 + 1 \land i_8 \leq j_8 + 1 \land i_9 \leq j_9 + 1 \land i_{10} \leq j_{10} + 1 \land i_{16} \leq j_{16} + 1 \land i_{17} \leq j_{17} + 1 \land i_{18} \leq j_{18} + 1 \land i_{19} \leq j_{19} + 1 \\
size(z_4) = i_4 - i_4 + 1 \land \phi \leq z_4 \land size(z_5) = j_5 - i_5 + 1 \land \phi \leq z_5 \\
size(z_7) = j_7 - i_7 + 1 \land \phi \leq z_7 \land size(z_8) = j_8 - i_8 + 1 \land \phi \leq z_8 \\
size(z_{10}) = j_{10} - i_{10} + 1 \land \phi \leq z_{10} \land size(z_{16}) = j_{16} - i_{16} + 1 \land \phi \leq z_{16} \\
size(z_{17}) = j_{17} - i_{17} + 1 \land \phi \leq z_{17} \land size(z_{19}) = j_{19} - i_{19} + 1 \land \phi \leq z_{19} \\
i_6 = i_6 \land i_8 = i_8 \land i_{18} - j_{18} \land i_4 = i_4 \land j - j_4 \land i_5 = i_5 \land k - k_5 \land i_6 = k \land k = i_6 \land k = k_1 \land m - 1 = i_7 \land i_8 = m \land m = j_8 \land i_9 = m + 1 \land n = j_9 \\
i_{10} = i_2 \land j - j_{10} \land i_5 = j \land k - k - j_5 \land i_6 = k \land k = j_6 \land i_7 = k + 1 \land n - 1 = i_7 \land i_8 = m \land m = j_8 \land i_9 = m + 1 \land n = j_9 \land i_{10} = n + 1 \land j_2 = j_{16} \land i_{17} = i_3 \land l - 1 = i_{17} \land
\[ i_{18} = 1 \land i = i_{18} \land i_{19} = 1 \land j_3 = i_{19} \land j_4 = i_{19} \land \]
\[ i_{19} < i \land i < j \land j < k \land k < l < 1 \land m < 1 \land m + 1 < n + 1 \land n + 1 < j_1 + 1 \land \]
\[ i_{10} < m \land m < n + 1 \land m + 1 < j_2 + 1 \land i_{13} < l \land l < j_3 + 1 \land \]
\[ j = i_2 \land n = i_2 \land i_4 \leq i \land i_7 \leq j \land j < k \land k \leq l \leq m \land m < n \land n + 1 < j_1 + 1 \land \]
\[ i_n + 1 \land \]
\[ \max(z_4) \leq \min(z_5) \land \max(z_5) \leq \min(t_{16}) \land \]
\[ \max(t_{16}) \leq \min(z_7) \land \max(z_7) \leq \min(t_{18}) \land \]
\[ \max(t_{18}) \leq \min(z_9) \land \]
\[ t_6 = t_8 \land t_6 = t_8 \land t_6 = i_{12} \land \]
\[ z_{10} + z_5 + \{t_6\} + z_7 + \{t_6\} + z_9 + z_{16} = z_{17} + \{t_{18}\} + z_{19} \land \]
\[ z_{10} + z_5 + \{t_6\} + z_7 + \{t_6\} + z_9 + z_{16} = z_{17} + \{t_{18}\} + z_{19} \land \]
\[ \max(z_{17}) \leq \min(t_{16}) \land \max(t_{18}) \leq \min(z_{19}) \land \]
\[ \min(t_{16}) + z_7 + \{t_6\} = t_{16} \land \min(t_{16}) + z_7 + \{t_6\} = t_{16} \]

The reader can now check that this formula contains no V-terms, and is thus a formula from the language BITZ. Obviously an enormous number of simplifications can be performed upon this formula, and any automatic theorem proving system would no doubt perform them. But we have stayed as close as possible to the letter of the algorithm as we wrote it, both to make the algorithm itself clear and to illustrate the kind of formula expansion that occurs in the course of the computation.

Let us review once more just what has been accomplished. We began by looking at one formula, \( w_0 \), which we claimed was valid. In Step 2, when we put the formula into 3-valued DNF, we split it into three formulae and followed their progress for a while.

But in Step 5 we broke each of those formulae into an extremely large number of cases. We could choose to follow the progress of only one of the many successors of each formula, but we continued until Step 9.
Finally, after Step 9, we had to abandon two of the three formulae we were working on and follow only one of them; otherwise the entire chapter would have been filled with nothing but formulae.

We applied Steps 10-15 on our remaining formula and exhibited the result.

The situation, then, is this. If the above formula is unsatisfiable, and if all of the formulae whose progress we did not follow were also unsatisfiable, then the original formula, \( w_0 \) is valid (and only then).

We must, of course, test the unsatisfiability of the final homogeneous formula. Since it is a formula of BITZ, we apply the Type Reduction BITZ-BIT from Chapter 7 to the formula \( \neg T(w) \), reducing the problem to that of deciding formulae in BIT. Then we continue with the rest of the decision procedure as described in Chapter 5.
9. Evaluation and Reflections

9.1 Introduction

In this final chapter I take the opportunity to present some thoughts on the strengths and weaknesses of the material in this thesis. Since the thesis has been a long time in gestation, I have had occasion to reflect on it at length, and I am by now much more aware of its deficiencies than of its strengths. This awareness accounts for the prevailing negative tone of the conclusions. At this writing I am still actively seeking approaches to the problems to be described, so they may be read under the heading of "Future Research".

I will present the conclusions on a chapter-by-chapter basis. The section numbers of the remaining sections correspond to the numbers of the chapters to which the conclusions apply.

9.2 Production Systems and Tree-Replacement Systems

When writing algorithms for operating on expressions in a formal language we frequently expect the algorithm to take the form of a recursion on the nesting structure of the expression. This is because most of the basic syntactic and semantic notions related to wffs are inductively or recursively defined. We might therefore jump to the conclusion that some Lisp-like language is ideally suited for describing theorem-proving algorithms.

Actually, however, that does not seem to be the case. Even preliminary experiments with basic wff-manipulating algorithms suggest first that recursion is often not the desired main control structure, and secondly that if one twists an algorithm into recursive form, the resulting program is often lengthy and complicated. For example, consider the problem of putting a propositional formula (of PC2) into Disjunctive Normal Form (DNF). If we attempt to write a recursive program for DNF, we need to be able to express the DNF of a wff \( w \) as a function of the root operator in \( w \) and of the DNFs of the one or two subformulae which are arguments to that operator.

However, if the root operator is negation (\(-\)), the DNF cannot be so expressed in any simple
way. One can get around this by defining DNF and CNF (Conjunctive Normal Form) simultaneously by mutual recursion, but this substantially complicates matters, and even then the problem is not solved. There simply is no way to define DNF recursively in a manner that is as simple, attractive and elegant as the five-production system given in Chapter 2 and here repeated.

$$\text{do}$$

$$\neg p \rightarrow p \quad \square$$

$$\neg (p \land q) \rightarrow (\neg p) \lor (\neg q) \quad \square$$

$$\neg (p \lor q) \rightarrow (\neg p) \land (\neg q) \quad \square$$

$$p \land (q \lor r) \rightarrow (p \land q) \lor (p \land r) \quad \square$$

$$(q \lor r) \land p \rightarrow (q \land p) \lor (r \land p) \quad \square$$

Most of the algorithm steps in the Type Reduction algorithms of Chapters 6 and 7 are written in the form of Production Systems of one kind or another, and the decision to do this was a key decision. It is my considered opinion that I would not have been able to conceive of algorithms so complex as these in any other formalism that I am familiar with.

The primary advantage of production systems over other styles of programming is in the use of pattern matching as a primitive. With pattern matching comes a tremendous expressive power, equivalent essentially to one level of existential quantification, as I discovered when I wrote down the proof rules for Basic Production Systems. But with this power comes a cost which I did not notice until much later.

In constructing algorithms for various parts of the Type Reductions I found it easy and tempting to add more and more rules to each production system in order to make the proof of weak correctness simple. The proof gets simpler because, as long as one uses the same loop invariant, more rules simply mean more hypotheses from which to prove the same conclusion. But what I had not anticipated was that increasing the number of rules in a PS, while it makes the proof of weak correctness easier, makes the proof of termination much
harder. In general, the more rules there are in a PS, the more nondeterministic paths there are in which the program might execute, and the stronger is the theorem which asserts that all computational paths halt.

To me, this trade-off between difficulty of weak correctness proof and difficulty of termination proof was a great surprise. I intend to investigate it further in future work. Such trade-offs are exhibited in other control structures besides PS's.

9.3 The three-valued error logic PFC3

When I was first constructing my algorithms for deciding sentences in BITZV they often got very confused because of the possibility of undefined values which some of the terms and atomic formulae could take. When a formula is exhibited which contains a partial function symbol such as integer division (/), I had always considered the "meaning" of such a partial function at one of the forbidden arguments to be a kind of "don't care" value. In such situations the usual convention was to make sure that there was a conjunction somewhere else in the formula which was true if the argument to a partial function had a forbidden value, as in the following example:

\[ i \neq 0 \land A(n/i) \]

Then one could reason loosely that "when the argument i has the forbidden value 0, the whole formula is false". But as I began to construct algorithms for actually reasoning in terms of formulae with partial function and predicate symbols in them, I found that such a vague convention would not work. The convention simply did not generalize in the environment of a large number of logical concepts. If we agree that \( A(n/i) \) is not valid (without the conjunct \( i \neq 0 \)), is it then unsatisfiable? How does the convention work in the presence of quantifiers? Do bound variables vary over the presumed "don't care" value? The convention of keeping a conjunct around which "prevents" an argument from taking a forbidden value cannot be preserved over even very simple inference rules. In fact, there was no convention I could invent which I found to be both mathematically sound and at the same time simple enough to handle in my algorithms and in the proofs of their correctness.
I finally decided that the problem is in the Predicate Calculus itself; it simply was not
designed to handle symbols for partial functions and predicates because it makes the explicit
assumption in its semantics that all functions and predicates are total. I felt I had no choice
but to develop a modified Predicate Calculus, which I subsequently named PFC3, to handle
partial functions and predicates. It must have the characteristic that every type (logical sort)
has an error value which represents the fact that a partial function or predicate has taken a
forbidden argument. This turned out to be a surprisingly difficult task because, since Boolean
is a type, there has to be an error value of type Boolean. Hence, we have a third truth
value, error, which is distinct from true or false, and thus the logic becomes three-valued.

The effort to create a three-valued error logic just to properly formalize BITZV was a
very large part of the task of this thesis (for me), but I am now convinced that it was
necessary, and that any other course would have been less satisfactory mathematically. One
piece of evidence for this is that its development yielded one very important and unexpected
dividend: I was able to use the calculus PFC3 to formalize a variant of Hoare's
weak-correctness calculus which I call the "firm-correctness" calculus. (See Chapter 3.) I
consider this calculus to have corrected a well-known slight defect in Hoare's.

However, despite the mathematical advantages of using the PFC3 logic, there are practical
disadvantages in using it. The main one is that its unfamiliarity causes people to make
frequent logical errors. It was my experience that the vast majority of algorithmic errors I
made in the course of this research--perhaps 2/3 of them--were directly related to my
forgetting some error condition which would not have arisen in the usual Predicate Calculus.
The human mind seems to have a penchant for binary choices and the law of the excluded
middle, but such conveniences must be abandoned when dealing with the partial function logic
PFC3.
9.4 The Language BITZV

The language BITZV grew during the course of its design. Once I had decided to use the method of Type Reduction it was apparent that the structure of the decision algorithms would consist of the removal of one function symbol after another until only a decidable core was left. But this strategy does not work with just any formal language. For this procedure to work the language must, at each stage, have sufficient expressive power so that all occurrences of at least one function symbol can be removed. In actual fact, this criterion strongly shapes the languages to which Type Reduction can reasonably apply.

For instance, in the course of a Type Reduction one might try to remove all occurrences of function symbol $f$. It often happens that there is no obvious way to remove $f$, but if function symbol $g$ were added to the language then all occurrences of $f$ could be removed by introducing expressions involving $g$.

So one adds $g$ to the language and starts over. At some point the problem arises of how to remove occurrences of $g$. Perhaps it is necessary to add function $h$ to the language in order to aid in the removal of $g$. The trend is to keep expanding the language until the process converges somehow to the point where all of the desired function symbols can be removed without the introduction of any new ones.

This growth of language actually happened in the course of defining BITZV. For example, my original intent was to have in the language a predicate $\text{perm}(v1,v2)$ taking two vector arguments and meaning that the vector $v1$ is a permutation of $v2$. But that predicate does not participate in many harmonious algebraic relations with the other functions that were in the language at the time. At some point I decided that it was better to introduce a new data type $M$, for multiset-of-$T$. That way the perm predicate could be removed from the language because we could express $\text{perm}(v1,v2)$ as

$$m(v1) = m(v2)$$

where the $m$-function returns the multiset of elements that occur in the vector which is its
Working with multiset equalities instead of the *perm* predicate immediately suggested other function and predicate symbols which could be added to the language more or less without cost, e.g. multiset sum (+) and multiset dominance (≤) and singleton multisets, which somehow had not occurred to me when the language contained the *perm* predicate.

Later, in the course of reasoning about multisets (developing the algorithm of Chapter 7) it dawned on me that many of the problems I was encountering were due to the fact that I could not freely move terms from one side of an equation to the other, negating them in the usual algebraic manner. (The reason was that there was no such thing as a "negative" in the domain of multisets.) Hence, I decided to enlarge the domain to include all of the necessary negatives, and I renamed the type to be *zset*-of-*T*.

The important thing about *zsets* is that they have such convenient algebraic properties. They form a linear space (a module) with respect to the operations of addition and scalar-multiplication-by-an-integer. Hence, one can manipulate *zset* equations (and inequalities) with the same kind of algebraic machinery (combining and cancelling of similar terms, solving for particular variables, etc.) that one can use for any algebra satisfying certain minimal linearity laws. The change to *zsets* from multisets again suggested adding more functions to the language, namely those which came for "free" in any linear algebra e.g. *zset* subtraction.

So, in trying to structure my language in order to apply Type Reduction to it I ended up removing one predicate, *perm*, and adding a whole new type, *Z*, with a variety of function symbols to go along with it.

A similar phenomenon was responsible for the introduction of the various max and min functions and the vector restriction function $o[0..0]$ into BITZV. My original intention had been to have a set of predicates like \(\text{LEQ}(v, i, j, t)\), which was to mean that every element in the vector \(v\) between the indices \(i\) and \(j\) was less than or equal to \(t\). But such functions did not obey strong enough algebraic laws, and reasoning about them seemed to call for the
9.4

Introduction of an ever more complicated set of functions to handle the interactions among those already added. The introduction of the max, min, and especially the $\alpha[\alpha]$ functions, simplified matters tremendously.

So the point is this: the language BITZV is the way it is because first, it had to be rich enough to express assertions about sorting and merging programs, and second, because it had to be amenable to the method of Type Reduction. In the course of "growing" the language so that Type Reduction would apply, expressive power of BITZV expanded to include within its province a large number of programs, such as binary search and Quicksort, which had not been part of the original goal.

Of course BITZV has expressive limitations which were described in Chapter 4. It may be possible to extend BITZV to include other types, such as Set-of-T or String-of T, etc., but I suspect that if too much more is included (especially Strings, with concatenation) then the theory will become undecidable and/or Type Reduction will become impossible. Perhaps it would be better to study not huge theories like BITZVSXYZ, but BITSV or BITXS, etc. (where $S, X, Y$ and $Z$ are other non-primitive types).

9.5 Type Reduction and Related Concepts

The method of Type Reduction is not really a practical method for finding decision procedures to be used in automatic theorem provers. The algorithms that are produced by Type Reduction tend to contain combinatorial explosions, and two composed Type Reductions therefore often represent a double exponential explosion. This is really to be expected because all Type Reductions have in common the fact that they take the sentences in a rich language and re-express them in a comparatively primitive sublanguage. This will always result in great expansion in the size of the formulae (on the average), and hence much greater processing time.

But although Type Reduction cannot be expected to produce optimal, or even near-optimal decision procedures for the theories it applies to, it still seems to be an important theoretical
methodology for exploring the decision problem for a new theory. It is a tool, like Quantifier Elimination, which a theorist should be familiar with when he approaches a decision problem. If he can find a decision procedure by Type Reduction he can study it to identify the combinatorial issues which are at the bottleneck. With this information he can throw away the Type Reduction algorithm and attack the problem with in a more sophisticated manner.

For example, in the Type Reduction of Chapter 6 we identified the major combinatorial bottleneck as being the number of cases into which we break the formula in Section 6.5. This number was exponential in the number of distinct index terms (carefully defined) appearing in the formula. Obviously, anyone who tries to make a practical decision procedure for BITZV will want to make this, and a similar bottleneck in Section 7.9 in the Type Reduction from his central concerns.
I. Index of Notation, abbreviation, terminology

I.1 Notation Introduced in Chapter 2

PS production system

TRS tree-replacement system, special kind of PS specialized for handling expression replacement

BPS, BTRS basic production system, basic tree-replacement system; systems in which the productions are unordered and there is nondeterministic choice as to which of several eligible productions is to execute.

MPS, MTRS Markov production systems, Markov tree-replacement systems; systems in which there is a partial ordering on the productions, where lesser productions in the ordering have priority over greater ones, and there is nondeterministic choice between productions incomparable in the ordering.

pattern ⇒ action a production from a PS

pattern ⇒ replacement a production from a TRS

dol(...) od delimits a BPS or BTRS

dol[...] od delimits an MPS or MTRS

Δp pattern p is satisfiable; if p has pattern variables v₁,...,vₙ and non-pattern variables u₁,...,uₘ, then Δp(u₁,...,uₘ, v₁,...,vₙ) ⇒ ∃v₁,...,∃vₙp(u₁,...,uₘ, v₁,...,vₙ)

DNF, CNF Disjunctive Normal Form, Conjunctive Normal Form

α pαw means that some tree, term or formula of the form "p" occurs somewhere as a subtree, subterm or subformula of w.

REPL(w, P₁ ⇒ P₂) Subtree or subexpression replacement; means the result of replacing one (nondeterministically chosen) occurrence of pattern P₁ in w by the corresponding tree or expression P₂. Notation can only be used when it is known that P₁αw.

assertion₁{program} assertion₂ Hoare-style weak correctness formula; in Chapters 3 and 4 the same notation is used for "firm" correctness.

and, or Conjunction and disjunction in the metalanguage, as distinct from ∧, ∨ which are object language symbols.

⊨ Truth or validity. The assertion ⊨w means that w is valid, either universally or in some particular model (depending on context).
1.2 Notation Introduced in Chapter 3

PC2  Any of the usual formulations of the (two-valued) propositional calculus.

FC2  Two-valued functional calculus; any of the usual formulations of the standard predicate calculus. We call it the functional calculus here to emphasize that predicates are Boolean-valued functions and to distinguish it from logics where symbols represent partial rather than total functions.

PC3  The three-valued propositional calculus introduced here; extension of PC2.

PFC3 The three-valued partial functional calculus introduced here; extension of FC2.

LPC2, LPC3, LFC2, LPFC3 The languages for PC2, PC3, FC2, PFC3.

true, false, error

The three truth values in PC3 and PFC3; also, constant symbols denoting those three values.

T, F, E

unary truth functions; T(p) means "p is true", F(p) means "p is false"; E(p) means "p is error".

# In the context of PC3 it is the "is defined" truth function, where #p = (T(p) v F(p)). In the context of PFC3, it is one of the functions #X for some type X; #X(x), for an expression x, means #X(x), i.e. "x is not the error value E_X for type X."

negation, either two- or three-valued, depending on context.

\( \land, \lor \) ordinary "and" and "or" in PC2 or FC2; "strong and" and "strong or" in PC3 and PFC3. "Strong and" is true iff both arguments are true; false iff either or both arguments are false; error otherwise. Symmetric definition for "strong or".

\( \text{wand}, \text{wor} \) Defined only in PC3, PFC3; mean "weak and" and "weak or". "Weak and" is true iff both arguments are true; false iff either or both arguments are error; and false otherwise. Symmetric definition for "weak or".

\( \text{cand}, \text{cor} \) Defined in PC3, PFC3; mean "conditional and", and "conditional or". "Conditional and" is true iff both arguments are true; false iff the left argument is false, or the left argument is true and the right argument is false; error otherwise. Symmetric definition for "conditional or".

\( = \) In FC2 this is the usual equality.

In PC3 it is "weak equivalence", i.e. has the value error if either argument is error; otherwise it is true if the two arguments have the same truth value and false if they have different truth values.

In PFC3 this is the generic weak equality; it represents \( =_X \), where the
type \( X \) is the type of the arguments. It has the value \texttt{error} if either argument is \( E_x \); otherwise it is \texttt{true} if the arguments have the same value, and \texttt{false} if they do not.

\[ X \]

In PC2 this is the usual equivalence connective.

In PC3 this is "three-valued equivalence". It takes the value \texttt{true} if its arguments have the same truth value, and \texttt{false} if they do not. It never takes the value \texttt{error}.

In PFC3 this is the generic strong equality operator; it represents \( \equiv_x \) where \( X \) is the type of its arguments. It has the value \texttt{true} if its arguments have the same value (including \texttt{error} values) and \texttt{false} if they do not. It never takes the value \texttt{error}.

\[ \rightarrow \]

In PC2 this is the usual material implication connective. In PC3 and PFC3 it is a kind of material implication defined by \((p \rightarrow q) \equiv \neg T(p) \lor T(q)\)

\((\circ | \circ | \circ | \circ)\)

In PC3 this is the three-way conditional connective; \((p \mid q_1 \mid q_2 \mid q_3)\) has the value of \(q_1\) if \(p\) is \texttt{true}; it has the value \(q_2\) if \(p\) is \texttt{false}; and it has the value \(q_3\) if \(p\) is \texttt{error}.

In PFC3 it is the generic three-way conditional function. The term \((b \mid x_1 \mid x_2 \mid x_3)\) takes the value of \(x_1\) when \(b\) is \texttt{true}, the value of \(x_2\) when \(b\) is \texttt{false}, and the value of \(x_3\) if \(b\) is \texttt{error}.

\((\circ | \circ | \circ)\)

In PC3, the two-way conditional connective. \((p \mid q \mid r) \equiv (p \mid q \mid r \mid \texttt{error})\)

In PFC3, the generic two-way conditional function \((b \mid x_1 \mid x_2) \equiv_x\)

\((b \mid x_1 \mid x_2 \mid E_x)\)

\((\circ | \circ)\)

In PC3, the one-way conditional connective \((p \mid q) \equiv (p \mid q \mid \texttt{error} \mid \texttt{error})\)

In PFC3, the generic one-way conditional function \((p \mid x) \equiv_x\)

\((p \mid x \mid E_x \mid E_x)\)

\[ = \]

Validity; In Chapter 3 this generally means universal validity, in PC2, PC3, FC2 or PFC3, depending on context.

In Chapter 4 and beyond it means validity in the model BITZV (or whatever model is currently under discussion).

\( F_2, F_3 \)

Universally valid in two-valued logic (PC2, FC2) or three-valued logic (PC3, PFC3) respectively.

\[ \equiv \]

Satisfiability; In Chapter 3 it generally means satisfiable in some model ("existential" satisfiability).

In Chapter 4 and beyond it means satisfiable in some standard interpretation of BITZV.

The usage \( \equiv_w \) means that assignment \( \varphi \) satisfies wff \( w \), a statement
usually denoted $\phi$-w in most notations due to the lack of the $\exists$-symbol or something similar.

$\lambda x.p$ Ordinary lambda-notation; $\lambda x.f(x,y)$ is that function of one argument $x$, whose value is computed by evaluating $f(x,y)$. Here, $y$ is a free variable.

$\square$ End of definition, theorem, lemma, proof, etc.

$V\phi$ The valuation (on terms or wffs of the language under discussion) which is induced by assignment $\phi$ of values to variables.

$E_x$ The error constant symbol for type $X$. Its interpretation must be the zeroary constant function returning the value $e_x$.

$e_x$ The error value for type $X$. In contrast to $E_x$, this is a value, not an object language symbol. We will frequently write $E_x$ instead of $e_x$, however.

$E_B$ Alternate notation for error, the error constant for type Boolean.
I.3 Notation Introduced in Chapter 4

BITZV The formal language defined in Chapter 4, usually restricted to the quantifier-free part. Also, used to represent any standard interpretation or model for that language.

BITZ The subset of BITZV which contains no terms of Type-V.

BIT The subset of BITZV and BIT containing no terms of Types Z or V; in other words, contains only terms of Types B, I or T.

BI BITZV restricted to terms of Types B and I. (This is the unquantified, three-valued language or arithmetic.)

BT BITZV restricted to terms of Types B and T; (This is the unquantified, three-valued language of total order.)

+ Either + for addition of integers, or +z for addition of zsets.

- Either - or neg for subtraction or negation of integers, or -z or negz for subtraction or negation of zsets.

* Either * for integer multiplication, or *z for scalar multiplication of zsets by integers.

/ integer division

Z The integers (positive, zero, and negative.)

ϕ, ϕz The empty set; the empty zset.

{ο}v The function which maps a vector into the zset of elements occurring in the vector.

{ο}T The function which maps a Type-T element into the singleton zset containing one copy of that element and zero copies of all other elements.

{} Ambiguous notation for {ο}v, {ο}T, “set-of” notation or a Hoare correctness assertion.
1.4 Notation Introduced in Chapter 6

\( w_0 \)  
The original unquantified wff of BITZV to which we are applying Type Reduction in order to decide its validity.

\( \Omega \)  
A finite set of wffs of BITZV. We maintain the invariant \( \exists w_0 \iff \exists \Omega \).

\( \equiv \)  
In this chapter it means valid in all standard interpretations of BITZV.

\( \not\equiv \)  
Unsatisfiable in any standard interpretation of BITZV. When applied to a set of wffs, as in \( \not\equiv \Omega \), it means each wff in \( \Omega \) is individually unsatisfiable.

\( \iff \)  
"If and only if"; used only in metalanguage; not a symbol of BITZV.

\( \leftarrow \)  
Variable assignment; the variable assigned-to may contain a wff or a set of wffs, but this symbol always means ordinary assignment.

\( \alpha \)  
\( p \leftarrow w \) means that some tree, term or formula of the form "p" occurs somewhere as a subtree, subterm or subformula of \( w \).

REPLALL(\( w, p_1 \rightarrow p_2 \))  
The result of replacing all occurrences of the pattern \( p_1 \) in \( w \) by the corresponding tree or expression \( p_2 \). This notation can only be used when all occurrences of \( p_1 \) in \( w \) are disjoint (as subexpressions or subtrees.)

SUBST(\( w, v \rightarrow e \))  
The result of substituting the expression \( e \) for all free occurrences of \( v \) in \( w \). This notation can only be used if \( v \) is a variable.

\( \underline{v} \)  
An underlined variable is a pattern variable.

\( v' \)  
When a variable appears with a prime, it indicates that the variable is now, distinct from all variables that have occurred previously in any BITZV expression participating in the computation. If \( v' \) occurs several times in the same expression, it indicates several occurrences of the same new variable.

BITZV with elem  
The language BITZV extended to include the additional function symbol \( \text{elem} \). \( \text{elem} \) is not part of BITZV proper.

EQ  
A (possibly empty) conjunction of expressions of the form \( v_1 = v_2 \), where \( v_1 \) and \( v_2 \) are \( V \)-terms.

EQ\(^\neg\)  
A (possibly empty) conjunction of expressions of the form \( \neg v_1 = v_2 \), where \( v_1 \) and \( v_2 \) are \( V \)-terms.

EQ\(^\#\)  
A (possibly empty) conjunction of expressions of the form \( \#(v_1 = v_2) \), where \( v_1 \) and \( v_2 \) are \( V \)-terms.

EQ\(^\neg\#\)  
A (possibly empty) conjunction of expressions of the form \( \neg \#(v_1 = v_2) \),
where \( v_1 \) and \( v_2 \) are V-terms.

**ORD**

A (possibly empty) conjunction of expressions of the form \( \text{ord}(v) \), where \( v \) is a V-term.

**ORD**

Similar to ORD, but with conjuncts of the form \( \neg \text{ord}(v) \).

**ORD**

Similar to ORD, but with conjuncts of the form \( \#\text{ord}(v) \).

**ORD**

Similar to ORD, but with conjuncts of the form \( \neg\#\text{ord}(v) \).

**\( U \)**

As used in Chapter 6 it refers to part of a wff \( w \) not containing any occurrences of \( \equiv \) or \( \lor \).

**\( i^o \)**

The simplification of the integer term \( i \). It stands for any deterministic function such that \( i^o \equiv i \) for all I-terms \( i \), and such that \( (i^o+1)^o \equiv (i+1)^o \), \( (i^o-1)^o \equiv (i-1)^o \), \( ((i+1)-1)^o \equiv i^o \) and \( ((i-1)+1)^o \equiv i^o \), for all terms \( i \).

**IX(\( w,v \))**

The index set for wff \( w \) and variable \( v \); if \( v \not\in w \) then \( \text{IX}(w,v) \) is empty; otherwise the terms \( \text{lb}(v) \in \text{IX}(w,v) \), \( \text{ub}(v) \in \text{IX}(w,v) \), and for all terms \( v[i_1..i_2] \) occurring in \( w \), the terms \( i_1^o \) and \( (i_2+1)^o \) are in \( \text{IX}(w,v) \).

This definition only applies to wffs satisfying the precondition for Step 5.

**IX(\( w \))**

The union for all \( v \in w \) of \( \text{IX}(w,v) \).

**ICS(\( w,v \))**

The set of all index constraint sets for wff \( w \) and \( v \)-variable \( v \). Each \( Y \in \text{ICS}(w,v) \) is a set of wffs involving \( \equiv_1 \), \( =_1 \) and \( <_1 \) and the terms in \( \text{IX}(w,v) \) such that if every element of \( Y \) is \( \text{true} \), the truth value of any propositional combination of wffs of \( Y \) is computable. If \( \text{IX}(w,v) \) is empty, then \( \text{ICS}(w,v) \) is \( \{\text{true}\} \).

**ICU(\( w,v \))**

The index constraint universe for \( w,v \), i.e., the set of all terms of the form \( \neg i, i=j \) or \( i<j \), for \( i,j \in \text{IX}(w,v) \). Each \( \gamma \in \text{ICS}(w,v) \) is a subset of \( \text{ICU}(w,v) \).

**A(\( w,v \))**

The set of all atoms (Boolean terms) constructable from the terms in \( \text{IX}(w,v) \) and the operators \( \equiv_1, =_1, <_1 \).

**\( \psi(n) \)**

A combinatorial function bounding the size of \( \text{ICS}(w,v) \) when \( \text{IX}(w,v) \) has size \( n \).

**L_{IX}(\( w,v \))**

The language of all unquantified formula constructable from the atoms in \( \text{A}(w,v) \) and from the three-valued propositional connectives.

**Y_{Tu}**

For \( \gamma \in \text{ICS}(w,v) \) and \( u \in \text{L}_{IX}(w,v) \) this relation means for all \( \phi \), if \( \phi \) satisfies all elements of \( Y \), then \( \phi \equiv u \). This is a recursive relation.

**ICF(\( w,v \))**

A set of formulae similar to the set of sets \( \text{ICS}(w,v) \). Each wff \( \gamma \in \text{ICF}(w,v) \) is the conjunction of the elements of some \( \gamma \in \text{ICS}(w,v) \), and there is a one-to-one correspondence between \( \text{ICF}(w,v) \) and \( \text{ICS}(w,v) \).

**Y_{Tu}**

Similar to \( Y_{Tu} \), but for \( \gamma \in \text{ICF}(w,v) \) instead of \( \gamma \in \text{ICS}(w,v) \).

**Y_{w}**

The \( Y \)-part of a wff \( w \) is a conjunction \( \gamma_1 \wedge \ldots \wedge \gamma_k \), where \( \gamma_i \in \text{ICF}(w,v) \)
and \( v_1 \ldots v_k \) are the V-variables occurring as arguments to \( \circ[\circ..\circ] \) in \( w \). This notation is only used after Step 5.
I.5 Notation Introduced in Chapter 7

**REPLALL**(_w_, _p_1 := _p_2, _p_3 := _p_4)

The result of replacing all occurrences of pattern _p_1 in _w_ by _p_2 and also all occurrences of pattern _p_3 in _w_ by _p_4. This notation can only be used when all occurrences of _p_1 and _p_3 are disjoint in _w_.

**TX(_w_)**

The set of all T-terms used as "indices" into zset terms in formula _w_. Terms of the form min( _z_ ) and max( _z_ ) are in TX(_w_) if they occur in _w_; also a term _t_ is in TX(_w_) if \{ _t_ \} or a term of the form _z_ < _t_ > occur in _w_. This definition applies only to wffs satisfying the Precondition to Step 9 in Chapter 7.

**TCS(_w_)**

The T-index Constraint Set for wff _w_. This is the set of formulae over the terms in TX(_w_) which satisfy Definition 2 in Section 7.9.

**TCU(_w_)**

The T-index Constraint Universe for _w_. This is the set of all terms of the form _-t_ = _t_ 1 or _t_ 1 = _t_ 2 for _t_ 1 and _t_ 2 in TX(_w_).

**A(_w_)**

The set of all Boolean terms (atomic formulae) constructable from the terms in TX(_w_).

**LTX(_w_)**

The set of all unquantified formulae constructable from the atoms in A(_w_) and the three-valued propositional connectives.

**TCF(_w_)**

The set of all formulae (unique up to commutativity and associativity of ∧) which are conjunctions of all the elements of some Y in TCS(_w_)

**Y_w**

The Y-part of _w_, i.e. that part of _w_ which was appended to _w_ from the set TCF(_w_) in Step 9.

**Yt_u**

For a Y in TCF(_w_) and a _u_ in LTX(_w_) we define Yt_u to mean "for all assignments _φ_, if _φ_ satisfies Y then _φ_ satisfies _w_".

**PNF**

Polynomial Normal Form; a zset equality or inequality atom is in PNF if it satisfies the criteria given in Definition 25, section 7.13.

**DISTINCT(_t_1..._t_n_)**

an abbreviation for the formula _t_ 1 ≠ _t_ 2 ∧ _t_ 1 ≠ _t_ 3 ∧ ... ∧ _t_ n-1 ≠ _t_ n.

**z_1 ~ z_2**

Represents either _z_ 1 = _z_ 2 or _z_ 1 ≠ _z_ 2.
II. Truth Tables for Three-valued Logic (PFC3)

1-ary:

<table>
<thead>
<tr>
<th></th>
<th>T</th>
<th>F</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
</tr>
</tbody>
</table>

2-ary:

<table>
<thead>
<tr>
<th></th>
<th>T</th>
<th>F</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>T</th>
<th>F</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>T</th>
<th>F</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>T</th>
<th>F</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
</tr>
</tbody>
</table>

4-ary: \[ (p | q_1 | q_2 | q_3) \equiv ((T(p) \land q_1) \lor (F(p) \land q_2) \lor (E(p) \land q_3)) \]

3-ary: \[ (p | q_1 | q_2) \equiv (p | q_1 | q_2) \equiv (p | q_1 | q_2 | \text{error}) \]

2-ary: \[ (p | q) \equiv (p | q | \text{error} | \text{error}) \]
III. The Language BITZV

III.1 Functions returning Type-B

<table>
<thead>
<tr>
<th>Function &amp; Args</th>
<th>Defined when...</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>Truehood truth value</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>Falsehood truth value</td>
</tr>
<tr>
<td>error or E_b</td>
<td>true</td>
<td>Errorhood truth value</td>
</tr>
<tr>
<td>T(b)</td>
<td>true</td>
<td>b ? true</td>
</tr>
<tr>
<td>F(b)</td>
<td>true</td>
<td>b ? false</td>
</tr>
<tr>
<td>E(b)</td>
<td>true</td>
<td>b ? error</td>
</tr>
<tr>
<td>(b_1</td>
<td>b_2</td>
<td>b_3</td>
</tr>
<tr>
<td>(b_1</td>
<td>b_2</td>
<td>b_3</td>
</tr>
<tr>
<td>(b_1</td>
<td>b_2</td>
<td>b_3</td>
</tr>
<tr>
<td>b_1 ∧ b_2</td>
<td>true</td>
<td>strong ∧ function</td>
</tr>
<tr>
<td>b_1 v b_2</td>
<td>true</td>
<td>strong OR function</td>
</tr>
<tr>
<td>b_1 vand b_2</td>
<td>true</td>
<td>weak ∧ function</td>
</tr>
<tr>
<td>b_1 xor b_2</td>
<td>true</td>
<td>weak OR function</td>
</tr>
<tr>
<td>b_1 cand b_2</td>
<td>true</td>
<td>conditional ∧ function</td>
</tr>
<tr>
<td>b_1 cut b_2</td>
<td>true</td>
<td>conditional OR function</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>NOT function</td>
</tr>
<tr>
<td>∀b_1 (b_1 &gt; b_2)</td>
<td>true</td>
<td>strong : substitutability</td>
</tr>
<tr>
<td>∀b_1 (b_1 ≡ b_2)</td>
<td>true</td>
<td>strong : equality</td>
</tr>
<tr>
<td>i_1 ≡ i_2</td>
<td>true</td>
<td>strong equality</td>
</tr>
<tr>
<td>t_1 ≡ t_2</td>
<td>true</td>
<td>strong equality</td>
</tr>
<tr>
<td>z_1 ≡ z_2</td>
<td>true</td>
<td>strong equality</td>
</tr>
<tr>
<td>v_1 ≡ v_2</td>
<td>true</td>
<td>strong equality</td>
</tr>
<tr>
<td>z(b)</td>
<td>true</td>
<td>b is defined, -b ? error</td>
</tr>
<tr>
<td>#_1(1)</td>
<td>true</td>
<td>i is defined : -i ? E_i</td>
</tr>
<tr>
<td>#_2(1)</td>
<td>true</td>
<td>t is defined : -t ? E_t</td>
</tr>
<tr>
<td>#_1(z)</td>
<td>true</td>
<td>z is defined : -z ? E_z</td>
</tr>
<tr>
<td>#_2(v)</td>
<td>true</td>
<td>v is defined : -v ? E_v</td>
</tr>
<tr>
<td>b_1 ≡ b_2</td>
<td>true</td>
<td>weak equality</td>
</tr>
<tr>
<td>i_1 ≡ i_2</td>
<td>true</td>
<td>weak equality</td>
</tr>
<tr>
<td>t_1 ≡ t_2</td>
<td>true</td>
<td>weak equality</td>
</tr>
<tr>
<td>z_1 ≡ z_2</td>
<td>true</td>
<td>weak equality</td>
</tr>
<tr>
<td>v_1 ≡ v_2</td>
<td>true</td>
<td>weak equality</td>
</tr>
<tr>
<td>z_1 &lt; z_2</td>
<td>true</td>
<td>standard integer order</td>
</tr>
<tr>
<td>ord(z)</td>
<td>true</td>
<td>type T order</td>
</tr>
<tr>
<td>z_1 &lt; z_2</td>
<td>true</td>
<td>z_1 dominates z_2, i.e., (∀i, z_1 &lt; i ≤ z_2 &lt; i+1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>vector v is ordered, i.e., v_1, (lub(v)) ≤ v ≤ (lub(v)[i+1])</td>
</tr>
</tbody>
</table>

NOTES:

1. Type subscripts have been dropped for E, ≡, =, (b_1 | b_2), (b_1 | b_2 | b_3 | b_4) except in left hand column.

2. The word "weak" means that if any argument is E, the result is E.

3. Bound variables range over all non-error values.
### III.2 Functions returning Type-I

<table>
<thead>
<tr>
<th>Function &amp; Args</th>
<th>Defined when...</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E_1 )</td>
<td>false</td>
<td>Integer error value</td>
</tr>
<tr>
<td>((b \mid i_1))</td>
<td>((b \mid i_1) \mid \text{false} \mid \text{false})</td>
<td>(\text{if } T(b) \text{ then } i_1; \text{ else } E_1)</td>
</tr>
<tr>
<td>((b \mid i_1 \mid i_2))</td>
<td>((b \mid i_1 \mid i_2 \mid \text{false}))</td>
<td>(\text{if } T(b) \text{ then } i_1; \text{ if } F(b) \text{ then } i_2; \text{ else } E_1)</td>
</tr>
<tr>
<td>((b \mid i_1 \mid i_2 \mid i_3))</td>
<td>((b \mid i_1 \mid i_2 \mid i_3))</td>
<td>(\text{if } T(b) \text{ then } i_1; \text{ if } F(b) \text{ then } i_2; \text{ if } T(b) \text{ then } i_3)</td>
</tr>
<tr>
<td>0, 1, 2...</td>
<td>int</td>
<td>integer constants</td>
</tr>
<tr>
<td>neg(i) (or -i)</td>
<td>#i</td>
<td>negation</td>
</tr>
<tr>
<td>#i + #i_2</td>
<td>#i + #i_2</td>
<td>addition</td>
</tr>
<tr>
<td>#i - #i_2</td>
<td>#i - #i_2</td>
<td>subtraction</td>
</tr>
<tr>
<td>#i * #i_2</td>
<td>#i * #i_2</td>
<td>multiplication</td>
</tr>
<tr>
<td>#i_1 / #i_2</td>
<td>#i_1 / #i_2 \land #i_2 \neq 0</td>
<td>division, truncation toward zero</td>
</tr>
<tr>
<td>#z \land #1</td>
<td>#z \land #1</td>
<td>number of occurrences of 1 in #z</td>
</tr>
<tr>
<td>size(z)</td>
<td>#z</td>
<td>number of elements in #z, i.e. (\sum z &lt; 1)</td>
</tr>
<tr>
<td>ub(v)</td>
<td>#v</td>
<td>upper bound for index into #v</td>
</tr>
<tr>
<td>lb(v)</td>
<td>#v</td>
<td>lower bound for index into #v</td>
</tr>
<tr>
<td>len(v)</td>
<td>#v</td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**

1. Type subscripts have been dropped for \(E, \#, \#\), and the conditional functions except in left-hand column.

2. Bound variables range over all non-error values.

3. Variables \(b, h_1\) etc. are Boolean; the \(i\)'s are Integer; the \(t\)'s are Type \(T\); the \(z\)'s are Type \(Z\); and the \(v\)'s are Type \(V\).
### III.3 Functions returning Type-T

<table>
<thead>
<tr>
<th>Function &amp; Args</th>
<th>Defined when...</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ E_T ]</td>
<td>( (b \mid t_1, t_2) )</td>
<td>Type T error value</td>
</tr>
<tr>
<td></td>
<td>( (b \mid t_1, t_2) )</td>
<td>( \text{if } T(b) \text{ then } t_1; \text{ else } E_T )</td>
</tr>
<tr>
<td></td>
<td>( (b \mid t_1, t_2) )</td>
<td>( \text{if } T(b) \text{ then } t_1; \text{ if } F(b) \text{ then } t_2; \text{ else } E_T )</td>
</tr>
<tr>
<td>( v[i] )</td>
<td>( (#v \times #i) \land (ib(v) \leq i \land i \leq ub(v)) )</td>
<td>( \text{Array access; indexing} )</td>
</tr>
<tr>
<td>( \text{elem}(v) )</td>
<td>( #v \land lb(v) = ub(v) )</td>
<td>The (unique) element in a singleton vector</td>
</tr>
<tr>
<td>( \text{min}_T(t_1, t_2) )</td>
<td>( #t_1 \land #t_2 )</td>
<td>min ( t_1, t_2 )</td>
</tr>
<tr>
<td>( \text{max}_T(t_1, t_2) )</td>
<td>( #t_1 \land #t_2 )</td>
<td>max ( t_1, t_2 )</td>
</tr>
<tr>
<td>( \text{min}_Z(z) )</td>
<td>( #z \land \langle z, 0 \rangle )</td>
<td>min (</td>
</tr>
<tr>
<td>( \text{max}_Z(z) )</td>
<td>( #z \land \langle z, 0 \rangle )</td>
<td>max (</td>
</tr>
<tr>
<td>( \text{min}_V(v) )</td>
<td>( #v \land lb(v) \leq ub(v) )</td>
<td>min ( v[i] )</td>
</tr>
<tr>
<td>( \text{max}_V(v) )</td>
<td>( #v \land lb(v) \leq ub(v) )</td>
<td>max ( v[i] )</td>
</tr>
</tbody>
</table>

**NOTES:**

1. Type subscripts have been deleted except for the left-hand column.

2. Bound variables range over the non-error values.

3. Variables \( b, b_i \), etc., are Boolean; \( i \)'s are Integer; \( t \)'s are Type \( T \); \( z \)'s are Type \( Z \); and \( v \)'s are Type \( V \).

4. The function ELEM is not properly part of the language BITZV; it is included in this table because it is introduced transiently during the Type Reduciton from BITZV to BITZ.
### III.4 Functions returning Type-Z

<table>
<thead>
<tr>
<th>Function &amp; Argn.</th>
<th>Defined when...</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>E₂</td>
<td>z₁₂</td>
<td>Type Z error value</td>
</tr>
<tr>
<td>(b</td>
<td>z₁</td>
<td>z₂)₂</td>
</tr>
<tr>
<td>(b</td>
<td>z₁</td>
<td>z₂</td>
</tr>
<tr>
<td>(b</td>
<td>z₁</td>
<td>z₂</td>
</tr>
<tr>
<td>φ</td>
<td>true</td>
<td>empty zset; φ*1 = 0</td>
</tr>
<tr>
<td>-z</td>
<td>#</td>
<td>zset negation</td>
</tr>
<tr>
<td>z₁ + z₂</td>
<td>#z₁ + #z₂</td>
<td>zset addition</td>
</tr>
<tr>
<td>z₁ - z₂</td>
<td>#z₁ - #z₂</td>
<td>zset subtraction</td>
</tr>
<tr>
<td>i₁</td>
<td>#</td>
<td>singleton zset; contains only 1</td>
</tr>
<tr>
<td>i₂</td>
<td>#</td>
<td>zset of elements in vector v</td>
</tr>
<tr>
<td>i₁ * z</td>
<td>#</td>
<td>scalar multiplication, e.g.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3:z = z₁ + z₂</td>
</tr>
</tbody>
</table>
### III.5 Functions returning Type-V

<table>
<thead>
<tr>
<th>Function &amp; Args</th>
<th>Defined when...</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_V(b \mid v_1)_V$</td>
<td>$false$</td>
</tr>
<tr>
<td>$(b \mid v_1 \mid v_2)_V$</td>
<td>$(b \mid #_1 \mid false \mid false)$</td>
</tr>
<tr>
<td>$(b \mid v_1 \mid v_2 \mid v_3)_V$</td>
<td>$(b \mid #_1 \mid #_2 \mid #_3)$</td>
</tr>
</tbody>
</table>
| $v[i_1..i_2]$ | $v 
\& 
\#_1 \n\& \n\#_2 \n\& \n lb(v)_1 \n\leq \ni_1 \n\leq \ni_2 + 1 \n\& \nsub(v)$ |
| $<v,i,t>$ | $v 
\& 
\#_1 \n\& \n\#_2 \n\& \n lb(v)_1 \n\leq \ni_1 \n\leq \ni_2 \n\& \nsub(v)$ |

**Interpretation**

- **Type V error value**
- **If** $T(b)$ **then** $i_1$ **else** $E_V$
- **If** $T(b)$ **then** $v_1$ **else** $v_2$
- **If** $T(b)$ **then** $v_3$
- **If** $T(b)$ **then** $v_4$
- **If** $F(b)$ **then** $v_5$
- **If** $F(b)$ **then** $v_6$

**Restriction of $v$ to**
- **Interval** $[i_1..i_2]$

**Assignment of $t$ to**
- **Position** $i$ **of vector** $v$
IV. Example programs written and annotated using BITZV

In this appendix we present a number of annotated programs whose purpose is to display the expressive range of BITZV and to document the claim that BITZV is sufficiently expressive to handle a wide variety of sorting, merging, searching and counting programs.

Each program is written using the conventional control structures ";", if-then-else, while-do and the assignment statement. In some programs we use recursion.

The data types used are strictly limited to B, I, T, Z and V, and all function and predicate symbols in both the programs and the assertions are drawn from those in the language BITZV. One notational compromise has been permitted: rather than writing the assignment statement

\[ v \leftarrow <v_{j}, t>; \]

we use the usual notation

\[ v[i] \leftarrow t; \]

Because the program assertions are written in BITZV, they are to be interpreted according to the semantics of the logic PFC3 developed in Chapter 3. They are intended to be sufficiently strong to verify the programs to the standard of "firm correctness". Also, the specifications for the programs are as strict as possible, and all boundary cases (such as zero-length vectors) have been included.

Although these programs are annotated with assertions, the full verifications have not been carried out; hence, there may still be errors. The point of this Appendix is merely to illustrate the expressive range of the language BITZV, not to practice hand-verification.
IV.1 Binary Insertion Sort Program

(1) \textbf{var} \quad A_0 : V, \quad !\text{Type declarations} \\
\quad A : V, \quad \text{!} \\
\quad i, j, \text{low}, \text{hi}, \text{split} : I, \\
\quad \text{temp} : T; \\

(2) \textbf{precondition} \quad A = A_0; \\
\quad \text{! The main loop cycles from } i=lb(A)+1 \text{ to } i=ub(A), \\
\quad \text{! each time inserting } A[i] \text{ into its proper place in} \\
\quad \text{! the already-sorted part } A[lb(A)..<i-1] \\

(3) \quad i \leftarrow lb(A)+1; \\

(4) \textbf{invariant} \quad \{A\} = \{A_0\} \land lb(A)=lb(A_0) \land lb(A)<i \land isub(A)+2 \land \\
\quad \left\{ \text{ord}(A[lb(A)..<i-1]) \lor i=ub(A)+2 \right\} \\

(5) \quad \textbf{while} \text{ isub}(A) \text{ do} \\
\quad \text{begin} \\
\quad \quad ! \text{ Binary-search } A[lb(A)..<i-1] \text{ to find the right place} \\
\quad \quad \text{! to insert } A[i]. \text{ At the end we will have} \\
\quad \quad \text{! low = hi = the final position for } A[i], \\
\quad \quad \text{!} \\
\quad \quad \text{low} \leftarrow lb(A); \\
\quad \quad \text{hi} \leftarrow i; \\

(6) \textbf{invariant} \quad \{A\} = \{A_0\} \land lb(A)=lb(A_0) \land \text{ord}(A[lb(A)..<i-1]) \land \\
\quad \quad lb(A) \leq \text{low} \land \text{low} \leq \text{hi} \leq i \land isub(A) \land \\
\quad \quad \left( A[\text{low-1}] \leq A[i] \lor \text{low} = \text{lb}(A) \right) \land (A[i] < A[\text{hi}] \lor \text{hi} = i) \\

(7) \quad \textbf{while} \text{ low}<\text{hi} \text{ do} \\
\quad \quad \text{begin} \\
\quad \quad \quad \text{split} \leftarrow (\text{low+hi})/2; \\
\quad \quad \text{if } A[\text{split}] \leq A[i] \text{ then} \\
\quad \quad \text{\textbf{else} \quad \text{hi} \leftarrow \text{split} \\
\quad \quad \text{\textbf{end} \quad \text{hi} \leftarrow \text{split}} \\
\quad \quad \text{end}; \\
\quad \quad ! \text{Move each element of } A[\text{hi}..<i-1] \text{ one location} \\
\quad \quad \text{! to the right so that } A[i] \text{ can be inserted} \\
\quad \quad \text{! at } A[\text{hi}] \\

(8) \quad \text{temp} \leftarrow A[i]; \\

(9) \quad \text{j} \leftarrow i;
(16) \[ \text{invariant } \{ A[\text{lb}(A) \ldots j-1] \} + \{ \text{temp} \} + \{ A[j+1 \ldots \text{ub}(A)] \} = \{ A_n \} \land \]
\[ \text{lb}(A) = \text{lb}(A_0) \land \text{ord}(A[\text{lb}(A) \ldots j-1]) \land \text{ord}(A[j+1 \ldots i]) \land \]
\[ \text{lb}(A) \leq \text{hi} \land \text{hi} \leq j \land j \leq \text{ub}(A) \land \]
\[ (A[\text{hi}-1] \leq \text{temp} \lor \text{hi} = \text{lb}(A)) \land (\text{temp} < A[\text{hi}] \lor \text{hi} = i) \]

(17) \[ \text{while } j > \text{hi} \text{ do} \]
\[ \hspace{1em} \text{begin} \]
\[ \hspace{2em} A[j] \leftarrow A[j-1]; \]
\[ \hspace{2em} j \leftarrow j-1 \]
\[ \hspace{1em} \text{end;} \]

! Insert the former A[i] (now temp) into its proper place.

(18) \[ A[\text{hi}] \leftarrow \text{temp}; \]

! Prepare to work with next element of A

(19) \[ i \leftarrow i+1 \]

(22) \[ \text{postcondition } \{ A \} = \{ A_0 \} \land \text{lb}(A) = \text{lb}(A_0) \land \text{ord}(A) \]
IV.2 Two-way Merge Program

This program merges two ordered vectors A, B into a third vector C of the correct size.

(1) \textbf{var} \ A_0, B_0, C_0: V. \quad ! \text{The initial values; used only in assertions, not in the program.}
\quad A, B, C: V. \quad ! \text{Vectors A, B to be merged into C.}
\quad i, j, k: I. \quad ! i, j, k index A, B and C respectively.

(2) \textbf{precondition} A = A_0 \land B = B_0 \land C = C_0 \land \text{ord}(A) \land \text{ord}(B) \land \\
\text{len}(C) = \text{len}(A) + \text{len}(B);

! Initialize the index variables

(3) \begin{align*}
  i & \leftarrow \text{lb}(A); \\
  j & \leftarrow \text{lb}(B); \\
  k & \leftarrow \text{lb}(C);
\end{align*}

! The main loop: each iteration assigns \text{min}(A[i], B[j]) ! to \text{C}[k], and increments \text{k} and either \text{i} or \text{j}.

(4) \textbf{invariant} A = A_0 \land B = B_0 \land \text{lb}(C) = \text{lb}(C_0) \land \\
\text{ub}(C) = \text{ub}(C_0) \land \\
\text{ord}(A) \land \text{ord}(B) \land \\
\text{len}(C) = \text{len}(A) + \text{len}(B) \land \\
\text{lb}(A) \leq i \land i \leq \text{ub}(A) + 1 \land \\
\text{lb}(B) \leq j \land j \leq \text{ub}(B) + 1 \land \\
\text{lb}(C) \leq k \land k \leq \text{ub}(C) + 1 \land \\
\{\text{C}[\text{lb}(C) \ldots k-1]\} = \{\text{A}[\text{lb}(A) \ldots i-1]\} + \{\text{B}[\text{lb}(B) \ldots j-1]\} \land \\
\text{ord}(\text{C}[\text{lb}(C) \ldots k-1])
\end{align*}

(5) \quad \textbf{while} \ \text{isub}(A) \land j \leq \text{ub}(B) \ \textbf{do}

(6) \quad \quad \textbf{if} \ A[i] \leq B[j] \quad \textbf{then}

(7) \quad \quad \quad \text{C}[k] \leftarrow A[i];

(8) \quad \quad \quad i \leftarrow i + 1

(9) \quad \quad \textbf{else}

(10) \quad \quad \quad \text{C}[k] \leftarrow B[j];

(11) \quad \quad \quad j \leftarrow j + 1

(12) \quad \quad \quad \textbf{end};

(13) \quad \quad \textbf{end};

(14) \quad \quad \textbf{end};
! Cleanup: if vector A was exhausted first, then copy the remainder
! of B into the remaining space in C; otherwise copy
! the remainder of A into the remaining space of C.

(12) \textbf{if} \ j > \text{ub}(B) \ \textbf{then}

! Copy remainder of A into C

(13) \textbf{invariant} \ A = A_0 \land B = B_0 \land \text{lb}(C) = \text{lb}(C_0) \land \text{ub}(C) = \text{ub}(C_0) \land \text{ord}(A) \land
    \text{len}(C) = \text{len}(A) + \text{len}(B) \land \text{lb}(A) \leq i \land i \leq \text{ub}(A) + 1 \land
    \text{lb}(C) \leq k \land k \leq \text{ub}(C) + 1 \land
    \{C[\text{lb}(C) \ldots k-1]\} = \{A[\text{lb}(A) \ldots i-1]\} + \{B\} \land
    \text{ord}(C[\text{lb}(C) \ldots k-1])

(14) \textbf{while} i \leq \text{ub}(A) \ \textbf{do}
    \textbf{begin}
    C[k] \leftarrow A[i];
    i \leftarrow i + 1;
    k \leftarrow k + 1
    \textbf{end}

\textbf{else}

! Copy remainder of B into C

(18) \textbf{invariant} \ A = A_0 \land B = B_0 \land \text{lb}(C) = \text{lb}(C_0) \land \text{ub}(C) = \text{ub}(C_0) \land \text{ord}(B) \land
    \text{len}(C) = \text{len}(A) + \text{len}(B) \land \text{lb}(B) \leq j \land j \leq \text{ub}(B) + 1 \land
    \text{lb}(C) \leq k \land k \leq \text{ub}(C) + 1 \land
    \{C[\text{lb}(C) \ldots k-1]\} = \{A[\text{lb}(A) \ldots i-1]\} + \{B[\text{lb}(B) \ldots j-1]\} \land
    \text{ord}(C[\text{lb}(C) \ldots k-1])

(19) \textbf{while} j \leq \text{ub}(B) \ \textbf{do}
    \textbf{begin}
    C[k] \leftarrow B[j];
    j \leftarrow j + 1;
    k \leftarrow k + 1
    \textbf{end};

(23) \textbf{postcondition} \ A = A_0 \land B = B_0 \land \text{lb}(C) = \text{lb}(C_0) \land \text{ub}(C) = \text{ub}(C_0) \land
    \{C\} = \{A\} + \{B\} \land \text{ord}(C)
IV.3 Linear Search Program

This program searches a vector A for the first occurrence of the value key. At termination the value of \( i \) will be the least index in A of key or, if key does not occur in A, \( i \) will be \( \text{ub}(A)+1 \).

1. \( \text{var} \ A_0, A; \ V, \)
   \( \text{key}_0, \text{key}; T, \)
   \( i; I; \)

2. \( \text{precondition} \ A=A_0 \land \text{key} = \text{key}_0; \)
   \( \text{Start the search for key at the low end of } A. \)

3. \( i=\text{lb}(A); \)
   \( \text{Search left-to-right in the vector, and terminate early if the key is found.} \)

4. \( \text{invariant} \ A=A_0 \land \text{key} = \text{key}_0 \land \{A[\text{lb}(A)..i-1]\}<\text{key}>B \land \text{lb}(A) \leq i \land i=\text{ub}(A)+1; \)

5. \( \text{while } i=\text{sub}(A) \text{ and } v[i] \neq \text{key} \text{ do} \)
   \( \text{begin} \)
   \( i+i+1 \)
   \( \text{end}; \)

6. \( \text{postcondition} \ A=A_0 \land \text{key} = \text{key}_0 \land \)
   \( \{A\}<\text{key}>B \land i=\text{ub}(A)+1 \lor \)
   \( \{A[\text{lb}(A)..i-1]\}<\text{key}>B \land A[i]=\text{key} \)
References

[Cook 78] Cook, Stephen A.
Soundness and Completeness of an Axiom System for Program Verification.


[Cooper 72] Cooper, D. C.
Theorem Proving in Arithmetic Without Multiplication.


Structured Programming.

[Dershowitz 73] Dershowitz, Nachum and Manna, Zohar.
Proving Termination with Multiset Orderings.
Sail Memo AIM-310, CSD Report STAN-CS-78-651, Stanford University
Computer Science Department, Mar., 1978.

[Dijkstra 76] DijkstraDiscipline.

*A Discipline of Programming*.

[Knuth 68] Knuth, Donald E.
Addison-Wesley, 1968.

[Knuth 73] Knuth, Donald E.

Mathematical Theory of Computation.

An Automatic Verifier for a Class of Sorting Programs.

A Decision Procedure for the Correctness of a Class of Programs.
Tech Report 78/1, Dept. of Computer Science, University of Melbourne,

Predicate Calculus with "undefined" as a Truth-Value.
AI Memo #1, Stanford University, Mar., 1963.

Simplification by Cooperating Decision Procedures.
<table>
<thead>
<tr>
<th>[Reynolds 79]</th>
<th>Reynolds, John C.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Reasoning About Arrays.</td>
</tr>
<tr>
<td>[Rogers 67]</td>
<td>Rogers, Hartley, Jr.</td>
</tr>
<tr>
<td></td>
<td><em>Theory of Recursive Functions and Effective Computability.</em></td>
</tr>
<tr>
<td>[Smullyan 71]</td>
<td>Smullyan, Raymond M.</td>
</tr>
<tr>
<td></td>
<td><em>First-Order Logic.</em></td>
</tr>
<tr>
<td></td>
<td>Springer-Verlag, 1971.</td>
</tr>
<tr>
<td></td>
<td>Verification Decidability of Presburger Array Programs.</td>
</tr>
<tr>
<td></td>
<td><em>JACM</em> 27(1), Jan., 1980.</td>
</tr>
</tbody>
</table>