Technical Report 595

The Definition and Implementation of a Computer Programming Language

Guy Lewis Steele Jr.

MIT Artificial Intelligence Laboratory

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AI-TR-595
2. GOVT ACCESSION NO.:
3. RECIPIENT'S CATALOG NUMBER: AD A096556

4. TITLE *(and Subtitle)*: The Definition and Implementation of a Computer Programming Language Based on Constraints

5. TYPE OF REPORT & PERIOD COVERED: Technical Report

6. PERFORMING ORG. REPORT NUMBER:

7. AUTHOR(s): Guy Lewis Steele Jr.

8. CONTRACT OR GRANT NUMBER(s): N00014-80-C-0505

9. PERFORMING ORGANIZATION NAME AND ADDRESS:
Artificial Intelligence Laboratory
545 Technology Square
Cambridge, Massachusetts 02139

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:

11. CONTROLLING OFFICE NAME AND ADDRESS:
Advanced Research Projects Agency
1400 Wilson Blvd
Arlington, Virginia 22209

12. REPORT DATE: August 1980

13. NUMBER OF PAGES: 372

14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*:
Office of Naval Research
Information Systems
Arlington, Virginia 22217

15. SECURITY CLASS. *(of this report)*: UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT *(of this Report)*

Distribution of this document is unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Constraints
Local Propagation
Theorem Provers
Automatic deduction
Programming Languages
Data Flow

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

The constraint paradigm is a model of computation in which values are deduced whenever possible, under the limitation that deductions be local in a certain sense. One may visualize a constraint "program" as a network of devices connected by wires. Data values may flow along the wires, and computation is performed by the devices. A device computes using only locally available information (with a few exceptions), and places newly derived values on other, locally attached wires. In this way computed values are propagated. An advantage of the constraint paradigm (not unique to it)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

is that a single relationship can be used in more than one direction. The connections to a device are not labelled as inputs and outputs; a device will compute with whatever values are available, and produce as many new values as it can. General theorem provers are capable of such behavior, but tend to suffer from combinatorial explosion; it is not usually useful to derive all the possible consequences of a set of hypotheses. The constraint paradigm places a certain kind of limitation on the deduction process.

The limitations imposed by the constraint paradigm are not the only one possible. It is argued, however, that they are restrictive enough to forstall combinatorial explosion in many interesting computational situations, yet permissive enough to allow useful computations in practical situations. Moreover, the paradigm is intuitive; it is easy to visualize the computational effects of these particular limitations, and the paradigm is a natural way of expressing programs for certain applications, in particular relationships arising in computer-aided design.

A number of implementations of constraint-based programming languages are presented. A progression of ever more powerful languages is described, complete implementations are presented, and design difficulties and alternatives are discussed. The goal approached, though not quite reached, is a complete programming system which will implicitly support the constraint paradigm to the same extent that LISP, say, supports automatic storage management.

A Dissertation on Research Concerning

The Definition and Implementation of A Computer Programming Language

intended as a platform on which to erect

Systems for Computer-Aided Design of Engineered Objects

based on

CONSTRAINTS

A Model for Computation

combining

A Simple Declarative Semantics

with

A Vivid Intuitive Visualization

as a network of

Simultaneously Active Physical Devices Computing in Parallel

Without Prior Prejudice as to the Direction of the Flow of Data

using the technique of

Local Propagation

augmented by

Dependency-Directed Backtracking for Detecting and Resolving Global Inconsistencies

Guy Lewis Steele Jr.

August 1980

This report reproduces a dissertation submitted on August 8, 1980 to the Department of Electrical Engineering and Computer Science of the Massachusetts Institute of Technology in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's artificial intelligence research is provided in part by the Advance Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505.

© Guy Lewis Steele Jr. 1980

The author hereby grants to M.I.T. permission to reproduce
and to distribute copies of this thesis document in whole or in part.

The Definition and Implementation of
A Computer Programming Language
Based on Constraints

Guy Lewis Steele Jr.

Submitted to the Department of Electrical Engineering and Computer Science
on August 8, 1980 in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

ABSTRACT

The constraint paradigm is a model of computation in which values are deduced whenever possible, under the limitation that deductions be *local* in a certain sense. One may visualize a constraint "program" as a network of devices connected by wires. Data values may flow along the wires, and computation is performed by the devices. A device computes using only locally available information (with a few exceptions), and places newly derived values on other, locally attached wires. In this way computed values are *propagated.*

An advantage of the constraint paradigm (not unique to it) is that a single relationship can be used in more than one direction. The connections to a device are not labelled as inputs and outputs; a device will compute with whatever values are available, and produce as many new values as it can. General theorem provers are capable of such behavior, but tend to suffer from combinatorial explosion; it is not usually useful to derive *all* the possible consequences of a set of hypotheses. The constraint paradigm places a certain kind of limitation on the deduction process.

The limitations imposed by the constraint paradigm are not the only one possible. It is argued, however, that they are restrictive enough to forestall combinatorial explosion in many interesting computational situations, yet permissive enough to allow useful computations in practical situations. Moreover, the paradigm is *intuitive*; it is easy to visualize the computational effects of these particular limitations, and the paradigm is a natural way of expressing programs for certain applications, in particular relationships arising in computer-aided design.

A number of implementations of constraint-based programming languages are presented. A progression of ever more powerful languages is described, complete implementations are presented, and design difficulties and alternatives are discussed. The goal approached, though not quite reached, is a complete programming system which will implicitly support the constraint paradigm to the same extent that LISP, say, supports automatic storage management.

Thesis Supervisor: Gerald Jay Sussman
Title: Associate Professor of Electrical Engineering

This work is dedicated to the greater glory of God Almighty.

Acknowledgements

I would like to acknowledge the contributions to this work of the following people and other entities, and offer them my profound gratitude:

Gerald Jay Sussman, advisor, colleague, friend, who has prodded me when I was stuck, encouraged me when I was depressed, enlightened me when I was blind, complimented me was I was actually working well, and who is generally a jolly fellow to be around; he *cares* about his students. He originally inspired nearly all of the good ideas in this dissertation.

Richard Brown, Peter Deutsch, Jon Doyle, Ken Forbus, David McAllester, and Howard Shrobe, who have worked on various versions of constraint systems or truth maintenance systems at M.I.T. and provided many useful comments, criticisms, insights, and ideas relevant to this research.

Other members of the M.I.T. Artificial Intelligence Laboratory or other M.I.T. laboratories who have taken an interest in my work and/or generally contributed to the comfortable and intellectually stimulating ambience there; I would like especially to mention Professors Jon Allen, Ed Fredkin, Carl Hewitt, Berthold Horn, Marvin Minsky, Paul Penfield, and Patrick Winston, and also Howard Cannon, Danny Hillis, Jack Holloway, Tom Knight, Neil Mayle, Margaret Minsky, Jonathan Rees, Chuck Rich, and Jon Taft.

The people who have worked on the hardware and software of the Lisp Machine, to make it the incredibly efficient and versatile programming environment which enabled this research to be conducted in a reasonable time; they include Alan Bawden, Howard Cannon, Richard Greenblatt, Jack Holloway, Tom Knight, Mike McMahon, David Moon, Richard Stallman, and Daniel Weinreb.

The Fannie and John Hertz Foundation, which provided the fellowship under which this research was done.

Donald E. Knuth, Richard Zippel, myself, and others who created and improved the TEX text formatter, which processed the text of this dissertation; and the people at Xerox PARC who produced the Alto and its software, used to produce the illustrations, and the Dover, which printed the original hard copies.

The congregation at the First United Presbyterian Church of Quincy, for their fellowship and support, including Pastor Roger Kvam, Sandy and Allison Willson, Harry and Mary Long, David and Karen Green, Bob and Laurie Gruel, Bruce and Nancy Rhodes, Wayne Wilson, Alden Drake, Jennifer Ward, Ruth Lambiase, Bob McCarthy, Ray Cornils, Lois Cornils, and many others.

Chuck, the peculiar poodle, who as far as I know *still* barks in the night.

David A. Steele, my brother, who used to keep me up to date on cultural affairs, but now *is* cultural affairs (hey, Dave!); and his wife, Suzanne Messer Steele.

Uncle Henry and Aunt Carolyn Damm, and cousin Chrissie, and also Cousin Sue Steele; it's nice to have family.

The Reverend Doctor Guy Lewis Steele, Sr., and Nalora Steele, my parents, who have provided unflagging and unbounded patience, support, encouragement, opportunity, resources, and just plain love for over twenty-five difficult years. Thank you, Dad; thank you, Mom.

Cordon Ruthven Kerns and Ruth Kerns, my other parents, and also Donald and David Kerns, and David and Patty Kerns Auwerda, and Jimbo Kerns and beyond, who have welcomed me as their own. It's *nice* to have family!

Barbara Kerns Steele, my wife. "What we cannot express in words, we must therefore pass over in silence." (Wittgenstein, *Tractatus Logico-Philosophicus*, §7). I love you, Barbara.

Oh, prettily preen the primly prose
That blooms amidst the Snunday snows
And gloom the glibly gleaming glows
While subtly supping sweet suppose.
—Walt Kelly (1952)
I Go Pogo

Contents

Acknowledgements 5

Contents 7

Figures 12

Tables 14

Chapter One: Introduction 19

1.1. The Constraint Model of Computation 21
1.1.1. Simple statement of relationships constitutes declarative programming . . 22
1.1.2. Constraints use local deduction techniques to compute solutions 24
1.1.3. Constraint networks can maintain the history of a computation 26
1.1.4. Assumptions provided limited non-monotonic behavior 28
1.2. The Thesis 29
1.3. Overview of the Dissertation 29
1.3.1. The author had a grand program for solving the entire problem 30
1.3.2. The author settled for doing half thoroughly rather than all poorly 30

Part One: Constraints

Chapter Two: Propagation 37

2.1. A Trivial Constraint Language 37
2.2. Implementation of a Trivial Constraint Language 42
2.2.1. Cells are used to represent variables 42
2.2.2. Constraints are instances of constraint-types 48
2.2.3. Equating of cells links them and propagates values 49
2.2.4. Constraints are implemented as sets of rules 52
2.3. Sample Execution of a Constraint Program 57
2.4. A Difficulty with Division 62
2.5. Summary of the Trivial Constraint Language 66

Chapter Three: Dependencies 68

3.1. Responsible Programs 69
3.1.1. Dependency information can be used to explain computations 69
3.1.2. Required parameters can be deduced from the network structure 72
3.2. Recording Dependencies 74
3.3. Producing Explanations 82
3.4. Representing Symbolic Results in the Network 89
3.4.1. Subgraphs of the network may be printed as algebraic expressions 89
3.4.2. Choosing a subgraph is guided by dependencies and better-name heuristics 96
3.5. Summary of Some Uses for Dependencies 102

Chapter Four: Retraction 104

4.1. Forgiving Systems 104
4.1.1. Connecting conflicting cells can cause contradictions 105
4.1.2. Propagation potentially poses problems for predefined pins 111
4.1.3. Erroneous equatings elicit execution exceptions equally easily 116
4.2. Implementation of Retraction Mechanisms 121
4.3. Summary of the Retraction Mechanisms 132

Chapter Five: Assumptions 133

5.1. Definition of Assumption Constructs 134
5.2. Implementation Problems 135
5.2.1. Nogood sets can be used to locally record contradictions 135

5.2.2. Resolution can derive new nogood sets from old ones 137
5.3. Implementation of Assumption Mechanisms 141
5.4. Examples of the Use of Assumptions 157
5.4.1. Simple assumptions are persistent 157
5.4.2. Oneof assumptions can express and solve the four queens problem . . . 162
5.5. Discussion of the Assumption and Nogood Set Mechanisms 174

Part Two: Engineering

Chapter Six: Efficiency 181

6.1. The New Improved Language 183
6.2. The New Improved Techniques 186
6.2.1. Cells explicitly record multiple support and equatings 186
6.2.2. Constraints use arrays indexed by pin number 192
6.2.3. Constants are considered an immutable part of the wiring 196
6.2.4. A queue-based control structure aids efficiency heuristics 196
6.2.5. Generalized algebraic notation can express any network 197
6.2.6. The size of nogood sets can be heuristically reduced 198
6.2.7. Statistics counters measure performance 200
6.3. The New Improved Implementation 201
6.3.1. Symbolic constants provide names for internal marker values 201
6.3.2. Statistics counters make it easy to instrument code 203
6.3.3. Rules are data structures and catalogued in arrays 204
6.3.4. Cells have fields that were formerly in repositories 207
6.3.5. The value of a cell may differ from the value of its node 210
6.3.6. A newly generated cell is its own puppet 211
6.3.7. Hash tables store and retrieve objects indexed by given keys 212
6.3.8. Constant, default, and parameter cells have dummy rules 215
6.3.9. Declaration of variables and constraints may require housekeeping . . . 217
6.3.10. A queue is yet another abstract data structure 219
6.3.11. The task scheduler simply scans the queues in order 223
6.3.12. The priority of a rule depends on its properties 224
6.3.13. Rule definitions explicitly specify output pins 225
6.3.14. The triggers of a rule must have values when it is run 227
6.3.15. Installing a value in a pin changes the pin's cell-state 228
6.3.16. Usurping a supplier simply reverses links from usurper to supplier . . . 232
6.3.17. Signalling a contradiction merely queues a contradiction task 234

6.3.18. Contradictions must still hold at the time of processing 236
6.3.19. Computation of premises also determines summarizations of defaults . . 238
6.3.20. Contradiction processing traces premises and chooses a culprit 242
6.3.21. Awakening selects only relevant rules for queuing 246
6.3.22. Forgetting a cell's value lets friends (or rebels) step in 248
6.3.23. The `lookup` functions scans the constraint-types's `vars` array 253
6.3.24. Equatings are recorded explicitly and initialize links 255
6.3.25. Node disconnections can be done by dissolving and reconnecting 260
6.3.26. Destroying a variable or constraint detaches it from everything 265
6.3.27. Primitive constraints are uniformly defined by `defprim` 268
6.3.28. Checking the nogood sets can advise rules about forbidden values . . . 276
6.3.29. The `why` function prints values forbidden by nogood sets 279
6.3.30. The `why-ultimately` function prints cell-link information 282
6.3.31. The `what` function uses the generalized algebraic form 284
6.4. The New Improved Example 289
6.5. The New Improved Summary 296

Chapter Seven: Correctness 297

7.1. The Structure of Nodes 297
7.2. Constraint-types and Constraints 299
7.3. Rules 300
7.4. Tasks and Queues 302
7.5. Nogood Sets 303
7.6. User Interface 304
7.7. Summary 304

Part Three: Abstraction

Chapter Eight: Hierarchy 309

8.1. New Features for the Constraint Language 309
8.1.1. The user can describe networks using the expression syntax 310
8.1.2. The user can define non-primitive constraints 311
8.1.3. Pathnames may be written is abbreviated form 313
8.1.4. The `vector` construct provides limited iteration 313
8.2. Implementation of Parsing and Macros 316
8.2.1. Macro-constraints are instances of macro-constraint-types 316
8.2.2. Owners can now be constraints *or* macro-constraints 320

8.2.3. Macro-constraints can be created and destroyed 321
8.2.4. The `the` construct can refer to parts of a macro-device 324
8.2.5. A "read-eval-print" loop processes user requests 325
8.2.6. User input forms are divided into three categories 327
8.2.7. Defining a macro generates a macro-constraint-type 327
8.2.8. Statements are reduced to simple statements 330
8.2.9. Pathnames with periods are one of many forms of reference 334
8.2.10. Vectors are easily defined in terms of macros 336
8.3. Example of the Use of Macro-Constraints 337
8.4. Discussion of the Macro Language 341

Chapter Nine: Compilation 344

Chapter Ten: Conclusions 346

10.1. Comparisons with Other Work 347
10.1.1. `sketchpad` relaxed constraints on geometric diagrams 347
10.1.2. Data flow computations use parallel directional devices 347
10.1.3. Waltz's algorithm filters scene labels by local propagation 348
10.1.4. Semantic networks propagate symbolic tags 349
10.1.5. Freuder's method propagates by synthesizing higher-order constraints . 349
10.1.6. `prolog` uses chronological backtracking on horn clauses 350
10.1.7. `thinglab` provides a class hierarchy and uses pathnames 352
10.1.8. `el` and `ars` analyze electrical circuits by local propagation 353
10.1.9. Truth maintenance systems are general dependency managers 353
10.1.10. A simple constraint language was designed two years before this 355
10.1.11. Other work using constraints 356
10.2. Present and Future Work 357
10.2.1. Tables can be done "the obvious way" or by "algebra" 357
10.2.2. Recursive constraint definitions require conditional expansion 359
10.2.3. Explanations should take advantage of the macro-call hierarchy 359
10.2.4. A constraint language should be meta-circular 360
10.2.5. Algebra is operating on the network structure 361
10.2.6. The system may need control advice from the user 361
10.2.7. Techniques are needed for run-time storage reclamation 362
10.3. Contributions of This Research 363

References 365

Est brilgum; tovi slimici
In vabo tererotitant;
Brogovi sunt macresculi,
Momi rasti strugitant.
—Lewis Carroll
"Gabrobocchia"
Aliciae per Speculum Transitus
Translation by Clive Harcourt Carruthers (1966)

Figures

FIGURE 2-1. Primitive Constraint Devices on Integers 39
FIGURE 2-2. A Constraint Network for Converting Temperatures 40
FIGURE 2-3. Computation of a Temperature Conversion 41
FIGURE 2-4. Some Organizations for Implementing Cells 42
FIGURE 2-5. Three Equivalent Cells with Value Five 44
FIGURE 2-6. The Result of the Creating a Pin for an Adder 47
FIGURE 2-7. The `adder` Constraint-Type and an Instance 50
FIGURE 2-8. Operation of the `maxer` Constraint 61
FIGURE 2-9. A Temperature Conversion Which "Failed" 63
FIGURE 2-10. Constraining Three Points to be Equally Spaced (i) 64
FIGURE 2-11. Constraining Three Points to be Equally Spaced (ii) 65
FIGURE 2-12. A Redundant Network for Equally Spacing Three Points 65
FIGURE 2-13. A Cycle-Free Network for Equally Spacing Three Points 66

FIGURE 3-1. Multiple Suppliers in the Equal-Spacing Network 74
FIGURE 3-2. A Dependency Structure for Which `premises` Takes Exponential Time 84
FIGURE 3-3. Constraining Four Points to be Equally Spaced 92
FIGURE 3-4. Computing Equal Spacing for Four Points 93

FIGURE 4-1. Computation of a Temperature Conversion, Using a Default Value 106
FIGURE 4-2. Recomputation of a Temperature Conversion 108
FIGURE 4-3. Another Recomputation of a Temperature Conversion 109
FIGURE 4-4. A Contradiction in a Four-Point Spacing Network 112
FIGURE 4-5. Defeating Two Defaults in a Four-Point Spacing Network 113
FIGURE 4-6. Redundant Premises for a Four-Point Spacing Network 114
FIGURE 4-7. Surviving Two Contradictions a Four-Point Spacing Network 116

FIGURE 4-8. A Partially Dissolved Four-Point Spacing Network 117
FIGURE 4-9. Computation in a Partially Dissolved Spacing Network 118
FIGURE 4-10. A Four-Point Spacing Network Modified by Reconnection 119
FIGURE 4-11. A Usefully Modified Four-Point Spacing Network 120
FIGURE 4-12. A Usefully Modified Four-Point Spacing Network 121

FIGURE 5-1. A Temperature Conversion Network with an Assumption 135
FIGURE 5-2. A Temperature Conversion Network, after Retracting an Assumption 135
FIGURE 5-3. A `oneof` Cell for which No Alternative Works 138
FIGURE 5-4. Assuming Zero Does Not Work 138
FIGURE 5-5. Assuming One Does Not Work 139
FIGURE 5-6. Assuming Two Does Not Work 139
FIGURE 5-7. Causing a Contradiction and Retraction Eventually Works 140
FIGURE 5-8. Situations Examined for Four Queens Using Chronological Backtracking . . . 165
FIGURE 5-9. Constraint Network for the Four Queens Problem 168
FIGURE 5-10. Situations Examined for Four Queens Using Non-chronological Backtracking 170
FIGURE 5-11. Constraint Network for Making a General Choice 178

FIGURE 6-1. Data Structure for a Node with No Value 188
FIGURE 6-2. Data Structure for a Node with a Confirmed Value 189
FIGURE 6-3. Data Structure for a Node in a Contradictory State 190
FIGURE 6-4. The Constraint-type `gate` and Its Rules 194
FIGURE 6-5. Summarizing Default Cells in the Network 199
FIGURE 6-6. Usurping a Supplier 231

FIGURE 8-1. User Definition of the `if` Device 312
FIGURE 8-2. Pictorial Representation of the Body Prototype for a Vector 314
FIGURE 8-3. An Entire Vector, and Its Connections 315
FIGURE 8-4. One Stage of a GCD Computation 338

The Moon is a Madness.
A Madness of mine.
I made her of mustard
And mulberry wine.

I garbed her in silver
And strawberry cheese
And halved her in quarters.
(Her quarters do please.)

I crowned her and gowned her
In Love all ashine.
So boot her and shoot her
This Madness of mine.
—Walt Kelly (1959)
The Pogo Sunday Brunch

Tables

TABLE 2-1. LISP Code Defining Cell and Repository Data Types 45
TABLE 2-2. Creation of Cells, Constants, and Variables 47
TABLE 2-3. Constraints and Constraint-Types 49
TABLE 2-4. Referring to Pins of a Constraint Device 49
TABLE 2-5. Equating of Cells and Propagation of Values 51
TABLE 2-6. A Simple Tracing Mechanism 52
TABLE 2-7. Implementation of the Constraint Boxes of Figure 2-1 53
TABLE 2-8. Definition of Primitive Constraints and Rules 55
TABLE 2-10. Expanded Second Rule of the `equality` Constraint 55
TABLE 2-9. Expanded Definition of the `equality` Constraint 56
TABLE 2-11. Implementation of `contradiction` and `setc` 57

TABLE 3-1. Extra Repository Fields for Recording Dependencies 75
TABLE 3-2. A Constant Cell Is Its Own Supplier 76
TABLE 3-3. Maintaining Supplier Components When Equating Cells 77
TABLE 3-4. An Incorrect Implementation of Equating 78
TABLE 3-5. Implementation of Primitive Constraints with Dependency Information 79
TABLE 3-6. Definition of `defprim` Which Saves Rule Information 80
TABLE 3-7. Definition of the Ancestor Relationship between Cells with Values 81
TABLE 3-8. Definition of `setc` for Handling Dependencies 82
TABLE 3-9. Code for `why`: Generating a One-Step Explanation 83
TABLE 3-10. Calculation of the Premises Supporting a Value 84
TABLE 3-11. Fast Calculation of Premises 85
TABLE 3-12. Determining Potential Premises for a Cell with No Value 86
TABLE 3-13. Implementation of `why-ultimately` 87

TABLE 3-14. Definition of the `what` Explanation Function 95
TABLE 3-15. The `tree-form` Function and Macros for Numerical Marks 96
TABLE 3-16. Tracing Out a Subgraph of Interest for `what` 98
TABLE 3-17. Copying a Traced Subgraph as a Set of Equations 100
TABLE 3-18. Resetting the Mark Components for `tree-form` 102

TABLE 4-1. Implementation of Constant and Default Cells 122
TABLE 4-2. Delaying Equating Decisions Until after the Merge 123
TABLE 4-3. Handling Contradictions in `setc` 124
TABLE 4-4. Processing and Recovering from Contradictions 125
TABLE 4-5. Retracting Values from the Network 127
TABLE 4-6. A Rewriting of the `premises` Function 127
TABLE 4-7. A Rewriting of the `fast-premises` Function 129
TABLE 4-8. Dissolving a Node—Carefully! 130
TABLE 4-9. Disconnecting a Cell from a Node 131

TABLE 5-1. Data Structure Modifications for Assumptions 142
TABLE 5-2. Implementation of the `assume` Construct. 143
TABLE 5-3. Implementation of the `oneof` Construct. 144
TABLE 5-4. The Rule for `oneof` 145
TABLE 5-5. Looking for Tentative Values for Use as Culprits 146
TABLE 5-6. Constructing and Recording a Nogood Set 147
TABLE 5-7. Merging Nogood Sets When Equating Cells 148
TABLE 5-8. Altering Nogood Sets for a New Repository 149
TABLE 5-9. Merging Two Collections of Nogood Sets 150
TABLE 5-10. Forgotten Values May Re-enable Suppressed Assumptions 151
TABLE 5-11. Disconnections Wreak Havoc with Nogood Sets 152
TABLE 5-12. Rapid Destruction of Potentially Invalid Nogood Information 153
TABLE 5-13. Assumptions Are Considered to be Premises 153
TABLE 5-14. New `treeforms` Definitions 154
TABLE 5-16. Constructing a Treeform with a `!` 154
TABLE 5-15. Tracing Missing Treeforms and Treeforms with `!` 155
TABLE 5-17. A More Reliable Version of `process-setc` 156
TABLE 5-18. A LISP Solution to the *N* Queens Problem 162
TABLE 5-19. Constraints for the Four Queens Problem (i) 166
TABLE 5-20. Constraints for the Four Queens Problem (ii) 166
TABLE 5-21. Constraints for the Four Queens Problem (iii) 167
TABLE 5-22. Constraints for the Four Queens Problem (iv) 167
TABLE 5-23. Implementation of the `firstoneof` Construct 175

TABLE 5-24. The Rule for `firstoneof` 177

TABLE 6-1. Definitions of Symbolic Constants 201
TABLE 6-2. Statistics Counter Mechanism 203
TABLE 6-3. Data Structures for Constraint-types, Constraints, and Rules 204
TABLE 6-4. Data Structures for Repositories and Cells 206
TABLE 6-5. Functions for Accessing Values of Cells and Nodes 209
TABLE 6-6. Generation of Repositories and Cells 211
TABLE 6-7. Hash Table Definition and Generation 212
TABLE 6-8. Hash Table Lookup and Install Operations 213
TABLE 6-9. Dummy Rules for Constant, Default, and Parameter Cells 215
TABLE 6-10. Generation of Constant, Default, and Parameter Cells 216
TABLE 6-11. Declaration of Variables and Constraints 217
TABLE 6-12. Queue Data Structure and Definition 219
TABLE 6-13. Queue Operations 220
TABLE 6-14. Constraint System Queue Definitions and Task Scheduler 222
TABLE 6-15. Deciding in Which Queue to Enqueue a Rule 224
TABLE 6-16. Applying a Rule to a Constraint 226
TABLE 6-17. Installing a Computed Value in a Pin (i) 228
TABLE 6-18. Installing a Computed Value in a Pin (ii) 229
TABLE 6-19. Usurping the Throne of the Supplier of a Node 231
TABLE 6-20. Signalling Contradictions 233
TABLE 6-21. Running a Contradiction Task 235
TABLE 6-22. Fast Computation of Premises and Related Quantities 237
TABLE 6-23. Gathering Premise and Link Information 239
TABLE 6-24. Tracing Premises for a List of Cells, and Unmarking 241
TABLE 6-25. Processing of Contradictions 242
TABLE 6-26. Formation and Installation of Nogood Sets 243
TABLE 6-27. Choosing a Culprit 243
TABLE 6-28. Awakening of Rules 245
TABLE 6-29. Forgetting a Cell's Value and Its Consequences 247
TABLE 6-30. Retracting a Value, and Tracing of Consequences 249
TABLE 6-31. Forgetting a Friendless King (Very Hairy!) 251
TABLE 6-32. Referring to Pins Using `the` 253
TABLE 6-33. Equating of Cells and Recording Equatings Explicitly 254
TABLE 6-34. Merging Values and Arranging Cell Links 256
TABLE 6-35. Merging Two Nodes with Values and Handling Conflicts 257
TABLE 6-36. Altering and Merging of Nogood Sets 258

TABLE 6-37. Testing Ancestorhood 259
TABLE 6-38. Dissolving a Node 260
TABLE 6-39. Detaching, Disconnecting, and Disequating Cells 262
TABLE 6-40. Fast Expunging of Nogood Information 264
TABLE 6-41. Destroying the Value of a Global Name 265
TABLE 6-42. Definition of Primitive Constraint-types (i) 266
TABLE 6-43. Definition of Primitive Constraint-types (ii) 267
TABLE 6-44. Definition of Primitive Constraint-types (iii) 268
TABLE 6-45. The `assume`, `oneof`, and `firstoneof` Constructs 270
TABLE 6-46. Definition of Primitives 272
TABLE 6-47. Expansion of the Definition of `gate` 273
TABLE 6-48. Definition of Rules 274
TABLE 6-49. Expansions of the Definitions of Two `gate` Rules 275
TABLE 6-50. Checking Whether a Value is Forbidden by a Nogood Set 276
TABLE 6-51. Filtering a Set of Possibilities Using Nogood Sets 278
TABLE 6-52. Implementation of the `why` Function 279
TABLE 6-53. Explaining a True-Supplier, and Printing Forbidden Values 280
TABLE 6-54. Implementation of `why-ultimately` 282
TABLE 6-55. Locating Desired Premises for an Unbound Cell 283
TABLE 6-56. Implementation of `what` 284
TABLE 6-57. Tracing Out an Algebraic Expression in the Network 285
TABLE 6-58. Determining a "Good" Artificial Supplier 286
TABLE 6-59. Constructing the Traced-out Algebraic Expression 287
TABLE 6-60. Checking for a Good Global Name, and Unmarking, for `tree-form` 288

TABLE 8-1. Macro-constraint-types and Macro-constraints 316
TABLE 8-2. New Printing Format for Cells 318
TABLE 8-3. Construction of Pathnames for Cells and Devices 319
TABLE 8-4. The Best Name for a Pin Is Its Pathname 319
TABLE 8-5. Creating and Destroying Things 321
TABLE 8-6. Generating a Constraint or Macro-constraint 323
TABLE 8-7. Looking Up Parts of a Macro-Constraint 324
TABLE 8-8. The Top-Level "Read-Eval-Print" Loop for the Constraint System 325
TABLE 8-9. Discrimination of Input Forms 327
TABLE 8-10. Processing a Macro Definition 328
TABLE 8-11. Generating a Macro-Constraint-Type 329
TABLE 8-12. Parsing Statements 330
TABLE 8-13. Parsing an "Algebraic Expression" 332

TABLE 8-14. Parsing a Reference to a Thing 334
TABLE 8-15. Parsing a Pathname Written with Periods 334
TABLE 8-16. Parsing a "Simple" (Ha!) Symbol 335
TABLE 8-17. Parsing a Global Symbol 336
TABLE 8-18. Parsing a `the` Expression 336
TABLE 8-19. Parsing a `vector` Statement 337

Oh, roar a roar for Nora.
Nora Alice in the night.
For she has seen Aurora
Borealis burning bright.

A furore for our Nora!
And applaud Aurora seen!
Where, throughout the summer, has
Our Borealis been?

—Walt Kelly (1953)
Ten Ever-Lovin' Blue-Eyed Years with Pogo

Chapter One

Introduction

IT IS BY NOW a firmly established piece of the computer science folklore that all sufficiently powerful models of computation are the same because they are all equivalent to a Turing Machine (this is known as Church's thesis). Nevertheless, some models of computation are more *tractable* than others for certain purposes, and this is perhaps as much a matter of psychology as of computer science. Some models evoke mental images and analogies which others do not, and these images and analogies guide one's thinking about a problem. Indeed, some models become so firmly entrenched in the folklore, or seem to correspond so naturally to the structure of certain classes of problems, that one's instinctive approach when faced with similar problems is to turn to those models, and so such models become paradigm solutions for such problems. For example, the notion of a finite-state machine is so closely associated with the parsing of strings that whenever a problem of the form, "Scan a stream of things looking for some cumulative property" arises, my first thought is to frame the solution as a finite-state machine; therefore I *assume* that the solution may be of this form, and then try to fill in the details, and usually this approach works and occasionally not.

Robert Floyd has remarked [Floyd 1979]:

> ... continued advance in programming will require the continuing invention, elaboration, and communications of new paradigms.

This dissertation is an exposition of one such paradigm: constraints. In this paradigm programs consist of statements of relationships among symbolically named quantities which are to be satisfied. What distinguishes a constraint-based language from others is the particular limitations

which are placed on the deductive process which manipulates the relationships in order to produce values.

In the cited work Floyd goes on to say:

> When a programming language makes a paradigm convenient, I will say the language *supports* the paradigm. When a language makes a paradigm feasible, but not convenient, I will say the language *weakly supports* the paradigm. ... most of our languages only weakly support simultaneous assignment, and do not support coroutines at all ... Even the paradigm of structured programming is at best weakly supported by many of our programming languages.

The constraint model of computation is not supported by any programming language in existence today; the closest approximation is probably PROLOG [Warren 1977b]. The research I shall discuss here is an attempt to build a constraint-based language from the ground up. This includes definition of appropriate primitives, means of combining primitives, run-time support, means of abstraction, and a simple compiler.

Again quoting from Floyd:

> A paradigm at an even higher level of abstraction than the structured programming paradigm is the construction of a hierarchy of languages, where programs in the highest-level language operate on the most abstract objects, and are translated into programs on the next lower level language.

This is the paradigm used in the construction of the constraint system presented here. It was built on top of a LISP system, the dialect known as Lisp Machine LISP [Weinreb 1979], developed at M.I.T. In this dissertation I will not only describe the capabilities of the constraint system, but also describe its implementation, making remarks along the way on the techniques used to build large systems quickly and reliably. These techniques include data abstraction, debugging tools, defensive programming, and most particularly building on an existing system rather than re-implementing everything from scratch. LISP provides the user with a data structure printer, a parser, hashing of identifiers to data structures, automatic storage management, and a host of run-time facilities (arithmetic, search procedures, sorting, etc.) right off the bat; the incremental cost of constructing a new language is small.

In summary, I will discuss three things in a somewhat mingled fashion:

(1) The constraint model of computation, and some associated imagery. This will include a static relational model for the meaning of constraints, as well as computational models.

(2) Methods of implementation, including consideration of alternatives. Possible data structures and control structures are compared.

(3) Techniques for construction of large systems, using the constraint system as an example. This point receives somewhat less emphasis in the text, and is represented largely by side remarks and footnotes, and demonstrated by example.

1.1. The Constraint Model of Computation

There are two images, or analogies, which I associate with constraints which make them useful to me. Neither of these is unique to constraints, but the combination is.

The first image is that a constraint is a *declarative statement of relationship.* If I place a constraint that the quantity named a is less than the quantity named b, then there is a known relationship between the two. Similarly, if the sum of three values x, y, and z is constrained to be zero, then there is a stated relationship among the three. This relationship can be viewed in more than one way: for example, one might find convenient for some purposes the asymmetric view that x is minus the sum of the other two.

Predicate calculus and related description methods also model computation by stating relationships. Predicate-calculus-based programs such as PROLOG ([Warren 1977b]; see also [Kowalski 1974]) provide both a relational model for interpretation of the meaning of the program, and a computational model for the algorithmic evolution of the canonical form for this meaning (the "output").

The second image is that a constraint is a *computational device for enforcing the relationship.* I mean for the word "device" to be taken quite literally. I visualize a constraint as a little plastic box with metal pins coming out, just like the dual-in-line (DIP) packages that digital integrated circuits come in. Just as a 7400 series NAND gate will force its output pin to be the logical negation of the logical product of its input pins, so a hypothetical 74000000 series NAND constraint would constrain its three pins to obey the NAND relationship. A constraint does not have designated "inputs" and "outputs", however. At this level of abstraction I say nothing about *how* the relationship is enforced, except to say that the enforcement mechansisms are (mostly) local to the device. I then visualize many of these little boxes being combined by running wires between their pins; these wires represent primitive equality relationships. A constraint program can be drawn very much like an electrical circuit diagram. (Indeed, it was research into analysis of electrical circuits that inspired the current flurry of interest in constraints at M.I.T.[Sussman 1975] [Stallman 1977]; and constraints in Sketchpad [Sutherland 1963] were also drawn as little "devices" connected by "wires" to their arguments.) As in an electrical circuit, all the devices are conceptually active at once; they operate in parallel.

It is this computational metaphor that distinguishes constraints from, say, PROLOG. Both PROLOG and constraints are based on a statement of relationships, but they differ in the additional imagery. PROLOG restricts statements to Horn-clause form, and then likens such clauses to procedure declarations and imposes a backtracking procedure-calling metaphor. Constraints provide the metaphor of interacting discrete physical devices. Other metaphors may also be useful.

The data flow model of computation [Dennis 1973] [Dennis 1975] also takes a view of programming as the wiring together of devices which can then perform computations with as much parallelism is permitted by the wiring structure. Constraints are like data flow in that one can visualize data as flowing from device to device along wires. The two differ in that data flow devices are directional, having specified input pins and output pins; constraints are (in general) adirectional. Data flow can be considered to be a (perhaps very important) special case of the constraint paradigm.

Analog computers also perform computations using devices which are wired together. They operate on a (conceptually) continuous domain, however, represented by voltages or currents. I focus here on constraints as a model of *discrete* computation, on discontinuous data domains. When discrete methods must be used rather than, say, relaxation techniques, the computational strategies are radically different.

1.1.1. Simple Statement of Relationships Constitutes Declarative Programming

The advantage of a relational semantics for a programming language is that a static meaning can be assigned to the program independent of any computational model. This allows various implementations to be judged by a uniform standard, and a new implementation need not reproduce exactly the inessential quirks of an old one. Moreover, the program may well be easier to understand, and even (it is fashionable to say this nowadays) to prove.[1]

It has been argued [Pratt 1977] [Kowalski 1979] [Clark 1980?] that a program is best divided into two components, the *competence* and *performance* components (Pratt's terminology, borrowed from Chomsky). The competence component contains factual information—statements of relationships—which must be manipulated and combined to calculate the desired result. The performance component then deals with the strategy and tactics of the manipulations and combinations. The competence component is responsible for the correctness of the program; the performance component is responsible for efficiency and termination. As an example, the following facts suffice for computation of the greatest common devisor r of two numbers x and y (the example is from Pratt, but the formulation is mine)[2]:

1. Note also that it is easier to start with a good semantics and then implement it than to begin with an implementation and then derive (if one can!) some kind of post-hoc semantics (typically of the Floyd-Hoare style) to justify it.

2. This formulation does not consist simply of universally quantified relationships about gcd along with a request to find $\gcd(x, y)$; such is the nature of the formulation in [Pratt 1977]. This formulation is somewhere between that and a deterministic algorithm, in that it expresses the idea that a sequence should be computed, that elements of the sequence may be computed according to a limited number of specified rules, and that some element of the sequence will be the result. Thus compared with Pratt's version, this already outlines much of the strategy. The freedom remaining is the precise choice of rules used to compute the sequence.

$$z_0 = x \qquad z_1 = y$$
$$\forall i \geq 2 \;\; (z_i = z_{i-1} - z_{i-2}) \vee (z_i = z_{i-2} - z_{i-1})$$
$$r = |z_k| \quad \text{where } z_{i+1} = 0$$

(Proof of formulation, using induction: certainly $\gcd(z_0, z_1) = \gcd(x, y)$. Now suppose that $\gcd(z_{i-1}, z_{i-2}) = \gcd(x, y)$. Either $z_i = z_{i-1} - z_{i-2}$ or $z_i = z_{i-2} - z_{i-1}$. In the first case $\gcd(z_i, z_{i-1}) = \gcd(z_{i-1} - z_{i-2}, z_{i-1}) = \gcd(z_{i-1}, z_{i-2}) = \gcd(x, y)$, and the other case is similar. Therefore for all i, $\gcd(z_i, z_{i-1}) = \gcd(x, y)$. Negative numbers do not matter, because we define $\gcd(-u, v) = \gcd(u, v)$. Moreover, the fact that $\gcd(u, 0) = u$ means that $r = \gcd(x, y)$.)

Now this set of rules is certainly competent; if a value is ever found for r, it will certainly be the gcd of x and y. Moreover, the declarative nature of the rules makes it easy to reason about them. However, the matter of performance is another thing. A strategy is needed for making the choice at each step about which way to subtract. For $x = 15$ and $y = 12$, one valid z sequence is

$$15, 12, 3, 9, 6, 3, 3, 0$$

whereupon $r = 3$. But another valid sequence is

$$15, 12, -3, 9, 12, 3, 9, 6, -3, 9, 6, 3, 3, 0, -3, 3, 6, 3, 3, 0$$

whereupon $r = 3$ (the rules don't say one must use the *first* zero value in the z sequence!). The computation may even fail to converge:

$$15, 12, -3, 15, 18, -3, 21, 24, -3, 27, 30, -3, 33, 36, -3, 39, \ldots$$

The behavior of the performance component cannot simply be left to chance; it can have a significant effect on the computational behavior of the system. (In some sense the only interesting difference between an $n \log n$ Quicksort and an n^2 Bubble Sort is performance.)

We will posit the hypothesis that we would like, in our programming, to concern ourselves first with competence (correctness; "what"), and only then, if at all, worry about performance (efficiency; "how"). This implies that a programming language ought to allow the division of a program into the two separate components. Ideally, the computation could proceed, at some cost perhaps, without any advice on performance; but the more advice the programmer could give, the more efficient the computation could proceed. (Compare this with the declaration of data types in existing programming languages. In most of the algebraic[3] languages the declaration of data types is inextricably bound up with the rest of the program, even though the information is in many cases

3. It used to be that "algebraic" meant "ALGOL-like" or perhaps "FORTRAN-like", but nowadays it seems to mean "PASCAL-like", for PASCAL is the currently popular, though poorer, re-invention of the excellent ALGOL wheel. Perhaps in another year, continuing the trend, the term will mean "ADA-like".

redundant. To my mind that is one of the great advantages of interactive languages such as BASIC, APL, and LISP: the programmer can proceed without all the redundant baggage to the heart of the matter, and then go back later to add the declarations as documentation or advice. In MACLISP [Moon 1974], for example, one often writes a program without any declarations, and then adds numerical declarations later to advise the compiler how to go about getting FORTRAN-like speed for numerical code [Fateman 1973] [Steele 1977]. This is not to say that the programmer should not have the possibility of declarations in mind as he first writes the program. I would suggest, however, that programs written to try out an idea are not necessarily best written using the same methodology with which one crafts the finished product.)

As already mentioned, some other programming languages provide a means for this separation of concerns. They are pretty much alike in their expression of competence: one sugaring or another of predicate logic. (Kowalski states [Kowalski 1980]: "There is only one language suitable for representing information—whether declarative or procedural—and that is first-order predicate logic." He may be right.[4]) However, predicate-calculus-based programming languages differ in the automatic computational methods applied for performance purposes.

In order that a constraint language behave as declaratively as possible, we will posit this design goal (there will be others later):

Design Goal 1

> As far as possible, the computational state of a constraint system should depend only on the relationships stated so far, and not on the order in which they were stated. (However, this order-independence is required only up to any ambiguities in the relationships.)

A constraint program should have a clear declarative semantics unrelated to questions of ordering and process. This is not to say that the results of a constraint program may be unaffected by the order in which things happen—but if ordering does matter, then it is because of an essential (possibly intentional) ambiguity in the stated relationships, in which case the system is explicitly free by fiat to make any choice among those possible.

1.1.2. Constraints Use Local Deduction Techniques to Compute Solutions

The very advantage of predicate calculus-like formalisms is that they inherently make no commitment as to computational technique; therefore predicate calculus is not a complete programming language. As noted in [Bobrow 1980]:

> [Predicate logic] is inadequate as a calculus because it does not make it perspicuous to model issues of memory and resource-limited reasoning ...

4. Kowalski goes on to say, "There is only one intelligent way to process information—and that is by applying deductive inference methods. The AI community might have realised this sooner if it weren't so insular." This seems a trifle strong to me.

And in [Sloman 1980]:

> When a new formalism is little more than a syntactic variant of predicate logic, translation is a useful debunking exercise. It is also useful in posing a challenge to clarify what the translation fails to capture, in other cases. More to the point, in some cases, would be translation into a good general-purpose programming language, e.g. LISP or POP2 with records or property lists and a few general-purpose library routines. I shall start being impressed by new formalisms when they are associated with powerful new useful *techniques*. [Italics his.]

It is computational technique, rather than syntactic notation, which distinguishes predicate-calculus or relational programming languages. The difficulty with general theorem provers is the combinatorial explosion which results from simply trying to deduce all possible consequences from a set of statements. There must be some means of limiting this explosion in a useful way. Such limitations may of course prevent a computing system from arriving at a potentially deducible result, either absolutely (by totally preventing consideration of certain deductions) or relatively (by postponing the relevant deductions beyond an economically feasible amount of other computation). The challenge is to invent a limiting technique which powerful enough to contain the explosion, permissive enough to allow deduction of useful results in most cases of interest, and simple enough that the programmer can understand the consequences of the limiting mechanism. It is this last point which encourages the system designer to base the limiting mechanism on some easily grasped metaphor.

In this dissertation I shall discuss the technique of *local propagation*. One can view the notion of local propagation from the declarative point of view: local propagation amounts to using only one relationship at a time to do *arithmetic* on known values (as opposed to using *algebraic* techniques for deriving new relationships by combining old ones). Of course, most programming languages share this same property; "algebraic" languages are really arithmetic languages, and only the symbolic mathematical systems such as MACSYMA, REDUCE, and SCRATCHPAD truly perform algebra. A constraint-based language would be somewhere between the two types, differing from arithmetic languages in that the direction in which relationships were used would be determined dynamically, on an as-needed basis. As an example, in a constraint language, one would state $a = b + c$, or perhaps $a - b - c = 0$, or some such thing; this statement might then be used computationally to derive b given a and c. In an ALGOL-like language one must explicitly write $b := a - c$. This adirectionality is a property shared by symbolic algebra systems and deductive theorem provers; but then again, they are oriented more towards algebra than arithmetic. By constrast, a constraint system avoids using the full power of algebraic transforms.

Local propagation is perhaps more easily visualized in terms of the discrete device image, however. Think of a 74000083 full adder constraint device, in a five-pin package, the pins being a, b, c, V_{QQ}, and ground. Whenever any two pins have numbers on them, a value is computed

for the third pin. (It is partly by this means that the relationships are enforced; whenever a value is forced by others, it is immediately asserted.) If all three pins have numbers, and they don't obey the relationship, then the device pushes back hard somehow, trying to make the numbers fit its relationship, or perhaps the device goes up in smoke if it doesn't succeed. By analogy, consider a resistor, which takes *two* numbers, called voltage and current—which happen to be represented in an interesting physical way—on each of its two terminals, and enforces certain numerical relationships—also, as it happens, expressed in a physical way—among these numbers. An ideal resistor operates on a conceptually continuous physical domain, but we shall be interested here in discrete domains. The mental image of a physical device is apt, however; its operation is *local* in that it operates only on information immediately to hand on its pins. All the devices are of this form, and by cooperating in parallel they can produce global effects.

One would expect that the constraint model might be a good model for multiprocessor computation for exactly the same reasons that data flow would be. Because each device computes when, and only when, the relevant information is available, a network of cooperating devices can exploit all possible parallelism in the computation. Constraints have the additional advantage that no prior commitment need be made by the programmer as to which pins of a device are input pins and which output, so a single network can be used to perform many different computations (not necessarily all at once, though). This generality of course has its price: a real hardware constraint architecture will be much more complicated than a data flow architecture.

Although the constraint-system implementations discussed here are in fact single-processor simulations, we would like the constraint model to serve as a model for parallel computation by machines each performing locally defined tasks. Therefore we posit another design goal:

Design Goal 2

As far as possible a constraint-based system shall perform its computations on the basis of locally available information only.

We shall indeed achieve this goal for computations in which no conflicts occur; but that is of course the simple case. As we shall see, the constraint model is most useful for analyzing and dealing with conflicts.

1.1.3. Constraint Networks Can Maintain the History of a Computation

Because it is not determined *a priori* whether a given device pin will be used for input or output, neither is it determined in which direction data will flow along a wire (connection, equality). Suppose that associated with each wire is a bit, indicating in which direction data has flowed last; the value of the bit does not matter if no data is on the wire. Then, assuming that no conflicting values have arisen on a wire, when the computation has settled down (all possible local deductions having been made), various nodes of the network will have values, and the wires with values

will form a directed graph, with their bits indicating the directions, describing the history of the computation. Such a graph will constitute the particular data flow program traced out dynamically by the computation on an as-needed (or rather, greedy) basis. Because the graph indicates which values depend on which other values, it is said to encode *dependency information* (*dependencies* for short).

Such a history can be used for many purposes. It can provide explanations of the computation. It can also be used to resolve conflicts. If there is a loop in the network, then a value may propagate around the loop and compute a new and different value for the same quantity. In continuous-domain networks (such as electrical circuits) this is typically used to provide feedback effects which by relaxation cause the conflict to be resolved. Note that relaxation is of necessity a global, not local, process, because it is used to resolve conflicts caused by global properties of the network (the individual devices being assumed to be locally consistent). In the discrete-domain networks to be discussed here relaxation is replaced by the global processes of backtracking and resolution.[5]

The current work at M.I.T. on constraint-based computation was inspired by the need to analyze physical systems and to aid the engineering processes by which such systems are designed. This dissertation began as an effort to provide a basis on which to build a design system for integrated circuits. The intention was to provide the underlying mechanisms for recording design constraints, in the form of rules by which parts of the designed object interact. For integrated circuits, some of these rules are geometric in nature, some electrical, some logical, and some on more than one of these levels. The intention was to provide a uniform mechanism by which rules and values could be recorded, deductions made, and consequences asserted in a way that would interact well with the physical descriptions of the design. Moreover, the mechanism should automatically detect conflicts, and provide information to aid in explaining to the user the reason for the conflict. While the results of this research have not yet been incorporated into a design system, similar mechanisms have been used in design systems such as Daedalus [Shrobe 1980]. My intention is that the research presented here should serve as a basis for building a self-contained constraint-based language to serve as a host system on top of which to build design systems. The simple implementations presented here are similarly built on top of a host LISP system. The constraint language will provide certain services to the implementor of a design system, such as recording design constraints and detecting and resolving conflicts, just as LISP provides certain services such as automatic storage management, which records given data in a structured form using a linear memory, and detects the implicit release of data structures and errors caused by incorrect access to structures.

---

5. Relaxation and resolution are not interchangeable, but complementary, being applicable to different situations. A full constraint-based system would probably need to use both techniques; this was done in THINGLAB [Borning 1979], for example. In this dissertation, however, I arbitrarily focus only upon discrete domains, because relaxation has already been more throughly explored.

In this rôle as a design utility, a constraint system ought to have the property that it works if one gives it but a little information, and works better if one gives it more information. That is, the more relationships it has to work with, the more can be computed. Hence a third design goal:

Design Goal 3

A constraint-based system should, so far as possible, be *monotonic*. The more is known, the more can be deduced, and once a value has been deduced, knowing more true things ought not to capriciously invalidate it.

1.1.4. Assumptions Provided Limited Non-monotonic Behavior

The word "capriciously" is included in Design Goal 3 for a reason. There are some situations where it is useful to make assumptions, in order to provide "default" behavior. For example, one might want to say that an object can be oriented in any of several ways by rotation and reflection, but that some one orientation may be assumed unless explicitly proven otherwise. This is necessary to be able to draw a picture of an incomplete design, for example. If one knows that an inverter is part of a circuit in a particular position but the designer hasn't yet specified whether it faces left or right, it is unsatisfying simply not to draw it; it might be better drawn in some default orientation, perhaps with an annotation to that effect.

Unfortunately, introducing assumptions violates the principle of monotonicity, because information computed on the basis of an assumption may no longer be valid when the assumption is overridden. Therefore providing *more* information may cause *fewer* (though sounder) results to be known. We will permit this limited form of non-monotonicity, but nevertheless desire the results of computations to be relatively stable; hence a value, once computed, should not be retracted by caprice, but rather only because new information has definitely rules it out. Likewise, if either of two values is possible and one is (arbitrarily) chosen, then that value remains until rules out, rather than oscillation occurring.

An assumption can be expressed as a deductive rule of the form "Deduce $x = y$ provided that the system remains consistent." Now of course consistency is a global property, and so an assumption mechanism also violates the design goal of locality. However, the rule can be phrased operationally as, "If one of x and y is known and the other not, deduce $x = y$," which is local, with the understanding that in the event of conflict a general global mechanism for conflict resolution will take over. This is in fact how assumptions are implemented in the systems described in this dissertation.

At least one dependency-recording system [Doyle 1978a] [Doyle 1978b] [Doyle 1979] has been so general as to allow deductions to be made on the basis of *anything* being unknown. Such a system has grossly non-monotonic semantics which leads to some logical difficulties. There has been some work done [McDermott 1979] on formalizing the semantics of non-monotonic logics.

These difficulties are avoided here by confining the non-monotonicity of the system to a fairly well-behaved special case. An assumption mechanism allows the constraint system to make guesses about possible extensions to the solution computed by local propagation, and thus provides a limited means of overcoming the limitations of locality.

1.2. The Thesis

- Constraints are a model for computation which has both a static declarative semantics and an intuitively appealing visualization as physical devices which perform dynamic local computations.
- The constraint paradigm places limitations on the deduction process which are stringent enough to prevent combinatorial explosion, loose enough to permit interesting computations to be performed, and sufficiently comprehensible to allow the programmer to predict the effects of the limitations.
- Constraints provide a natural way to express and enforce the relationships of designed objects, and therefore a constraint-based programming language is a suitable base for building systems for computer-aided design (CAD).
- A constraint system can easily retain information about the history of the computation which can be used to produce explanations of the system's behavior, and to trace the root causes of conflicts.
- Constraints include data flow as a special case. A suitable compiler can reduce a constraint program to a set of data flow programs, one for each possible partitioning of the program's terminals into inputs and outputs.
- Local propagation as the normal mode of computation, plus dependency-directed backtracking for resolving global conflicts, can serve as the implementational basis of an expressively powerful and potentially very efficient computational language.
- A constraint-based language can be efficiently implemented by letting the structure of the implementation correspond in a direct way to the structure of the physical device imagery for constraints.

1.3. Overview of the Dissertation

This overview has two sections. One describes how this document was *supposed* to be organized (and there are reasons for describing this, for it provides perspective on what was done and part of what remains to be done). The other section of course describes how it *is* organized.

1.3.1. The Author Had a Grand Program for Solving the Entire Problem

When I set out to pursue this research, I had a plan, as many do. This dissertation was to have been divided into four parts with the following outline (this is *not* the actual outline for this dissertation), in which each italicized heading represents the title for one chapter:

Original (Not Current) Outline

Part I. Constraints.

Propagation: implementing adirectional devices which propagate values.

Dependencies: recording computation histories and giving explanations.

Retraction: using dependencies to resolve conflicts.

Assumptions: limited non-monotonic computation based on guesses.

Graphics: drawing constraint networks; constraints on graphical objects.

Tables: handling compound objects such as arrays, whose values may be only partly known.

Part II. Hierarchy.

Abstraction: packaging networks to look like single constraint devices.

Closures: devices as data objects; constraints on constraints; meta-circularity.

Lemmas: using hierarchy to guide the production of explanations.

Part III. Algebra

Notation: an abbreviated nested expression notation.

Slices: aiding propagation through multiple redundant points of view.

Transformations: pattern-directed invocation; automatic network augmentation; loop-breaking.

Part IV. Efficiency

Control: explicit control; propagation of desires; meta-constraints; heuristic assumptions.

Specialization: case-splitting in the primitives to handle common situations quickly.

Compilation: producing primitive devices from network specifications.

Reclamation: garbage collection on the network; reclaiming reconstructible histories.

(I do not expect the reader to comprehend the complete significance of all cryptic notes above. They are explained later in the dissertation.)

As the research progressed, however, it became clear that within imposed time limits I had the choice of examining all of these topics in a cursory manner, or exploring a subset of them thoroughly. I chose the second option.

1.3.2. The Author Settled for Doing Half Thoroughly Rather Than All Poorly

This dissertation does not by any means encompass all of the material in the preceding outline, but that which is covered here is covered thoroughly. All but the last chapter (Conclusions) concerns existing constraint systems that have been demonstrated to work. All the code for all

these systems is included in this dissertation and documented in the text.[6] Not all the text concerns low-level details of the code, however. Each chapter is typically split into a high-level discussion of issues, and a low-level discussion of implementation. I have attempted to arrange the text so that the reader may read the entire dissertation; or just the English text and not the code; or just the text, and skipping those sections which contain code. All sample computer input and output appearing in the text are actual transcripts of the operation of the systems presented and documented in the text.

This dissertation is divided into three parts. A condensed outline and a summary of each part and chapter follow. The arrangement of the material is vaguely similar to the originally proposed outline. The material in the above outline which does not appear below is discussed at some length in the chapter on Conclusions.

Brief Outline of Dissertation

Part I. Constraints. *Propagation. Dependencies. Retraction. Assumptions.*
Part II. Engineering. *Efficiency. Correctness.*
Part III. Abstraction. *Hierarchy. Compilation.*

Full Outline of Dissertation

Part I. Constraints. In this part a constraint language is defined incrementally and the system for executing it implemented by stages. Each chapter builds on the work of the previous one, until by the end of the part a moderately sophisticated constraint system has been constructed.

Propagation. A minimal toy constraint language is defined; it permits the statement of equalities and some simple arithmetic relationships. An implementation representation is chosen, and LISP code for a constraint interpreter is presented. Sample runs of a trivial constraint program are exhibited, and some problems and deficiencies discussed.

Dependencies. Mechanisms are introduced for recording the history of a computation. Utility procedures for extracting explanations from computation histories are demonstrated. Ways of using the network as a symbolic (algebraic) representation of a quantity are discussed.

Retraction. Conflicts can arise in a network in a number of ways; all are a consequence of global properties of the network. Hence a global process, dependency-directed backtracking, must be used to determine the precise causes of a conflict. Means of choosing which premise to retract are considered.

Assumptions. Constructs are introduced for advising the system on when to make assumptions or "educated guesses" about the value of some quantity. Such guesses may be inconsistent because of global considerations, and so *nogood sets* are introduced as a mechanism for recording in a locally accessible way the global reason for forbidding a guess. An implementation of assumption

6. The code is written in Lisp Machine LISP [Weinreb 1979], a dialect of LISP descended from MACLISP [Moon 1974]. Constructs which are peculiar to this dialect are described along the way.

mechanisms and automatic retraction of incorrect assumptions is presented. A large example (the n queens problem) is discussed and solved using a constraint program, which is shown to be potentially a more efficient technique than the usual chronological backtracking method.

Part II. Engineering. The first part is concerned primarily with language definition and a clear and simple implementation which demonstrates the concepts involved. However, that implementation is not particularly efficient, and is not obviously correct. This part contains a complete re-implementation of the same language, with issues of efficiency and correctness in mind. The entire state of the system is made explicit as data structures, rather than letting part be implicit in the program state of the LISP code which implements the system.

Efficiency. A complete re-implementation is presented of the language defined in the first part. Multiple reasons for a believing a value are explicitly recorded. The computation rules are pre-catalogued to permit efficient dispatching. A queue-based control structure is introduced; the queues contain tasks to be scheduled, and most tasks compute for only a limited time, enqueuing other tasks. While priority ordering of tasks is introduced for efficiency, the tasks may be correctly scheduled in any order. Effort is expended to make it possible to characterize the state of the system at the time a new task is to be selected; this is intended to ease a demonstration of correctness. The state of the system when contradictions are outstanding is still well-defined, and both explanation procedures and modifications to the network are designed to operate correctly even when the existing network contains contradictions. The queue-based structure is similar to that used by multi-programming schedulers, and is intended to mimic the standard single-processor simulation of a multi-processor system, thereby making it easier to transfer the ideas to a true multi-processor implementation.

Correctness. This chapter contains *no programs.* It reflects on the implementation of the previous chapter. While no attempt is made to provide a rigorous proof of the implementation, a large number of the necessary invariants are presented to sketch a possible approach to a proof. The program is *very large* and would take considerable effort to prove rigorously. However, attention to the intended invariants stated here certainly aided the implementation process and served to detect many difficulties.

Part III. Abstraction The language developed in the first two parts has primitive devices and a means of combining them, but no means of abstraction for packaging up a combination to make it look like a primitive device. In this part we define a macro mechanism for this purpose.

Hierarchy. The "flatness" of the language is relieved by introducing hierarchy in two ways. One is a macro mechanism by which a network can be packaged up and made to look like a primitive device; this induces a macro-call hierarchy. The other is a parser for a generalized nested algebraic notation, so that arithmetic expressions of roughly the usual sort can be used to notate constraint networks. The parser also implements convenient abbreviations.

Compilation. When a macro device is instantiated, a copy of the defining network is produced

to perform the actual computations. Hence there are no computation rules associated with macro definitions, but only with true primitive devices. In this chapter a compiler is described which from the network for a macro deduces all possible computation rules of interest and constructs a definition for an equivalent true primitive device.

Conclusions. The research described in the dissertation is summarized. Tentative results not contained in the dissertation proper are discussed, as well as foreseeable extensions to this work. Work by other researchers is discussed and compared with this research.

[This page intentionally left blank.]

Part One

Constraints

While constraints are superficially similar to fish, in actuality they are more closely related to alligators: they snap up their inputs greedily. Scaly green alligators swim lazily among the cypresses, evoking images of astronauts working in in silent, black space, their feet (almost non-existent in the case of alligators) dangling in whatever direction chance occasions, inasmuch as gravity is of little relevance in the void.

In space, as elsewhere, except in swamps, and other places which are also exceptions to the rule, there are, generally speaking, no alligators, or for that matter their ostensible and ostentatious cousins, the ferocious (so people seem to think, in their dreams and fantasies, although I must say I cannot personally vouch for this notion as a hard and established fact) crocodiles. This is, however, a subject for fierce debate.

—Anonymous

The meanin' of the mornin'
'Midst the moanin' of the moon
Bespeaks a specious speck of speech
That quarters past the Noon.
—Walt Kelly (1952)
I Go Pogo

Chapter Two

Propagation

THE CENTRAL IDEA behind constraint-based systems is the notion of *local propagation*—that a number of small processes arranged in a network, each processing information locally and sending results on to neighbors, can cooperatively produce a useful global effect. The direction of computation is determined dynamically; a constraint attempts to enforce a relationship among several parameters without any prejudice as to which are inputs and which outputs. It is willing to compute any parameter from others when those others have been determined.

In this chapter we introduce a trivial constraint language. This language is exceedingly weak, and hardly useful for practical purposes. It is intended as a toy for didactic purposes. It has purposely been pared to the bone, stripped of all features not directly needed to illustrate the principle of local propagation. The implementation of this language is likewise trivial, and consequently suffers certain inefficiencies (which will be remedied in later chapters).

2.1. A Trivial Constraint Language

The data objects of our language are the integers.

It is possible to speak of an object without knowing precisely what it is by using a name for it. Such a name is a variable. A variable can be declared so:

```
(variable x)
```

Then `x` is understood to denote an integer, though which integer it is may not yet have been computed. If the name `x` is mentioned later, it is understood to refer to this variable.

An integer constant may be explicitly mentioned in the language by using the `constant` construct:

```
(constant 43)
```

In effect this declares an anonymous variable and also declares that the object named by this nameless variable is the integer 43. Of course, this is not very useful by itself; because the variable has no name, it cannot be referred to later. However, the form `(constant 43)` itself serves as a denotation of the variable, as will be seen.

Two variables may be declared to denote the same object by using the `==` declaration:

```
(== x y)
```

As we shall see, the computational effect of this will be that when a value is computed for one variable, that will also become the value of the other variable. As a special case, one can assign a specific value to a variable by equating it to a constant:

```
(== x (constant 43))
```

This states that the value of `x` is 43 (and also that the value of `y` is 43, since `(== x y)` is in effect).

One can also state more complex relationships among variables by using constraints. We draw a constraint relationship as if it were a little TTL device.[1] Logic devices, however, "compute" in only one direction—some pins only accept inputs and some only produce outputs—but our constraint boxes generally treat each "pin" as bidirectional. Each pin of a constraint device is a variable: it has a name, and can be equated to other variables.

Our language provides an assortment of devices for stating relationships among integers. In describing them, we list the name of the constraint type, the names of the pins, and the relationship enforced by the constraint. The pictures we will use for these constraints appear in Figure 2-1.

`adder {a, b, c}`	$c = a + b$ (alternatively, $a = c - b$ or $b = c - a$).
`multiplier {a, b, c}`	$c = a \times b$ (alternatively, $a = c/b$ or $b = c/a$). Note that c/b is not defined in this language if $b = 0$ *or* if c/b is not an integer. (When $b = 0$ then c/b is many-valued (indeterminate) when $c = 0$, and no-valued (contradictory) when $c \neq 0$.)
`maxer {a, b, c}`	$c = \max(a, b)$.

1. Indeed, the inspiration for this computational paradigm was the mental imagery associated with electrical circuits. [Sussman 1975] [Stallman 1977]

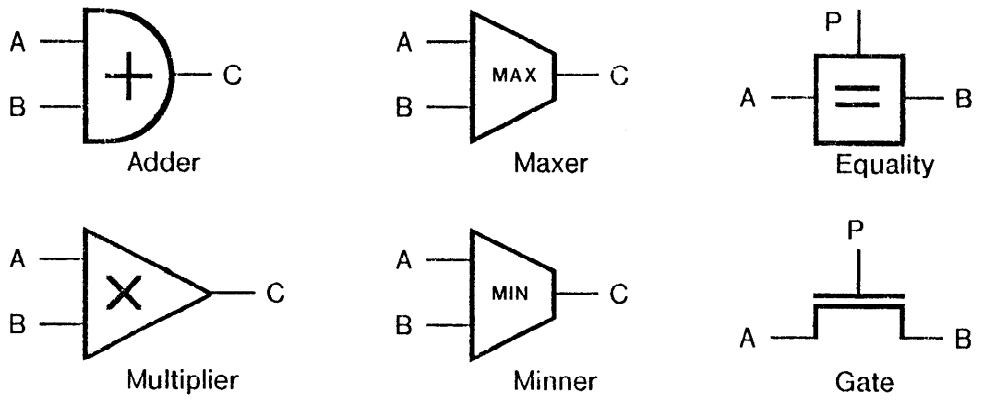


FIGURE 2-1. Primitive Constraint Devices on Integers.

`minner {a, b, c}` $c = \min(a, b)$.

`equality {p, a, b}` $p \Leftrightarrow (a = b)$, where the truth value for p is represented by 0 for *false* or 1 for *true*, as in APL.

`gate {p, a, b}` $p \Rightarrow (a = b)$ (alternatively, $(a \neq b) \Rightarrow \sim p$).

No primitive is provided in the language for subtraction, because that is simply another way of viewing an addition constraint; similarly for division. More generally, a single constraint box can represent a given relationship and also all of its inversions. For example, a single exponentiation box could represent all of $x = y^z$, $z = \log_y x$, and $y = \sqrt[z]{x}$.

ItiS InTeReStInG TocOnSider the inversions of other operators as well—certainly they must be considered in order to provide a complete implementation of a constraint box for that operator. For example, what is the inversion of $c = \max(a, b)$ which finds b given a and c? Let us denote this by $\text{arcmax}_c a$. Then we can provide the following definition:

$$\text{arcmax}_c a = \begin{cases} c & \text{if } a < c \\ \text{unknown} & \text{if } a = c \\ \text{error} & \text{if } a > c \end{cases}$$

Note that sometimes the value cannot be computed because the inputs are inconsistent (as in the case of dividing by zero), and so there is no consistent value. At other times the inverse may have multiple consistent values, and so no unique result can be computed. This occurs for $\text{arcmax}_c a$ when $a = c$, for the result can be any integer not greater than c. A more familiar example is that the square root operation is double-valued for positive inputs. We will return to this subject later.

If we have several constraint boxes, we can "wire them together" by connecting their pins. Since each pin is a variable, two pins can be connected by equating them as variables. We indicate this in a diagram by drawing lines among the pins, according to the usual conventions of logic

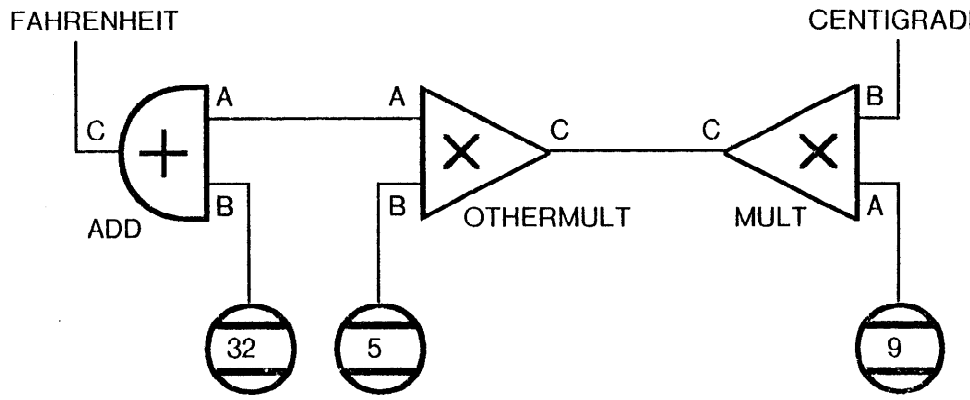


FIGURE 2-2. A Constraint Network for Converting Temperatures.

diagrams. Textually, we notate the interconnection of several constraints in two steps. First we declare and name instances of constraint devices:

```
(create add adder)
(create mult multiplier)
(create othermult multiplier)
```

This creates an adder named `add` and two multipliers named `mult` and `othermult`. Next we state the connections among the pins:

```
(== (the b add) (constant 32))
(== (the a add) (the a othermult))
(== (the c othermult) (the c mult))
(== (the b othermult) (constant 5))
(== (the a mult) (constant 9)))
```

We have used the `the` construct to refer to pins. The expression `(the x y)` refers to the pin named `x` of the device named `y`, and may be read "the `x` of `y`".

Let us also declare two variables `fahrenheit` and `centigrade` and connect them to (i.e., make them alternative names for) two pins:

```
(variable fahrenheit)
(variable centigrade)
(== fahrenheit (the c add))
(== centigrade (the b mult))
```

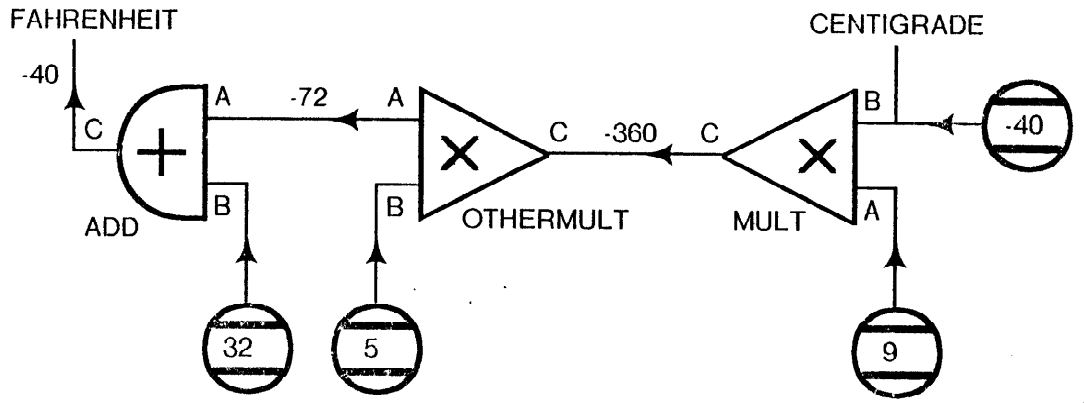


FIGURE 2-3. Computation of a Temperature Conversion.

The result is shown in Figure 2-2. This network in fact represents the familiar temperature conversion constraint between the variables `fahrenheit` and `centigrade`.[2]

$$\texttt{centigrade} = \frac{5 \times (\texttt{fahrenheit} - 32)}{9}$$

Suppose now, for example, that we state that `centigrade` is -40.

```
(== centigrade (constant -40))
```

This results in the following sequence of computations:

- From: `centigrade` = `(the b mult)` = -40
  `(the a mult)` = 9
  the constraint `mult` deduces `(the c mult)` = $-40 \times 9 = -360$.
- From: `(the c mult)` = `(the c othermult)` = -360
  `(the b othermult)` = 5
  the constraint `othermult` deduces `(the a othermult)` = $(-360)/5 = -72$.
- From: `(the a othermult)` = `(the a add)` = -72
  `(the b add)` = 32
  the constraint `add` deduces `(the c add)` = $(-72) + 32 = -40$.

This computation is pictured in Figure 2-3.

This computational technique is called *local propagation*. Each deduction is performed locally by a single primitive constraint device, from data immediately available to it. This results in a step-by-step propagation of values from one device to the next.

2. This example was borrowed in spirit from [Borning 1979].

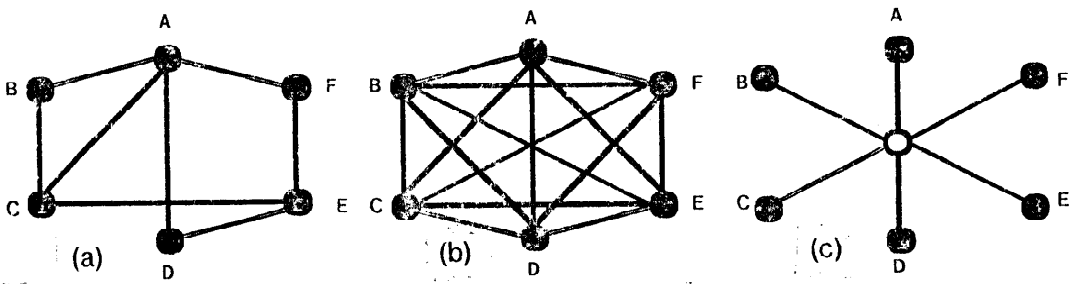


FIGURE 2-4. Some Organizations for Implementing Cells.

In summary, the statements permitted in our trivial constraint language are:

- `(create` *constraint-name constraint-type*`)`, to create a constraint instance.
- `(variable` *variable-name*`)`, to declare a global variable.
- `(==` *thing-1 thing-2*`)`, to equate two variables.

The forms that may be mentioned in a `==` statement are:

- *variable-name*, the name of a declared global variable.
- `(the` *pin-name constraint-name*`)`, which means the pin *pin-name* of the created constraint *constraint-name*.
- `(constant` *integer*`)`, which effectively means an anonymous variable with *integer* as its associated value.

The constraint-types provided by the language are `adder`, `multiplier`, `maxer`, `minner`, `equality`, and `gate`.

2.2. Implementation of a Trivial Constraint Language

Here we discuss a complete implementation of our trivial constraint language in LISP (more specifically, Lisp Machine LISP [Weinreb 1979]). First we describe the data structures used to represent variables and values; then the representation of constraints; after that, the "evaluation mechanism" which effects computation by local propagation; and finally, definitions of primitive constraints.

2.2.1. Cells are Used to Represent Variables

A *cell* is a data structure used to represent a variable. It is used not only to contain a value, but to record the equating of the variable to other variables.

Equating of variables is transitive. Whenever a value is determined for one variable, then all variables equated to it must also receive that value, and all variables equated to *them*, and so on. This fact constitutes a propagation requirement.[3] Variables transitively equated obviously form an equivalence class. This class can be organized in one of several ways for purposes of propagation.

(a) All equivalences are explicitly recorded. Each cell contains the set of all cells to which it has been directly equated; call this set its neighbors. When a cell receives a value, the value is propagated to its neighbors, which will recursively propagate it. (See Figure 2-4a.)

On a sequential machine this technique requires space linear in the number of equivalences (which may be anywhere from linear to quadratic in the number of cells); constant time to add an equivalence; and time linear in the number of equivalences (between linear and quadratic in the number of cells) to propagate a value throughout the class. A recursive propagation procedure is required.

(b) The transitive closure of the equivalence relationship is explicitly recorded. Each cell contains the set of all cells to which it is transitively equivalent. If an equivalence is added between two cells not of the same class, then every cell of each class must have all cells of the other class added to its set. When a cell receives a value, then it sends the value to each neighbor, but no recursive propagation is required. (See Figure 2-4b.)

On a sequential machine this requires space quadratic in the number of cells; time linear in the number of cells to add an equivalence; and time linear in the number of cells to propagate. The propagation procedure is simpler, however, being iterative.

(c) Note that in the previous technique all the sets of neighbors would be identical if a cell were considered its own neighbor. Therefore let this set be represented only once and be shared among all cells. Furthermore let the value not be propagated at all, but be stored in only one shared place. (See Figure 2-4c.)

On a sequential machine this requires space linear in the number of cells; time linear in the number of cells to add an equivalence (because a new set of neighbors must be constructed—the perhaps simplistic assumption here is that it takes linear time to take the union of two sets of neighbors); and constant time to propagate.

We choose the last option for reasons of performance and pedagogy.[4] (The time to create n equivalences is still quadratic in n (because each one can be linear in the number of equivalences already made). It would be possible to use even more clever techniques but we shall forego that here.)

Let us therefore define a cell to be a data structure with four components:

3. Equality is therefore a special kind of constraint which also performs local propagation of values. It could be represented in the same way as other constraints, but for the fact that we intend to use equality itself as the means for connection of devices. Hence equality must be handled specially to avoid infinite regress.

4. And perversity? We shall see that this cleverness makes things difficult later, when compound objects and dependencies are introduced. We shall then have to choose another representation.

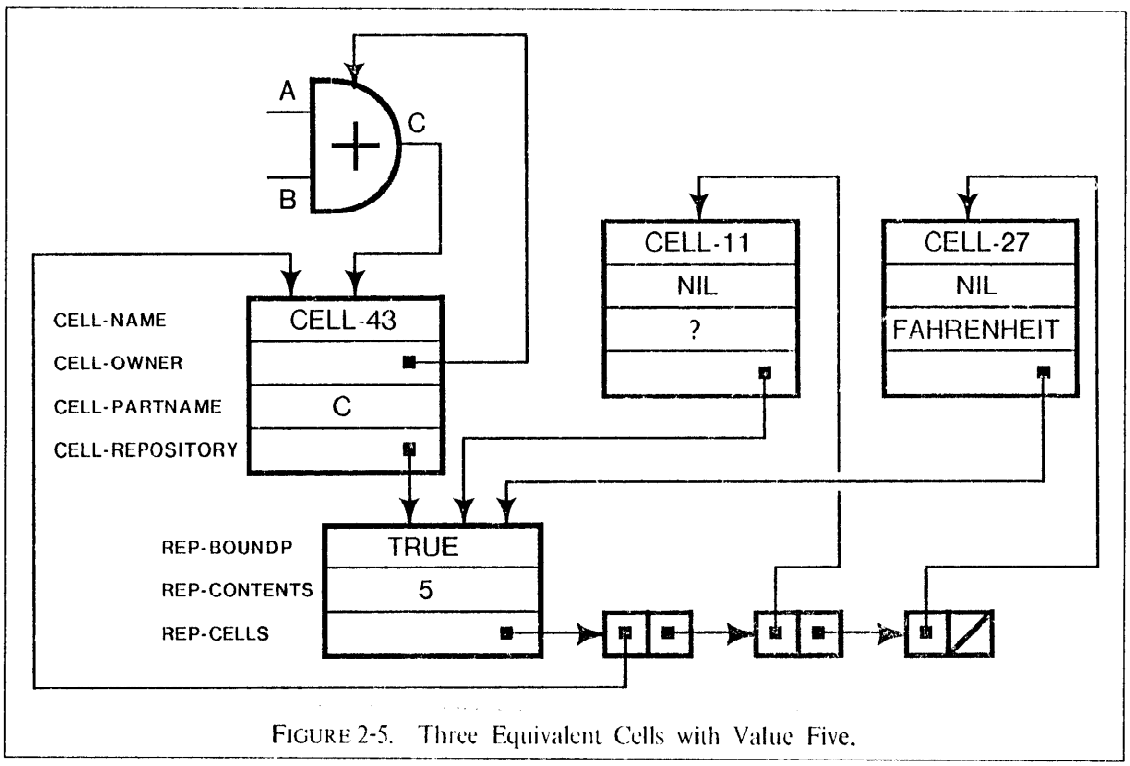


FIGURE 2-5. Three Equivalent Cells with Value Five.

(1) An identification *id*, used primarily for user meta-language interaction and debugging. (This component is not essential to the computational ability of the system.)

(2) An *owner*, which may be null (indicating that the cell represents a globally named variable), or may be a constraint (in which case the cell is a pin of that constraint).

(3) A *name*, which is a global name if the owner is null, or the pin name if the owner is a constraint.

(4) A *repository*, which is a data structure representing things shared with other cells.

Names of variables should be unique; hence all the cells with the same owner should have distinct names, and all the global cells should have distinct names. As an exceptional special case, all constants have a null owner and the name "?".

Similarly, we define a *repository* to be a data structure with three components:

(1) A flag *boundp*, indicating whether or not a value has been computed for this equivalence class of cells.[5]

(2) The *contents*, which is the computed value if the *boundp* flag is true.

5. Instead of having a separate flag, one could simply have a reserved value (`nil` or "unbound" or something) indicate the absence of an explicitly computed value. This technique can save space in an efficient implementation. We choose to use a flag here for clarity.

```
(deftype repository ((rep-contents ()) (rep-boundp ()) (rep-cells '()))
  (format stream "<Repository~:[~*~;: ~S~]~@[ for ~{~S~^,~}~]>"
          (rep-boundp repository)
          (rep-contents repository)
          (cell-ids repository)))

(defmacro node-contents (cell) `(rep-contents (cell-repository ,cell)))
(defmacro node-boundp (cell) `(rep-boundp (cell-repository ,cell)))
(defmacro node-cells (cell) `(rep-cells (cell-repository ,cell)))

(deftype cell (cell-id cell-repository cell-owner cell-name)
  (format stream "<~S~:[~2*~; (~S of ~S)~]: ~:[~*no value~;~S~]>"
          (cell-id cell)
          (cell-owner cell)
          (cell-name cell)
          (cell-owner cell)
          (node-boundp cell)
          (node-contents cell)))

(defun cell-ids (rep)
  (forlist (x (rep-cells rep)) (cell-id x)))
```

TABLE 2-1. LISP Code Defining Cell and Repository Data Types.

(3) The *cells*, a list of all cells which have this data structure as its repository. Thus this constitutes a set of back-pointers. From any cell all cells to which it is equivalent can be discovered.

Figure 2-5 shows three cells which have been equated and given the value 5. One cell is the `c` pin of an `adder` constraint; the one with name `?` is a constant (which was probably the source of the value 5); and the one with name `fahrenheit` is a globally named cell.

It is often convenient to consider a repository and all its associated cells to be a single object; we shall call such a collection a *node*, because it corresponds to a node of a network graph in the pictorial representation. A node is the conjunction of two or more arcs (edges, wires). Alternatively, a node represents an equivalence class of variables. We can draw a fine distinction by speaking of a value as being associated with a node or with a cell; the former case is a simple statement that all the cells of the node have the same value, but in the latter case we draw specific attention to an important relationship between the value and that particular cell of the node.

The LISP code in Table 2-1 defines `cell` and `repository` to be LISP user data types. Each `deftype` definition of the form

`(deftype` *name* (*component-1 component-2* ...) *printer*)

defines a new user data type called *name*. This will be a record-style data type with a fixed number of named components. It implicitly defines a number of functions to perform construction, selection, and predication for that type. Also, the method for printing objects of that data type is specified. Once the type definition above has been made:

- (`make-` *name*) will create and return a new data structure of type *name*.
- (*name*`-p` *x*) is a predicate true iff *x* is of type *name*.
- (`require-` *name* *x*) signals an error if *x* is not of type *name*. This is useful for error-checking and in-code documentation.
- Each *component-j* in the definition specifies the name of one record component. The *component-j* can be of either the form *cname* or the form (*cname* *initval*) . In either case *cname* is the name of a component of the data structure, and is the name of a selector function (actually a LISP macro) for extracting that component from an object. Thus (*cname* *x*) will return the contents of the *cname* component of the object *x*, which must be of type *name*. Moreover, the form (`setf` (*cname* *x*) *newval*) will change the *cname* component of *x* to be *newval*. If the first form for *component-j* is used, then the component value of a newly created instance of type *name* is undefined; otherwise, the component is initialized to *initval*.[6]
- The form *printer* is used by the LISP system to print objects of type *name*. Within the *printer* form the variable *name* names the object to be printed and the variable `stream` names the stream to which to send the output. The details of the `format` function are unimportant here; examples of printed objects will appear later.

As an example, the definition of `repository` in Table 2-1 defines the function `make-repository` of no arguments, which generates an object of type `repository` with three components named `rep-contents`, `rep-boundp`, and `rep-cells`. It also defines three functions `rep-contents`, `rep-boundp`, and `rep-cells` which extract conponents from objects of type `repository`. Saying `(setf (rep-contents x) 43)` takes the value of the LISP variable `x` (which must be a repository) and alters its `rep-contents` component to be 43. The function `repository-p` is a predicate which can be applied to any LISP datum, but is true only of repositories. The procedure `require-repository` signals an error if its argument is not a repository. Finally, a repository with no value and two cells in its list of cells might print as "`<Repository for CELL-34,CELL-36>`".

Table 2-1 defines not only the data types `cell` and `repository`, but also some extra macros for dealing with nodes. We do not define a separate LISP data type called `node`; instead, any cell of a node may serve to represent the node. The repository of a node holds data belonging to the node, and so the macros get the repository of the given cell and then extract the desired information from the repository. Thus, for example,

```
(node-contents x) → (rep-contents (cell-repository x))
```

and similarly for `node-boundp` and `node-cells`.

6. As a matter of programming style, I have written initialization forms iff the program depends on those initial values; components with no default values specified in the `deftype` declaration *must* be initialized by the program before being read. Another quirk of my programming style is that I write `()` to mean the constant "false" and `'()` to mean the constant "null list". Some LISP systems do not identify the null list with the atomic symbol `NIL`.

```
(defun gen-cell (&optional (name '?) (owner ()))
  (and owner (require-constraint owner))
  (let ((c (make-cell))
        (r (make-repository))
        (n (gen-name 'cell)))
    (setf (cell-id c) n)
    (set n c)
    (setf (cell-owner c) owner)
    (setf (cell-name c) name)
    (setf (cell-repository c) r)
    (push c (rep-cells r))
    c))

(defun constant (value)
  (let ((cell (gen-cell)))
    (setf (node-contents cell) value)
    (setf (node-boundp cell) t)
    cell))

(defmacro variable (name) `(setq ,name (gen-cell ',name)))
```

TABLE 2-2. Creation of Cells, Constants, and Variables.

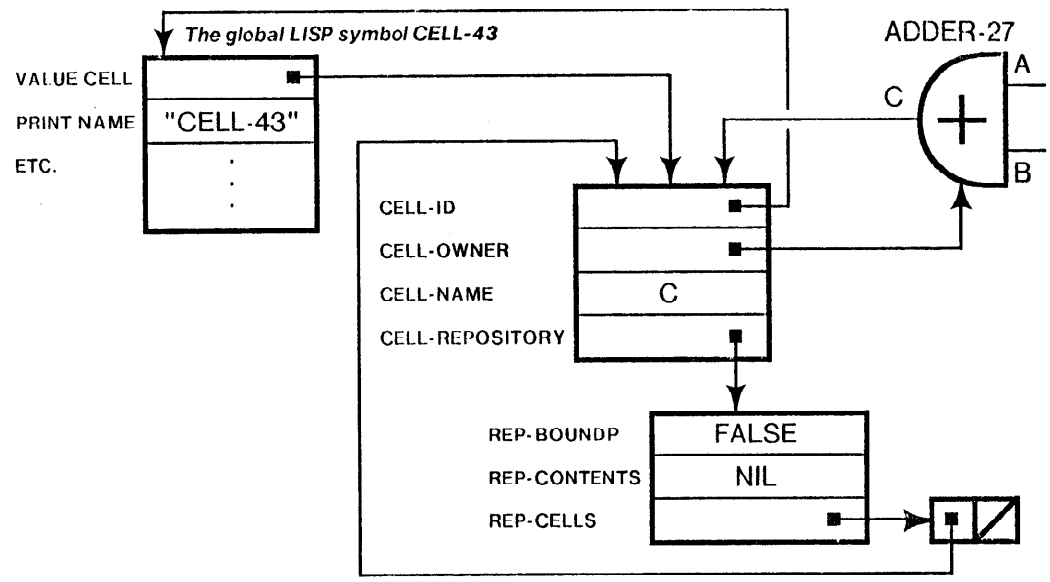


FIGURE 2-6. The Result of the Creating a Pin for an Adder.

Some utility procedures for generating new cells are shown in Table 2-2. The function `gen-cell` generates a new cell with given name and owner (if omitted, then the name is ? and the owner is null). A non-null owner must be a constraint (this is checked and enforced by the call to `require-constraint` in `gen-cell`). A new name of the form `cell-` *nnn*

is created for the cell; this is a (global) LISP variable which receives the cell as its value. This fact is of no consequence in the constraint computation proper, but is useful for meta-linguistic and debugging purposes. A repository is also created for the cell and linked up to it; thus every newly created cell constitutes a new node unto itself. Figure 2-6 shows the result of the call `(gen-cell 'c adder-27)`.

Saying `(constant 5)` will generate a new cell whose value is 5 (and whose boundp flag has been set true!).[7] Saying `(variable foo)` will cause the global LISP variable named `foo` to have as its value a cell whose global name is also `foo`.

2.2.2. Constraints are Instances of Constraint-Types

Just as in implementations of ordinary programming languages one conveniently divides a procedure into a constant part (the program text) and a variable part (parameters), so it will be convenient to split off the constant part of a constraint. We will call this the "constraint-type". A constraint-type contains the "program text" (rules for computation in various circumstances), its own name (for identification purposes), and the names of parameters.

Any given instance of this constraint-type we call a constraint. Such an instance refers to its constraint-type for the sake of the constant information the latter contains. The constraint also has a list of parameter "values" which correspond to the parameter names in the constraint-type. These "values" are actually cells used to contain the values. Finally, each constraint has a generated id and a global user name in the same way each cell does.

Table 2-3 gives definitions for the data structures `constraint-type` and `constraint`. The function `gen-constraint` creates an instance of a given constraint-type. It invents a name for the constraint instance; if the name of the constraint-type is `bazola`, then the name of the instance will be `bazola-` *nnn*. It also creates new parameters cells to serve as pins. Figure 2-7 shows the data-structure representation of an adder.

In Table 2-4 is the code for implementing the `the` construct for referring to the pins of a constraint. The implementation is in layers. First, the macro `the` is purely a bit of syntactic sugar for a call to the functon `*the`, to avoid having to write a quote mark.[8] The function `*the` in turn calls `lookup` to do the real work, signalling an error if `lookup` fails to locate the pin. Now

7. The reason for using the `constant` construct in our constraint language is purely pragmatic: we wish to use the standard LISP evaluator to do as much work as possible for us, by conforming to a few restrictions we can arrange for all construct of our language to be executable LISP code with the proper effect; this saves us the work of writing our own language interpreter (in much the same way that conforming to the usual parenthetical LISP syntax saves us the work of writing an input parser, because we can simply use the standard LISP function `read`). If we were willing to write our own interpreter, then that interpreter could unambiguously interpret an integer to mean a cell containing that integer, for example.

8. This is an example of wanting LISP to do the work without fully accommodating LISP syntax. Macros allow a slight bending of the rules.

```
(deftype constraint-type (ctype-name ctype-vars (ctype-rules '()))
  (format stream "<Constraint-type ~S>" (ctype-name constraint-type)))

(deftype constraint (con-id con-name con-ctype con-values)
  (format stream "<~@[~S:~]~S>" (con-name constraint) (con-id constraint)))

(defmacro create (name type) `(setq ,name (gen-constraint ,type ',name)))

(defun gen-constraint (ctype)
  (require-constraint-type ctype)
  (let ((c (make-constraint))
        (n (gen-name (ctype-name ctype))))
    (setf (con-id c) n)
    (set n c)
    (setf (con-name c) name)
    (setf (con-ctype c) ctype)
    (setf (con-values c)
          (forlist (var (ctype-vars ctype))
            (gen-cell var c)))
    c))
```

TABLE 2-3. Constraints and Constraint-Types.

```
(defmacro the (x y) `(*the ',x ,y))

(defun *the (name con)
  (require-constraint con)
  (or (lookup name con) (lose "~S has no part named ~S." con name)))

(defun lookup (name thing)
  (require-constraint thing)
  (do ((names (ctype-vars (con-ctype thing)) (cdr names))
       (cells (con-values thing) (cdr cells)))
      ((null names) ())
    (and (eq (car names) name) (return (car cells)))))
```

TABLE 2-4. Referring to Pins of a Constraint Device.

`lookup` merely searches the list of parameter names in the constraint-type of the constraint, and if the given name is found it returns the corresponding cell.

2.2.3. Equating of Cells Links Them and Propagates Values

In §2.2.1 we saw that every newly created cell has its own associated repository, and so is a minimal-size node. More generally, of course, every cell must always have a repository, which may be shared with other cells to form a larger node.

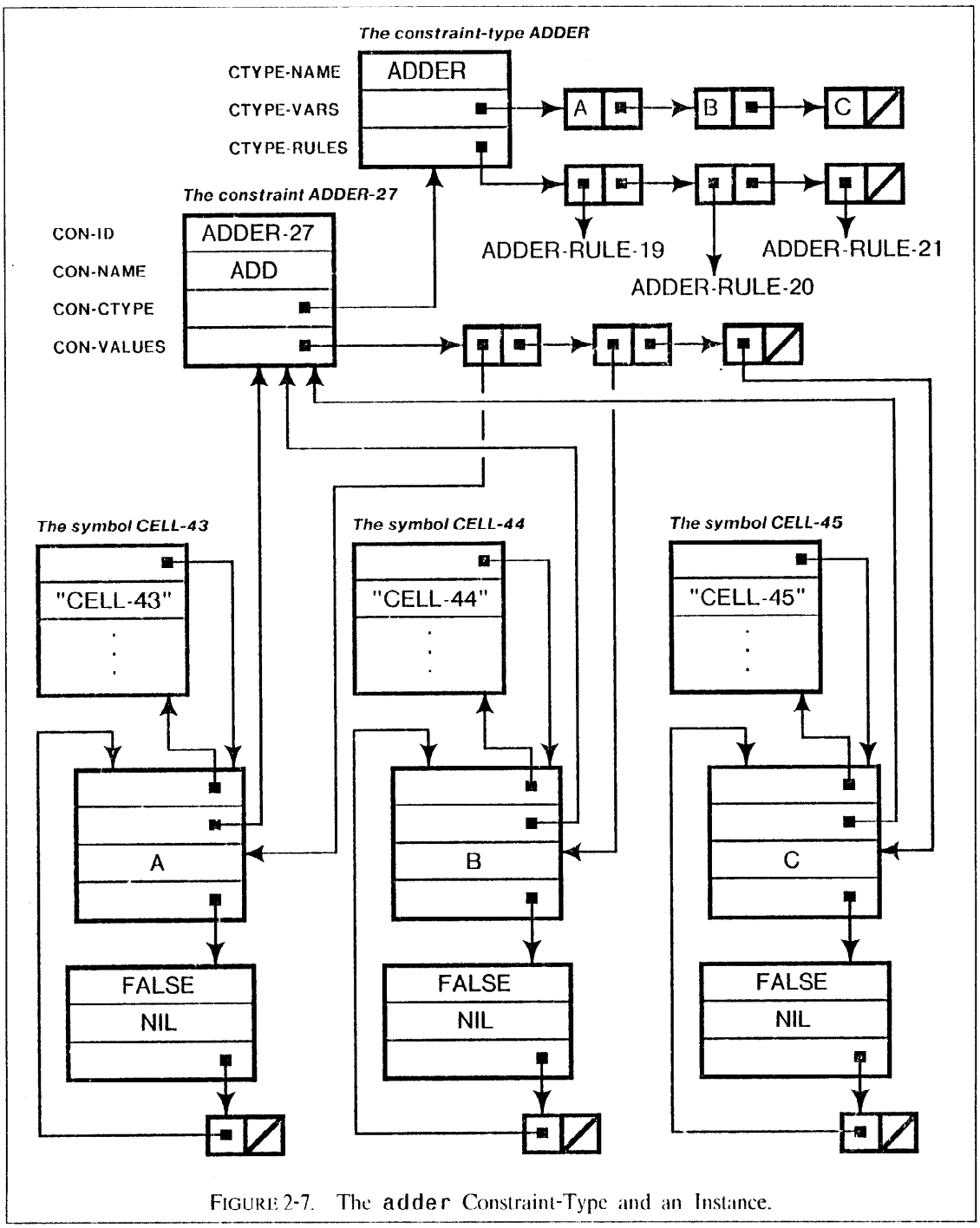


FIGURE 2-7. The `adder` Constraint-Type and an Instance.

When two cells are equated, then one repository is no longer needed. For simplicity and symmetry we shall simply throw them both away and create a new one, about which a node containing all the cells of the two input nodes is built.

```
(defun == (cell1 cell2)
  (require-cell cell1)
  (require-cell cell2)
  (or (eq (cell-repository cell1) (cell-repository cell2))
      (let ((r (make-repository))
            (cb1 (node-boundp cell1))
            (cb2 (node-boundp cell2)))
        (setf (rep-boundp r) (or cb1 cb2))
        (setf (rep-contents r) (merge-values cell1 cell2))
        (setf (rep-cells r) (append (node-cells cell1) (node-cells cell2)))
        (let ((newcomers (if cb1
                             (if cb2 '() (node-cells cell2))
                             (if cb2 (node-cells cell1) '()))))
          (dolist (cell (rep-cells r))
            (setf (cell-repository cell) r))
          (dolist (cell newcomers)
            (cond ((cell-owner cell)
                   (ctrace "Awakening ~S because its ~S got the value ~S."
                           (cell-owner cell)
                           (cell-name cell)
                           (rep-contents r))
                   (awaken (cell-owner cell)))))
          'done))))

(defun merge-values (cell1 cell2)
  (require-cell cell1)
  (require-cell cell2)
  (let ((val1 (node-contents cell1))
        (val2 (node-contents cell2)))
    (cond ((not (node-boundp cell1)) val2)
          ((not (node-boundp cell2)) val1)
          ((equal val1 val2) val1)
          (t (lose "Contradiction between ~S and ~S." cell1 cell2)))))

(defun awaken (con)
  (require-constraint con)
  (dolist (rule (ctype-rules (con-ctype con)))
    (funcall rule con)))
```

TABLE 2-5. Equating of Cells and Propagation of Values.

Table 2-5 shows how two cells are equated. The function `==` takes two cells, and if they are not yet equivalent it creates a new common repository for them. This repository will have a value if *either* of the input nodes had a value. Moreover, if both nodes had values, then when they are equated the values must *coincide*. The contents for the new repository are calculated by the function `merge-values`, which merely decides which node's value to use, and if both have values checks that they are equal, signaling a contradiction otherwise.[9] The new repository's set of cells is the union (which must in fact be a disjoint union) of the sets of cells for the two cells' repositories. The *newcomers* are defined to be those cells which formerly had no value but will

9. When new features are added to the language later, `merge-values` will have the more complicated task of merging two structured objects, taking some attributes from each value.

```
(declare (special *tracing*))
(setq *tracing* t)
(defun trace-on () (setq *tracing* t))
(defun trace-off () (setq *tracing* ()))
(defmacro ctrace (string . args)
  `(and *tracing* (format t "~%;|~10{~:}" ,string ,@args)))
```

TABLE 2-6. A Simple Tracing Mechanism.

now have a value because of the new equivalence. There can be newcomers only if exactly one node had a value, in which case the cells of the other node are the newcomers.[10] After all the cells involved are hooked up to the new repository, the owner (if any) of each cell is *awakened.* The function `awaken` when applied to a constraint runs all the rules associated with that constraint, telling each rule which constraint instance was involved (because the rules of a constraint-type are shared among all instances).

The `ctrace` statement is included purely for debugging purposes. It prints a formatted message so that the inner workings of the system can be traced. The LISP code for `ctrace` is shown in Table 2-6. The arguments to `format` are rather cryptic, but an obvious feature of the `ctrace` facility is that it can be turned off! The functions `trace-on` and `trace-off` set and clear the global tracing flag.

2.2.4. Constraints Are Implemented as Sets of Rules

The implementation of primitive constraint devices is best seen by example. Table 2-7 contains the LISP code for the devices whose symbols appeared in Figure 2-1. Each is expressed in terms of a special macro `defprim`:

```
(defprim name pin-names
         ( input-pins-1 rule-body-1)
         ( input-pins-2 rule-body-2)
         ...
         ( input-pins-n rule-body-n))
```

This defines a constraint-type called *name* which has parameter names *pin-names.* Each rule has a list of input pins and a piece of code[11] to execute when those pins all have values. (This restriction is simply a convenient filter which all rules desire. Recall that the function `awaken` given in Table 2-5 runs *all* the rules for a constraint. We shall see that `defprim` provides the code to check for

10. Again, when the task of `merge-values` will be to merge two structured objects, the cells of *both* nodes may be newcomers. We will see this later.

11. This code is not written in the constraint language, but in the implementation language; these definitions are for *primitive* constraints. Later we shall consider the definition of non-primitive constraints.

```
(defprim adder (a b c)
  ((a b) (setc c (+ a b)))
  ((a c) (setc b (- c a)))
  ((b c) (setc a (- c b))))

(defprim multiplier (a b c)
  ((a) (and (zerop a) (setc c 0)))
  ((b) (and (zerop b) (setc c 0)))
  ((a b) (setc c (* a b)))
  ((a c) (and (not (zerop a)) (zerop (\ c a)) (setc b (// c a))))
  ((b c) (and (not (zerop b)) (zerop (\ c b)) (setc a (// c b)))))

(defprim maxer (a b c)
  ((a b) (setc c (max a b)))
  ((a c) (cond ((< a c) (setc b c))
               ((> a c) (contradiction a c))))
  ((b c) (cond ((< b c) (setc a c))
               ((> b c) (contradiction b c)))))

(defprim minner (a b c)
  ((a b) (setc c (min a b)))
  ((a c) (cond ((> a c) (setc b c))
               ((< a c) (contradiction a c))))
  ((b c) (cond ((> b c) (setc a c))
               ((< b c) (contradiction b c)))))

(defprim equality (p a b)
  ((p) (or (= p 0) (= p 1) (contradiction p)))
  ((a b) (setc p (if (= a b) 1 0)))
  ((p a) (and (= p 1) (setc b a)))
  ((p b) (and (= p 1) (setc a b))))

(defprim gate (p a b)
  ((p) (or (= p 0) (= p 1) (contradiction p)))
  ((a b) (or (= a b) (setc p 0)))
  ((p a) (and (= p 1) (setc b a)))
  ((p b) (and (= p 1) (setc a b))))
```

TABLE 2-7. Implementation of the Constraint Boxes of Figure 2-1.

each input cell having a value.) Thus we implement our non-directional constraint boxes in terms of a directional language such as LISP.

Consider the definition of the `adder` constraint. It specifies three rules. When any two values are known, the third can be computed by the rules $c \leftarrow a + b$, $b \leftarrow c - a$, and $a \leftarrow c - b$. The form `(setc c (+ a b))` means "set cell `c` to the value of `(+ a b)`".

Look now at the definition of `multiplier`. It has rules which compute new values conditionally. For example, a value for `c` can be computed from `a` alone provided that `a` is zero. Similarly, computing `b` from `a` and `c` is conditional on being able to express the result as an integer (that is, the remainder `(\ c a)` must be zero). The LISP value produced by the rule-body computation is ignored; only the `setc` construction specifies new values for cells.

Next reflect upon the definition of `maxer`. When `a` and `c` are known, then as we saw in §2.1 there are three cases for computing $\text{arcmax}_c\, a$. If $a < c$ then $b \leftarrow c$; if $a = c$ then b is unknown; and if $a > c$ then it is not a matter of computing b at all: it is simply a contradiction, a violation of the constraint. This is all expressed in the second rule for `maxer`(the case $a = c$ implicitly holds if both `cond` tests fail). The form `(contradiction a c)` signals that a contradiction has occurred, and that the values of cells `a` and `c` are at fault.

---

*Digression.*There is another implementation strategy which does not require the use of `setc` at all, which was used in the constraint system reported in [Steele 1979]: that paper did not describe the technique, however, and therefore it is outlined here. It is assumed that each rule computes a value for exactly one cell, and that this value is computed by the rule-body; thus the LISP value of the rule-body actually *is* used. Each rule is therefore defined by a list of input cells, a rule-body, *and* an output cell. There are two global variables `*lose*` and `*dismiss*`, whose values are distinguishable from any value normally computed by a rule. If the value of a rule-body is that of `*lose*`, then a contradiction is signalled (the assumption being that it is precisely all the input cells that are at fault). If the value is that of `*dismiss*`, then no value is specified into the output cell. The definition of `maxer` using this technique would be:

```
(defprim maxer (a b c)
  (c (a b) (max a b))
  (b (a c) (cond ((< a c) c)
                 ((> a c) *lose*)
                 (t *dismiss*)))
  (a (b c) (cond ((< b c) c)
                 ((> b c) *lose*)
                 (t *dismiss*))))
```

One advantage of this technique is that one is required to cover all cases explicitly. On the other hand, it may require duplication of code if the same tests are used in rules for setting more than one cell (which, however, is not the case for the constraints of Table 2-7). Also, each rule is *required* to specify an output pin, which for some rules may be irrelevant. Consider the first rule of `equality` in Table 2-7, for example; it merely performs a consistency check on the pin `p`. It never computes a new value; the only possible outcomes are contradiction or dismissal.

This techique is useful in some situations, however, and we will use a variant of it in a later implementation. For now, however, the use of `setc` seems more instructive and intuitively appealing.

Note that the two values `*dismiss*` and `*lose*` of this technique may be interpreted to mean $\bot$ and $\top$, extra values adjoined to the value domain to represent under- and over-constrained values.

(*End of digression.*)

The LISP macro definition of the `defprim` construction (Table 2-8) is rather involved, but its effect is straightforward. It declares *name* to be a global LISP variable, and sets that variable to a newly created constraint-type data structure. The *name* and *pin-names* are installed in this structure, and then the rules are defined using the `defrule` construct. The `defrule` construct arranges for the LISP variable `*me*` to be bound to the constraint for which this rule is being invoked (the constraint which was awakened). The code for each rule binds variables named *pin-name*`-cell` for each pin of the constraint, and then checks to see that the input pins for that

```
(defmacro defprim (name vars . rules)
  `(progn 'compile
          (declare (special ,name))
          (setq ,name (make-constraint-type))
          (setf (ctype-name ,name) ',name)
          (setf (ctype-vars ,name) ',vars)
          ,@(forlist (rule rules)
              `(defrule ,name
                 (let ,(forlist (var vars)
                         `(,(symbolconc var "-CELL") (the ,var *me*)))
                   (and ,@(forlist (var (car rule))
                            `(node-boundp ,(symbolconc var "-CELL")))
                        (let ,(forlist (var (car rule))
                                `(,var (node-contents ,(symbolconc var "-CELL"))))
                          ,@(cdr rule))))))
          '(,name primitive)))

(defmacro defrule (typename . body)
  (let ((rulename (gen-name typename 'rule)))
    `(progn 'compile
            (push ',rulename (ctype-rules ,typename))
            (defun ,rulename (*me*) ,@body)
            '(,typename rule))))
```

TABLE 2-8. Definition of Primitive Constraints and Rules.

```
(progn 'compile
       (push 'equality-rule-23 (ctype-rules equality))
       (defun equality-rule-23 (*me*)
           (let ((a-cell (the a *me*))
                 (b-cell (the b *me*))
                 (p-cell (the p *me*)))
             (and (node-boundp a-cell)
                  (node-boundp b-cell)
                  (let ((a (node-contents a-cell))
                        (b (node-contents b-cell)))
                    (setc p (if (= a b) 1 0))))))
       '(equality rule))
```

TABLE 2-10. Expanded Second Rule of the `equality` Constraint.

rule are bound. If so, then the rule-body appearing in the `defprim` construct is executed. Table 2-9 shows the LISP code into which the call on `defprim` macro for `adder` expands.

The `defrule` construction simply generates a name for the rule, and defines a LISP function by that name. This function takes one argument, a constraint, calls it `*me*`, and executes the rule code. The name of the function is also added to the set of rules in the constraint-type. Table 2-10 shows the LISP code into which the second `defrule` in Table 2-9 expands.

Table 2-11 shows the implementation of `contradiction` and `setc`. An invocation of `contradiction` expands, for example, as:

```
(progn 'compile
       (declare (special equality))
       (setq equality (make-constraint-type))
       (setf (ctype-name equality) 'equality)
       (setf (ctype-vars equality) '(a b p))
       (defrule equality
                (let ((a-cell (the a *me*))
                      (b-cell (the b *me*))
                      (p-cell (the p *me*)))
                  (and (node-boundp p-cell)
                       (let ((p (node-contents p-cell)))
                         (or (= p 0) (= p 1) (contradiction p))))))
       (defrule equality
                (let ((a-cell (the a *me*))
                      (b-cell (the b *me*))
                      (p-cell (the p *me*)))
                  (and (node-boundp a-cell)
                       (node-boundp b-cell)
                       (let ((a (node-contents a-cell))
                             (b (node-contents b-cell)))
                         (setc p (if (= a b) 1 0))))))
       (defrule equality
                (let ((a-cell (the a *me*))
                      (b-cell (the b *me*))
                      (p-cell (the p *me*)))
                  (and (node-boundp p-cell)
                       (node-boundp a-cell)
                       (let ((p (node-contents p-cell))
                             (a (node-contents a-cell)))
                         (and (= p 1) (setc b a))))))
       (defrule equality
                (let ((a-cell (the a *me*))
                      (b-cell (the b *me*))
                      (p-cell (the p *me*)))
                  (and (node-boundp p-cell)
                       (node boundp b-cell)
                       (let ((p (node-contents p-cell))
                             (b (node-contents b-cell)))
                         (and (= p 1) (setc a b))))))
       '(equality primitive))
```

TABLE 2-9. Expanded Definition of the `equality` Constraint.

```
(contradiction a c) → (signal-contradiction *me* (list a-cell c-cell))
```

The function `signal-contradiction` causes an error, and prints information as to the source of the contradiction.

The `setc` construct is also implemented as a LISP macro. A call to `setc` expands, for example, as:

```
(setc c (+ a b)) → (process-setc *me* 'c c-cell (+ a b))
```

```
(defmacro contradiction vars
  `(signal-contradiction *me* (list ,@(forlist (v vars) (symbolconc v "-CELL")))))

(defun signal-contradiction (constraint cells)
  (require-constraint constraint)
  (lose "Contradiction in ~S~@[ among these pins: ~:{~S=~S~:*, ~}~]."
        constraint
        (forlist (cell cells)
          (require-cell cell)
          (list (cell-name cell) (node-contents cell)))))

(defmacro setc (cellname value)
  `(process-setc *me* ',cellname ,(symbolconc cellname "-CELL") ,value))

(defun process-setc (*me* name cell value)
  (require-constraint *me*)
  (require-cell cell)
  (ctrace "~S computed the value ~S for its ~S." *me* value name)
  (== cell (constant value)))
```

TABLE 2-11. Implementation of `contradiction` and `setc`.

The first and third arguments to `process-setc` are provided purely for the sake of the `ctrace` operation. The setting of a cell could be performed by forcibly inserting the value into the cell, but it is easier simply to create a constant cell containing the value and then equate it to the cell to be set. It is inefficient, in that an extra repository is created by `constant` and another by `==`. However, it lets the existing machinery in `==` do all the work of checking for contradictions. (We will fix this inefficiency in the next chapter.)

2.3. Sample Execution of a Constraint Program

Here we consider an interactive session with our trivial constraint language. We shall construct the temperature conversion network of Figure 2-2. User input appears in lower case, and the LISP value produced by this input appears in upper case. The `ctrace` statements in the code produce comment lines beginning with "`;|`".

First we create instances of the constraint devices we shall need, in this case an adder and two multipliers. The value returned by `create` is the data structure for the constraint, which prints as the unique name of the constraint, surrounded by angle brackets (thanks to the printing code which appears in the definition of the `constraint` data type in Table 2-3).

```
(create add adder)
<ADD:ADDER-20>
(create mult multiplier)
<MULT:MULTIPLIER-24>
(create othermult multiplier)
```

```
<OTHERMULT:MULTIPLIER-28>
```

The unique number (appended by the LISP function `gen-name`) is incremented by four each time because for each of these constraint instances three cells are also generated to serve as pins. We can refer to a pin by using the `the` construction.

```
(the a othermult)
<CELL-29 (A of OTHERMULT): no value>
(the b othermult)
<CELL-30 (B of OTHERMULT): no value>
(the c othermult)
<CELL-31 (C of OTHERMULT): no value>
```

The ability to do this interactively is not really part of our defined constraint language; it is, however, a decided convenience in interacting with the system. The fact that generated names contain numbers in increasing order is also irrelevant to the defined computational abilities of the system, but do aid in understanding in what order certain actions happen to occur. Note that when a cell is printed, the unique name and also the pin name and owner are printed, and also the value if any (the code which does this appears in Table 2-1). We did not define any input/output operators for our language, but the ability to examine cells interactively in this way will allow us to see the results of the computation.[12]

Next we declare that we will need two global variables `fahrenheit` and `centigrade`.

```
(variable fahrenheit)
<CELL-32 (FAHRENHEIT): no value>
(variable centigrade)
<CELL-33 (CENTIGRADE): no value>
```

Now each cell must have a repository. We can examine the repository of a cell.

```
(cell-repository fahrenheit)
<Repository for CELL-32>
```

We now wire the network together. We begin by equating `fahrenheit` to the `c` pin of the adder `add`.

```
(== fahrenheit (the c add))
DONE
```

12. All of these remarks of course have little to do with the design of a constraint language as such. Rather, they are intended to show how a toy system can be imbedded in a larger system (in this case a LISP system) with a minimum of work to get it off the ground just enough to exhibit a principle, without having to re-implement a host of trivial details (such as I/O). By arranging for the interpreter of the constraint language to be that of LISP, and that the forms of the constraint language are simple certain evaluable LISP forms, then when interacting with the system we can evaluate constraint forms or LISP forms at will. More abstractly, at any time we may shift freely from language to meta-language and back.

Examination of the repository of the cell `fahrenheit` reveals that it has been linked to another cell `cell-23`. This is the name of a cell which turns out (not very surprisingly) to be the `c` pin of `adder-20`, which is also called `add`.

```
(cell-repository fahrenheit)
<Repository for CELL-32,CELL-23>
cell-23
<CELL-23 (C of ADD): no value>
(the c add)
<CELL-23 (C of ADD): no value>
adder-20
<ADD:ADDER-20>
add
<ADD:ADDER-20>
```

Specifying a constant creates a cell with no owner and name "?".

```
(constant 32)
<CELL-34 (?): 32>
```

We now connect this constant to the `b` pin of the adder.[13]

```
(== (the b add) cell-34)
;|Awakening <ADD:ADDER-20> because its B got the value 32.
DONE
```

The `ctrace` statement in the definition of `==` (see Table 2-5) printed a comment indicating that one pin got a value and so all rules were being run. However, no rule of the `adder` constraint type can do anything with only one input.

If we examine the `b` pin of `add` we can see that it indeed also has the value 32.

```
(the b add)
<CELL-22 (B of ADD): 32>
```

Let us without further ado wire up the rest of the network of Figure 2-2. Two more `ctrace` comments are produced when the constants 5 and 9 are wired up.

```
(== (the a add) (the a othermult))
DONE
(== (the c othermult) (the c mult))
DONE
(== (the b othermult) (constant 5))
;|Awakening <OTHERMULT:MULTIPLIER-28> because its B got the value 5.
```

13. Of course, this statement is not properly part of the constraint language, but a mixture of the constraint language and its meta-language LISP (because the variable `cell-34` is part of the meta-language—indeed the very fact that we know of the existence of that name indicates that we have gone outside the constraint language and examined the internals of the implementation!).

```
DONE
(== centigrade (the b mult))
DONE
(== (the a mult) (constant 9))
;|Awakening <MULT:MULTIPLIER-24> because its A got the value 9.
DONE
```

The network, now completely wired, can be used to perform a computation. Here we will try the computation of Figure 2-3. The cell `centigrade` is equated to the constant —40.

```
(== centigrade (constant -40))
;|Awakening <MULT:MULTIPLIER-24> because its B got the value -40.
;|<MULT:MULTIPLIER-24> computed the value -360 for its C.
;|Awakening <OTHERMULT:MULTIPLIER-28> because its C got the value -360.
;|<OTHERMULT:MULTIPLIER-28> computed the value -72 for its A.
;|Awakening <ADD:ADDER-20> because its A got the value -72.
;|<ADD:ADDER-20> computed the value -40 for its C.
;|Awakening <ADD:ADDER-20> because its C got the value -40.
;|<ADD:ADDER-20> computed the value -72 for its A.
;|<ADD:ADDER-20> computed the value 32 for its B.
;|<ADD:ADDER-20> computed the value -40 for its C.
;|Awakening <OTHERMULT:MULTIPLIER-28> because its A got the value -72.
;|<OTHERMULT:MULTIPLIER-28> computed the value -72 for its A.
;|<OTHERMULT:MULTIPLIER-28> computed the value 5 for its B.
;|<OTHERMULT:MULTIPLIER-28> computed the value -360 for its C.
;|<OTHERMULT:MULTIPLIER-28> computed the value 5 for its B.
;|<OTHERMULT:MULTIPLIER-28> computed the value -360 for its C.
;|Awakening <MULT:MULTIPLIER-24> because its C got the value -360.
;|<MULT:MULTIPLIER-24> computed the value 9 for its A.
;|<MULT:MULTIPLIER-24> computed the value -40 for its B.
;|<MULT:MULTIPLIER-24> computed the value -360 for its C.
DONE
```

Note that each constraint device here has computed new values for its pins, including those pins which originally provided input values! For example, given `a` and `c` the adder computed a value for `b`—but once `b` was in hand, it could be used with `a` to compute `c`, and with `c` to compute `a`. Nothing detects the fact that the adder itself computed the value for `b`. On the other hand, the process does not iterate indefinitely (a common bug indeed when implementing this sort of thing!) because `==` does not run any rules when a value is equated to a cell which already has a value (because then the set of newcomers is empty).

If we now examine the cell `fahrenheit`, we see that indeed it has acquired the value —40.

```
fahrenheit
<CELL-32 (FAHRENHEIT): -40>
```

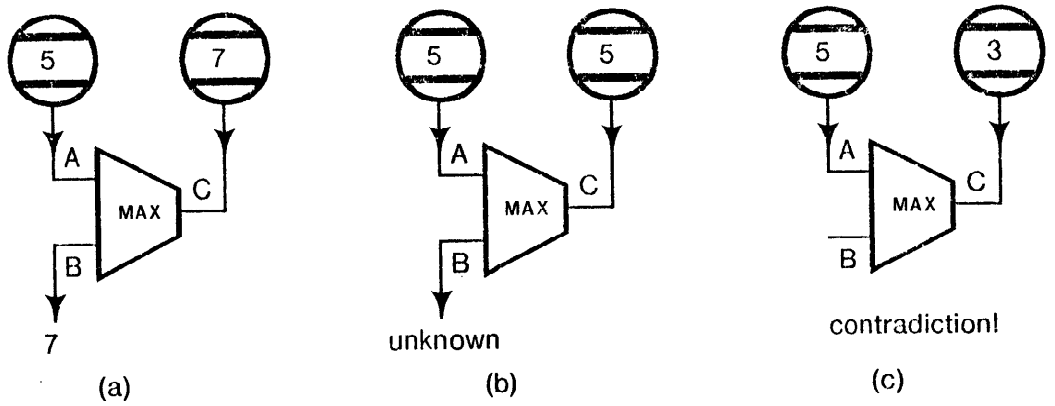


FIGURE 2-8. Operation of the `maxer` Constraint.

Suppose now that we attempt to set `fahrenheit` to 32. When `merge-values` gets the values —40 and 32, it finds that they are incompatible, and invokes `lose`.

```
(== fahrenheit (constant 32))
>>ERROR: Contradiction between <CELL-32 (FAHRENHEIT): -40>
         and <CELL-51 (?): 32>.
...
```

As another toy example to show off the `contradiction` mechanism, consider a `maxer` box with its `a` pin equated to 5. We will take three such boxes and equate their `c` pins to 7, 5, and 3, respectively.

```
(create m1 maxer)
<M1:MAXER-68>
(create m2 maxer)
<M2:MAXER-72>
(create m3 maxer)
<M3:MAXER-76>


(== (the a m1) (constant 5))
;|Awakening <M1:MAXER-68> because its A got the value 5.
DONE
(== (the a m2) (constant 5))
;|Awakening <M2:MAXER-72> because its A got the value 5.
DONE
(== (the a m3) (constant 5))
;|Awakening <M3:MAXER-76> because its A got the value 5.
DONE
```

From the values $a = 5$ and $c = 7$, `m1` can deduce $b = 7$. (See Figure 2-8a.)

```
(== (the c m1) (constant 7))
;|Awakening <M1:MAXER-68> because its C got the value 7.
;|<M1:MAXER-68> computed the value 7 for its B.
;|Awakening <M1:MAXER-68> because its B got the value 7.
;|<M1:MAXER-68> computed the value 7 for its B.
;|<M1:MAXER-68> computed the value 7 for its C.
;|<M1:MAXER-68> computed the value 7 for its C.
DONE
```

Note that values were computed for `b` and `c` *twice* each. This is because in this implementation when a value is received on *any* pin, *all* rules are fired. Since two pins got values, all rules are fired twice. It all settles down in the end, but is a source of inefficiency.

When $a = 5$ and $c = 5$, no specific value can be computed for b. All that is known is that $b \leq 5$. (See Figure 2-8b.)

```
(== (the c m2) (constant 5))
;|Awakening <M2:MAXER-72> because its C got the value 5.
DONE
```

When $a = 5$ and $c = 3$, we have an inconsistent situation. (See Figure 2-8c.)

```
(== (the c m3) (constant 3))
;|Awakening <M3:MAXER-76> because its C got the value 3.
>>ERROR: Contradiction in <M3:MAXER-76> among these pins: A=5, C=3.
```

The inconsistency has caused a fatal error. (Later we will see how such errors can be useful rather than fatal, and can cause the system to search for ways to resolve the problem.)

2.4. A Difficulty with Division

There is an intentional peculiarity in our definition of the `multiplier` primitive in Table 2-7, which is that if a division does not come out exactly it simply fails to compute a result. One might argue that since we have defined the data objects of our language to be the integers, then it is an error to try to divide, say, 7 by 3, because there is no object n such that $3 \times n = 7$.

Suppose that we construct another temperature conversion network as in §2.3, just before we assign the value -40 the `centigrade`. Let us see what happens if we instead equate `centigrade` to the constant 37.

```
(== centigrade (constant 37))
;|Awakening <MULT:MULTIPLIER-100> because its B got the value 37.
;|<MULT:MULTIPLIER-100> computed the value 333 for its C.
;|Awakening <OTHERMULT:MULTIPLIER-104> because its C got the value 333.
;|Awakening <MULT:MULTIPLIER-100> because its C got the value 333.
```

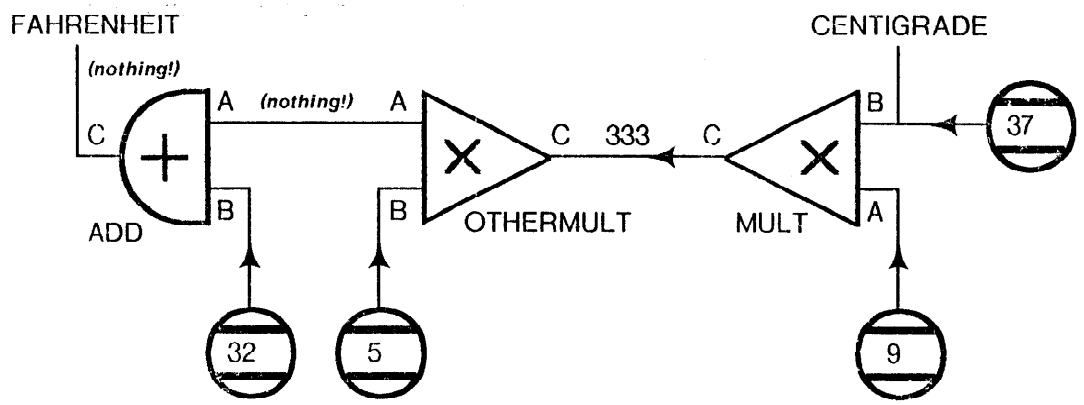


FIGURE 2-9. A Temperature Conversion Which "Failed".

```
;|<MULT:MULTIPLIER-100> computed the value 9 for its A.
;|<MULT:MULTIPLIER-100> computed the value 37 for its B.
;|<MULT:MULTIPLIER-100> computed the value 333 for its C.
DONE
```

It seems that `mult` performed some useful work, but `othermult` did not, and `add` was not even awakened. (See Figure 2-9.)

```
fahrenheit
<CELL-108 (FAHRENHEIT): no value>
```

Indeed `fahrenheit` has not had any value computed for it.

```
(the a othermult)
<CELL-105 (A of OTHERMULT): no value>
(the b othermult)
<CELL-106 (B of OTHERMULT): 5>
(the c othermult)
<CELL-107 (C of OTHERMULT): 333>
```

The constraint `othermult` has values for `b` and `c` , but cannot compute a value for `a` because the division 333/5 is not exact. However, we do find it useful in mathematics to extend the integers to the rational numbers, and say that there *is* an object which represents the result of this devision, even though we don't have any better way to represent it than as "333/5", that is to say, "that object which is the quotient of 333 and 5". If we examine the state of the network in Figure 2-9, we can see that this quotient is represented *by the network itself*, considered as a data structure. Moreover, the network represents the fact that `fahrenheit` is the difference between this quotient (whatever it is) and 32.

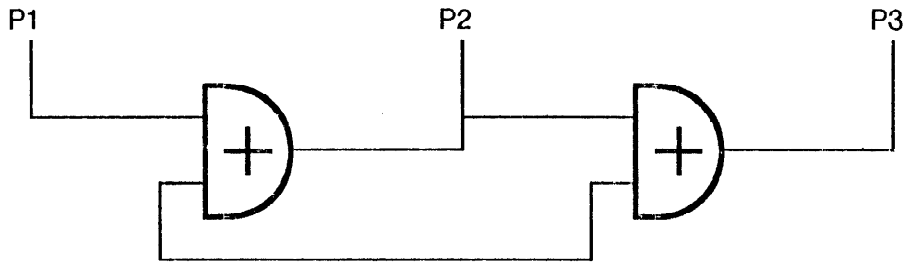


FIGURE 2-10. Constraining Three Points to be Equally Spaced (i).

We could introduce rational numbers as a primitive data type. Such an implementation would presumably use a LISP data structure to hold the numerator and denominator of a rational number, and provide LISP functions for manipulating such data objects, providing primitives for rational arithmetic. This would be a strange move at this point, however, as the data structure merely copies what the constraint network represents anyway: a data structure (the multiplier constraint, considered as a division box) with two known values (numerator and denominator). The term "rational arithmetic" is a misnomer, for it is actually a curious combination of arithmetic and algebra—and thus far our language, which can propagate values within the network but cannot augment the network, encompasses only arithmetic.

There is one more way in which local propagation can fail to compute a result. If a constraint network contains cycles, then propagation may not be able to make progress. The difficulty is that such a network expresses a set of simultaneous equations which must be solved by algebra. Consider the network in Figure 2-10. There are three variables `p1`, `p2`, and `p3`, intended to represent the positions of three points along an axis. The network constrains the three points to be equally spaced; that is, `p2` is midway between `p1` and `p3`. The network actually expresses the first description of the last sentence more closely; the two adder constraints determine the spacing between adjacent points, and then the two distances are equated.

$$p2 - p1 = p3 - p2$$

The second description corresponds more to the formula

$$p2 = \frac{p1 + p3}{2}$$

Yet a third formulation is that the spacing between the endpoints is twice the spacing between either set of adjacent points.

$$p3 - p1 = 2 \times (p3 - p2) = 2 \times (p2 - p1)$$

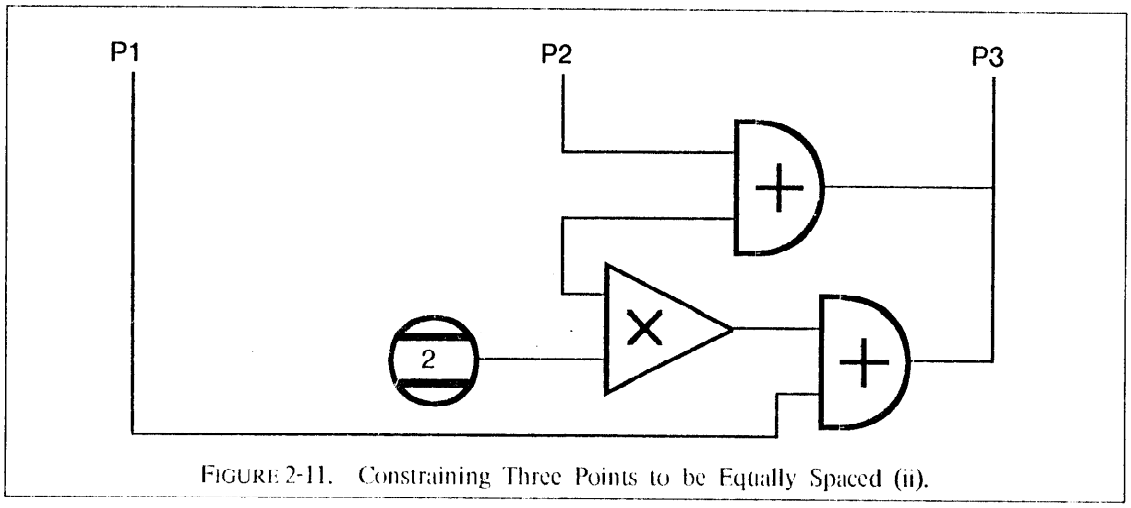


FIGURE 2-11. Constraining Three Points to be Equally Spaced (ii).

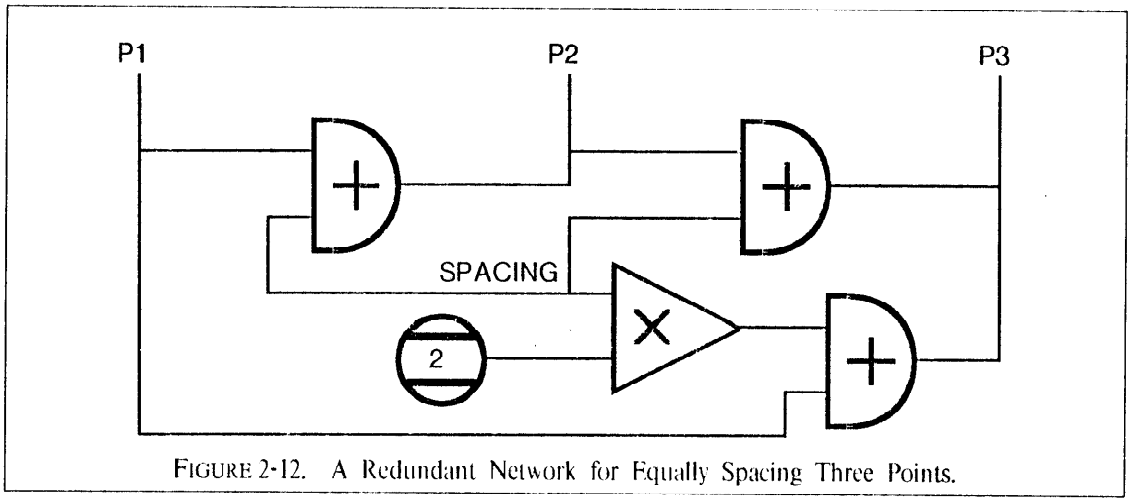


FIGURE 2-12. A Redundant Network for Equally Spacing Three Points.

However, the network expresses only the first formulation. Given either pair of adjacent points, the position of the third is easily computed: one adder calculates the spacing between them, then the other adds or subtracts this spacing from the midpoint to locate the endpoint not given. However, if the two endpoints are given and not the midpoint, neither adder can compute anything.

Similarly, if we were to use just one version of the third formulation (see Figure 2-11), then given `p1` and `p2` it would not be possible to compute `p3` by local propagation.

One way to enable any point to be computed given the other two is to use a redundant network expressing multiple ways of viewing the problem [Sussman 1977] (see Figure 2-12). Another way is simply to use an entirely different network, such as the second formulation above (see Figure 2-13); however, deriving this from the first network requires some non-trivial algebra, and

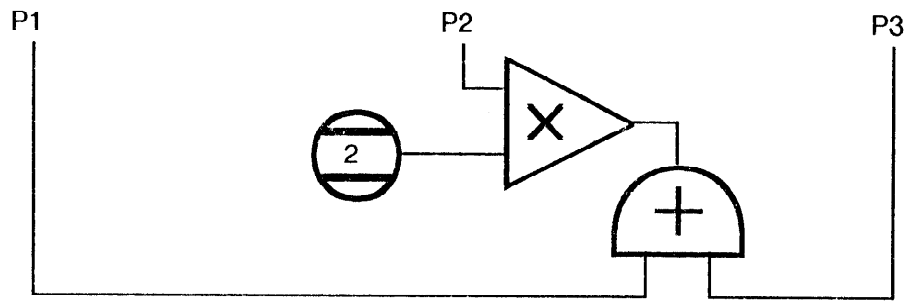


FIGURE 2-13. A Cycle-Free Network for Equally Spacing Three Points.

indeed a new concept: the other networks express direct spacing requirements, whereas Figure 2-13 uses the concept of "averaging the positions of the endpoints".

2.5. Summary of the Trivial Constraint Language

Our little language illustrates the principle of computation by local propagation, with the direction of propagation determined dynamically as needed. It certainly leaves much to be desired:

- The data objects are limited to the integers. We are used to having other kinds of objects in programming languages, including compound objects such as arrays.
- There is no abstraction capability: the language is "flat", which is to say that we can build very large networks but cannot in any way encapsulate portions into modules. We would like to have something analogous to subroutines.
- A given network can be built and then used once, but then must be thrown away. For example, in §2.3, after using a temperature conversion network to convert —40° C to —40° F, we could not then use the same network to convert 37° C—we got a contradiction because the network was already "used up". This means we cannot use a constraint network in a dynamic manner to track a changing input—and this would be one of the most useful attributes of a constraint language, if we could only implement it.
- The mechanism of local propagation can fail to compute a result for any of several reasons. A relationship may be multiple-valued (as $\mathbf{arcmax}_x x$), and there is no good way to choose among the possibilities. A relationship may "have a value", but one which is not really in the domain of the language (for example, rational numbers in a system providing only integers). These are both local properties of a single constraint. It is also possible that the difficulty is global, involving cycles in the network, and cannot be handled by a local technique. This can be handled by algebraic techniques, which involve transformations of the network, which for now is outside the computational scope of our system.

- When contradictions occur (for whatever reason—re-use of a network, a mistake, etc.) there is presently no easy way to determine why the contradiction occurred. We know what the difficulty is locally (for example, division by zero), but we do not know the global causes. A related difficulty is that when something fails to be computed, as in §2.4, we don't know why that happened either.
- The system is computationally inefficient. It often recomputes the same value many times.

We will deal with all these difficulties in one way or another. To deal with all at once would introduce overwhelming complexity of detail; therefore we shall examine each feature separately before combining them.

The best of the worst is full alive.
Tho' wurst is not at first—
No livery may deliver me
An aliver *liverwurst.*
—Walt Kelly (1952)
I Go Pogo

Chapter Three

Dependencies

WHEN A DECISION IS MADE, we very often wish to ask the person who made it not only, "What is the result?", but also, "Why is that so? Why didn't you choose something else? What factors went into your decision?" This is particularly true of design decisions in engineering. There are several reasons for asking such questions. If the result is not obvious, or by itself doesn't carry enough information, then the structure of the process which derived it may shed more light. If something goes wrong later, we need this information to determine how to fix the problem; we need to know what can be changed without affecting the result, or what to change to change the result. More generally, if several decisions have been made, and one must be altered, one can do this with minimum effort if one can determine parameters of the decision which will not affect others. Another possibility is that no decision was reached. In this case one wants to know why, and what additional facts are needed to make a decision.

Now all of this is especially true of computers, which make so many decisions and computations so rapidly that it is very difficult to determine what happened (or didn't happen) after the fact. The entire art of analyzing post-mortem core dumps is devoted to answering the questions outlined above. The notion of an audit trail (computerized or not) is also intended to permit the reconstruction of the computational process. It would be much simpler if programs were to keep track of the reasons for their computations from the start. Such programs could be held accountable for their actions, and required to explain themselves on request.

In this chapter we will alter the system of Chapter Two to record the history of the computation as propagation occurs. Facilities will be developed for extracting this history from the network in a useful form.

3.1. Responsible Programs

It is not at all difficult to augment our trivial constraint system to record reasons for each computational action. We call such reasons *dependencies*, because they indicate for each quantity on what other quantities its derivation depends.

3.1.1. Dependency Information Can Be Used to Explain Computations

Let us suppose that we have been given a temperature conversion network with which we are unfamiliar, but we have been assured that it correctly constrains two variables called `fahrenheit` and `centigrade`. We equate `centigrade` to —40, then ask for the value of `fahrenheit` , and are told that it also is —40.

```
fahrenheit
<CELL-35 (FAHRENHEIT): -40>
```

We don't trust the calculation (perhaps we suspect that there is a "short circuit" between the `fahrenheit` and `centigrade` variables—people sometimes make that sort of mistake!). We ask *why* `fahrenheit` is —40.

```
(why fahrenheit)
;The value -40 is in CELL-35 because that is connected to (THE C ADD),
;   and <ADD:ADDER-23> computed it using rule ADDER-RULE-1
;   from: CELL-24 (A) = -72, CELL-25 (B) = 32.
Q.E.D.
```

So the adder `add` computed —40 from —72 and 32. Let us investigate further.

```
(why (the b add))
;The value 32 is in CELL-25 because that is connected to CELL-37,
;   and that is a constant.
Q.E.D.
(why (the a add))
;The value -72 is in CELL-24 because that is connected to (THE A OTHERMULT),
;   and <OTHERMULT:MULTIPLIER-31> computed it using rule MULTIPLIER-RULE-8
;   from: CELL-33 (B) = 5, CELL-34 (C) = -360.
Q.E.D.
```

One value is a constant, and the other was computed by a multiplier called `othermult`. Let us examine this multiplier.

```
(why (the b othermult))
```

```
;The value 5 is in CELL-33 because that is connected to CELL-38,
;   and that is a constant.
Q.E.D.
(why (the c othermult))
;The value -360 is in CELL-34 because that is connected to (THE C MULT),
;   and <MULT:MULTIPLIER-27> computed it using rule MULTIPLIER-RULE-6
;   from: CELL-28 (A) = 9, CELL-29 (B) = -40.
Q.E.D.
```

Now `othermult` got its `c` value from another multiplier called `mult`. We press on . . .

```
(why (the a mult))
;The value 9 is in CELL-28 because that is connected to CELL-39,
;   and that is a constant.
Q.E.D.
(why (the b mult))
;The value -40 is in CELL-29 because that is connected to CELL-40,
;   and that is a constant.
Q.E.D.
```

We have now traced out the entire computation, and if we reconstruct the flow of information, we can deduce that the structure of the computation can be expressed as the formula

$$\texttt{fahrenheit} = \frac{9 \times -40}{5} + 32$$

which is certainly the correct computation.

We could also inquire as to the status of `centigrade` (for example, we might have forgotten that we set it ourselves[1]).

```
(why centigrade)
;The value -40 is in CELL-36 because that is connected to CELL-40,
;   and that is a constant.
Q.E.D.
(why cell-40)
;The value -40 is in CELL-40 because that is a constant.
Q.E.D.
```

Now perhaps we don't care about the form of the computation, but only wish to know what input parameters were used to compute the result. (This is trivial for our example, but for very complicated networks this may not be at all obvious.)

```
(why-ultimately fahrenheit)
;The value -40 is in CELL-35 because that is connected to (THE C ADD),
```

1. In general, we reserve the right to have a poor memory—the computer, however, is supposed to remember! Poor computer.

```
;   and it was ultimately derived from:
;   <CELL-37 (?): 32>,
;   <CELL-40 (?): -40> = CENTIGRADE,
;   <CELL-39 (?): 9>,
;   <CELL-38 (?): 5>.
Q.E.D.
```

Four values went into the computation, one of which has the name `centigrade`. Indeed, if names were given to the other values, we would like to see them also.

```
(variable linear-offset)
<CELL-41 (LINEAR-OFFSET): no value>
(variable linear-scale-factor-denominator)
<CELL-42 (LINEAR-SCALE-FACTOR-DENOMINATOR): no value>
(variable linear-scale-factor-numerator)
<CELL-43 (LINEAR-SCALE-FACTOR-NUMERATOR): no value>
(variable another-name)
<CELL-44 (ANOTHER-NAME): no value>
(== (the b add) linear-offset)
DONE
(== (the b othermult) linear-scale-factor-denominator)
DONE
(== (the a mult) linear-scale-factor-numerator)
DONE
(== centigrade another-name)
DONE
(why-ultimately fahrenheit)
;The value -40 is in CELL-35 because that is connected to (THE C ADD),
;   and it was ultimately derived from:
;   <CELL-37 (?): 32> == LINEAR-OFFSET,
;   <CELL-40 (?): -40> == CENTIGRADE == ANOTHER-NAME,
;   <CELL-39 (?): 9> == LINEAR-SCALE-FACTOR-NUMERATOR,
;   <CELL-38 (?): 5> == LINEAR-SCALE-FACTOR-DENOMINATOR.
Q.E.D.
```

We can of course use the two query types together. After using `why` to trace down the computation tree a few steps, we can use `why-ultimately` to determine which parameters an intermediate value depends on.

```
(why-ultimately (the c mult))
;The value -360 is in CELL-30 because it was ultimately derived from:
;   <CELL-40 (?): -40> == CENTIGRADE == ANOTHER-NAME,
;   <CELL-39 (?): 9> == LINEAR-SCALE-FACTOR-NUMERATOR.
Q.E.D.
```

3.1.2. Required Parameters Can Be Deduced from the Network Structure

Dependency information indicates how information was propagated within the network; the information exists only after computations have been performed. Because any computation performed by local propagation follows the structure of the network (having choices only in the *direction* of the flow over existing paths), however, we can consider the network to prescribe the set of *potential* dependency relationships. Hence the network structure can be used to explain why a computation did not occur, or to indicate how one could occur which has not yet.

Let us take another temperature conversion network, as at the beginning of §2.4. Before assigning any value to `centigrade`, let us ask "`(why fahrenheit)`", this time meaning "Why does `fahrenheit` *not* have a value?"

```
fahrenheit
<CELL-108 (FAHRENHEIT): no value>
(why fahrenheit)
;CELL-108 has no value.  I could compute it
;   from pins A, B of ADD by rule ADDER-RULE-1.
Q.E.D.
```

This tells us that `fahrenheit` has no value, and suggests a way in which it might be computed.

```
(why centigrade)
;CELL-109 has no value.  I could compute it
;   from pins A, C of MULT by rule MULTIPLIER-RULE-7.
Q.E.D.
```

Of course `centigrade` has no value either. It could be computed if `fahrenheit` were given, for example.

```
(why-ultimately fahrenheit)
;CELL-108 has no value.  Perhaps knowing the value of CENTIGRADE would help.
Q.E.D.
(why-ultimately centigrade)
;CELL-109 has no value.  Perhaps knowing the value of FAHRENHEIT would help.
Q.E.D.
```

Ultimately the computation of either variable depends on the other plus the existing constants (5, 9, and 32) in the network. Only missing parameters are given by `why-ultimately`.

```
(why (the c mult))
;CELL-103 has no value.  I could compute it
;   from pins A, B of OTHERMULT by rule MULTIPLIER-RULE-6; or
;   from pin B of OTHERMULT by rule MULTIPLIER-RULE-5; or
;   from pin A of OTHERMULT by rule MULTIPLIER-RULE-4; or
;   from pins A, B of MULT by rule MULTIPLIER-RULE-6; or
;   from pin B of MULT by rule MULTIPLIER-RULE-5; or
```

```
;   from pin A of MULT by rule MULTIPLIER-RULE-4.
Q.E.D.
```

The intermediate point `(the c mult)` could be computed in any of a number of ways, by either of two constraint devices.

```
(why-ultimately (the c mult))
;CELL-103 has no value.  Perhaps knowing the value of CENTIGRADE or
;   FAHRENHEIT would help.
Q.E.D.
```

Ultimately either of the two variables could be used to compute a value for `(the c mult)`.

Suppose now that we equate `centigrade` to 37 as in §2.4. As before, `othermult` will be unable to compute a value because the division is not exact.

```
(== centigrade (constant 37))
;|Awakening <MULT:MULTIPLIER-100> because its B got the value 37.
;|<MULT:MULTIPLIER-100> computed 333 for its part C from pins A, B.
;|Awakening <OTHERMULT:MULTIPLIER-104> because its C got the value 333.
;|Awakening <MULT:MULTIPLIER-100> because its C got the value 333.
DONE
```

Therefore `fahrenheit` has no value.

```
fahrenheit
<CELL-108 (FAHRENHEIT): no value>
(why fahrenheit)
;CELL-108 has no value.  I could compute it
;   from pins A, B of ADD by rule ADDER-RULE-1.
Q.E.D.
(why-ultimately fahrenheit)
;CELL-108 has no value.
Q.E.D.
```

A sad state of affairs indeed. About all can be said is that the computation has failed. There are no missing parameters—`centigrade` has been supplied. The computation has broken down at `othermult`.

```
(why (the a othermult))
;CELL-105 has no value.  I could compute it
;   from pins B, C of ADD by rule ADDER-RULE-3; or
;   from pins B, C of OTHERMULT by rule MULTIPLIER-RULE-8.
Q.E.D.
(the b othermult)
<CELL-106 (B of OTHERMULT): 5>
(the c othermult)
<CELL-107 (C of OTHERMULT): 333>
```

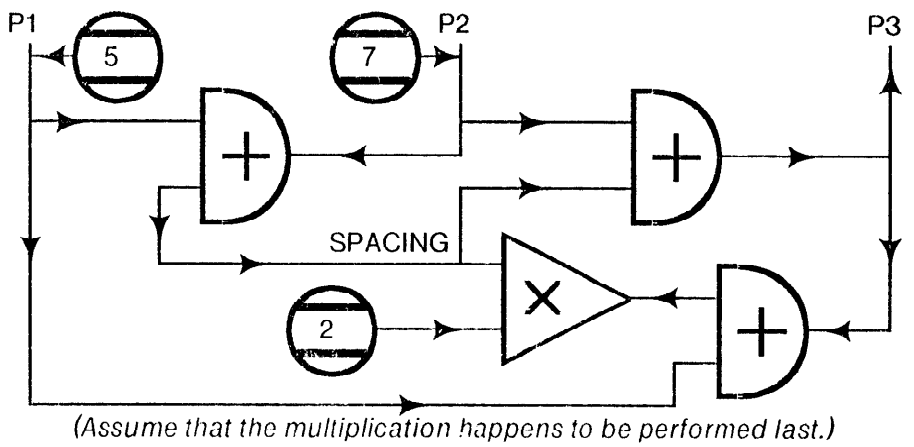


FIGURE 3-1. Multiple Suppliers in the Equal-Spacing Network.

The only rule `othermult` has for computing its `a` is `multiplier-rule-8`, which requires `b` and `c`. However, the `b` and `c` do have values, and nevertheless no value was computed for the `a`. So there is no hope. Also, it would not do for `(why-ultimately fahrenheit)` to say, "Perhaps knowing the value of `(the a othermult)` would help"; there is no integer value that can be given it that is consistent with the other pins of `othermult` already known.

We will return to this problem of "failed computations" in a later section. First let us discuss how to implement the recording of dependencies, and the mechanisms needed for the operation of `why` and `why-ultimately`.

3.2. Recording Dependencies

Recording dependency information simply amounts to remembering the directions of the arrows of Figure 2-3 (page 41), plus which rule was used to compute each outgoing value from a constraint box. Observe that a repository which has a value can have first acquired that value from exactly one of its associated cells (a constant, or a pin of a constraint). We refer to this cell as the *supplier* of the value. Later other cells may also provide values, but such values will merely confirm or contradict the first value.[2]

It is possible to regard other cells which provide values as subsidiary suppliers, and to record them along with the distinguished supplier. There are difficulties with using subsidiary suppliers, however. Recall that in §2.3 the adder `add` computed `b` from `a` and `c`, and so the

2. This argument implicitly assumes a sequential interpreter for the language such as we have presented here. The language certainly admits parallel evaluation, however, in which case computed values may arrive at a repository "simultaneously". In this case we assume that an arbiter chooses one to be first.

```
(deftype repository ((rep-contents ()) (rep-boundp ()) (rep-cells ())
                     (rep-supplier ()) (rep-rule ()) (rep-mark ()))
  (format stream "<Repository~:[~*~;: ~S~]~@[ for ~{~S~^,~}~]>"
          (rep-boundp repository)
          (rep-contents repository)
          (cell-ids repository)))

(defmacro node-contents (cell) `(rep-contents (cell-repository ,cell)))
(defmacro node-boundp (cell) `(rep-boundp (cell-repository ,cell)))
(defmacro node-cells (cell) `(rep-cells (cell-repository ,cell)))
(defmacro node-supplier (cell) `(rep-supplier (cell-repository ,cell)))
(defmacro node-rule (cell) `(rep-rule (cell-repository ,cell)))
(defmacro node-mark (cell) `(rep-mark (cell-repository ,cell)))
```

Compare this with Table 2-1 (page 45).

TABLE 3-1. Extra Repository Fields for Recording Dependencies.

cell `(the b add)` served as supplier for its repository. However, the adder then proceeded to awaken to the fact that its `b` had just received a value, and computed `c` from `a` and `b` (and similarly `a` from `c` and `b`). Thus `(the c add)` became a subsidiary supplier for its repository. To make use of this fact, however, in explaining the computation of `(the b add)` would involve circular reasoning. While we might circumvent this particular problem by avoiding the awakening of the adder when it computed a value for its own pin, the problem would remain in general in networks with large cycles. For example, in the equal-spacing network of Figure 2-12 (page 65), the propagation of values might proceed as in Figure 3-1, and the multiplier would be a subsidiary supplier of the spacing factor. However, we would not want to use this fact to justify the computation of the spacing factor, because then the value of `p3` would appear to to have been computed indirectly from itself.

In order not to produce circular explanations, it is necessary for the dependency structures to be well-founded. We will achieve this be recording only primary suppliers, which guarantees that no cycles will occur. (One can think of information as a fluid spreading from constants throughout the constraint network by propagation, different flows combining within constraint boxes, but stopping short just before meeting in a repository.)

In the implementation we therefore introduce some new components for repositories. (It will not be necessary to change the definitions for cells, constraints, or constraint-types.) These are:

- A *supplier*, which is that cell among the node-cells which first provided the value for the repository. The supplier is null if the repository has no value (boundp is false).
- A *rule*, which is the name of the rule used to compute the value. The rule component is null if the repository has no value, or if the supplier has no owner (i.e., is a constant).
- A *mark*, which is normally null but is available to serve as a mark bit or a counter by various graph-marking algorithms to be intoduced later.

```
(defun constant (value)
  (let ((cell (gen-cell)))
    (setf (node-contents cell) value)
    (setf (node-boundp cell) t)
    (setf (node-supplier cell) cell)
    cell))
```

Compare this with Table 2-2 (page 47).

TABLE 3-2. A Constant Cell Is Its Own Supplier.

The new definition of the `repository` data structure appears in Table 3-1. Vertical lines to the left of the code draw attention to differences from the previous version. As before, extra macros like `node-supplier` are defined to make it easier to refer to components of a node (in the repository) given one of its cells.

New code is needed to maintain the supplier and rule components of repositories. No new code is needed for generating a cell (`gen-cell`); the initial-value mechanism of `deftype` correctly initializes the new components of a repository. When a constant cell is created, however, that cell should be its own supplier (see Table 3-2).

Several changes to the code for `==` appear in Table 3-3. One improvement which is superficially unrelated to maintaining dependencies is that no new repository is created when two cells are equated; instead one repository is re-used. Whichever repository has a value is the one chosen, so it is unnecessary to explicitly update the supplier and rule components. Also, it is only necessary to update (in the `dolist` loop) the `cell-repository` components of cells which belonged to the repository not chosen. The loop for awakening all the owners of a set of cells has been abstracted out as a separate procedure `awaken-all`, which will also be used by `process-setc` later.

A particularly nasty opportunity for implementation bugs arises in the situation where both the cells being equated already have values. As before, `merge-values` will ensure that the two values are compatible. However, it previously did not matter which of two compatible values was used; but now, when values have dependency information attached, it is crucial not to pick the wrong one, lest circularities arise in the dependency structure. Suppose, for example, that a multiplier `m` is created, and its `a` is equated to zero.

```
(create m multiplier)
<M:MULTIPLIER-23>
(== (the a m) (constant 0))
;|Awakening <M:MULTIPLIER-23> because its A got the value 0.
;|<M:MULTIPLIER-23> computed 0 for its part C from pin A.
;|Awakening <M:MULTIPLIER-23> because its C got the value 0.
DONE
```

```
(defun == (cell1 cell2)
  (require-cell cell1)
  (require-cell cell2)
  (or (eq (cell-repository cell1) (cell-repository cell2))
      (let ((r1 (cell-repository cell1))
            (r2 (cell-repository cell2))
            (cb1 (node-boundp cell1))
            (cb2 (node-boundp cell2)))
        (let ((r (if (or (not cb2) (and cb1 (ancestor cell1 cell2))) r1 r2))
              (rcells (append (rep-cells r1) (rep-cells r2))))
          (setf (rep-contents r) (merge-values cell1 cell2))
          (let ((newcomers (if cb1 (if cb2 '() (rep-cells r2))
                               (if cb2 (rep-cells r1) '()))))
            (setf (rep-cells r) rcells)
            (dolist (cell (rep-cells (if (eq r r1) r2 r1)))
              (setf (cell-repository cell) r))
            (awaken-all newcomers)
            'done)))))

(defun awaken-all (cells)
  (dolist (cell cells)
    (require-cell cell)
    (cond ((cell-owner cell)
           (ctrace "Awakening ~S because its ~S got the value ~S."
                   (cell-owner cell)
                   (cell-name cell)
                   (node-contents cell))
           (awaken (cell-owner cell))))))
```

Compare this with Table 2-5 (page 51).

TABLE 3-3. Maintaining Supplier Components When Equating Cells.

Now that the a and c both have the value zero, they are equated.

```
(== (the c m) (the a m))
DONE
(why (the a m))
;The value 0 is in CELL-24 because that is connected to (THE C M),
;   and <M:MULTIPLIER-23> computed it using rule MULTIPLIER-RULE-4
;   from: CELL-24 (A) = 0.
Q.E.D.
```

This is what occurs if the version of == in Table 3-4 is used (which is a version the author used for quite a while before finding the bug while trying to "prove" it correct to himself!). The repository belonging to c is arbitrarily chosen for use by the two cells for a and c, and the result is that c appears to be the primary supplier rather than the constant zero. Now zero is not the only value consistent with the network constructed (a and c could be any value if b were 1), and so it is quite improper for the value zero in (the a m) to claim to support itself.

That this dependency cycle arises in this example is of course accidental. Had the equating of a and c been written as

```
(defun == (cell1 cell2)
  (require-cell cell1)
  (require-cell cell2)
  (or (eq (cell-repository cell1) (cell-repository cell2))
      (let ((r1 (cell-repository cell1))
            (r2 (cell-repository cell2))
            (cb1 (node-boundp cell1))
            (cb2 (node-boundp cell2)))
        (let ((r (if cb1 r1 r2))                          ;There is a bug here!
              (rcells (append (rep-cells r1) (rep-cells r2))))
          (setf (rep-contents r) (merge-values cell1 cell2))
          (let ((newcomers (if cb1 (if cb2 '() (rep-cells r2))
                               (if cb2 (rep-cells r1) '()))))
            (setf (rep-cells r) rcells)
            (dolist (cell (rep-cells (if (eq r r1) r2 r1)))
              (setf (cell-repository cell) r))
            (awaken-all newcomers)
            'done)))))
```

Compare this with Table 3-3.

TABLE 3-4. An Incorrect Implementation of Equating.

```
(== (the a m) (the c m))
```

then the *correct* repository would have been accidentally chosen, and all would be well. Again, if the connection between `a` and `c` been made before the connection to the constant zero, then all would have been well. However, we would like a constraint language to be as free as possible of such accidental ordering problems. The system must always do things in a consistent and correct manner. This is the reason for the use of the `ancestor` predicate in the (correct) definition of `==` in Table 3-3. Given two cells which have values, `ancestor` returns "true" if and only if the value in the second cell was supplied by a computation depending in part on the first cell; in this case the first cell is said to be an ancestor of the second. This predicate defines a partial order on cells with values *if* the dependencies are kept consistent and cycle-free; indeed, the predicate is precisely that partial order defined by the transitive closure of the "primary supplier" relation plus the "triggers-for" relation which indicates what values were used by a rule to compute a new value. The definition of `ancestor` will appear a little later when details of the new representation of rules have been elaborated upon.

In the last chapter rules were simply LISP functions which could be run whenever a cell got a value. This will still be true, but for explanation purposes it will be useful to associate other information with rules. As a matter of implementation convenience[3] the property list of the symbol naming the rule is used to store this extra information. It would be perfectly reasonable to define

3. or laziness—but this illustrates a common technique of LISP programming: the use of the property list. It also illustrates a general technique of interactive programming: do as little work as you can while trying out an idea—the time to polish the code is after the idea is known to work. Put another way, it's not worth investing a lot of effort for the sake of elegance or speed in an idea that may not work anyway.

```
(defprim adder (a b c)
  (c (a b) (setc c (+ a b)))
  (b (a c) (setc b (- c a)))
  (a (b c) (setc a (- c b))))

(defprim multiplier (a b c)
  (c (a) (and (zerop a) (setc c 0)))
  (c (b) (and (zerop b) (setc c 0)))
  (c (a b) (setc c (* a b)))
  (b (a c) (and (not (zerop a)) (zerop (\ c a)) (setc b (// c a))))
  (a (b c) (and (not (zerop b)) (zerop (\ c b)) (setc a (// c b)))))

(defprim maxer (a b c)
  (c (a b) (setc c (max a b)))
  (b (a c) (cond ((< a c) (setc b c))
                 ((> a c) (contradiction a c))))
  (a (b c) (cond ((< b c) (setc a c))
                 ((> b c) (contradiction b c)))))

(defprim minner (a b c)
  (c (a b) (setc c (min a b)))
  (b (a c) (cond ((> a c) (setc b c))
                 ((< a c) (contradiction a c))))
  (a (b c) (cond ((> b c) (setc a c))
                 ((< b c) (contradiction b c)))))

(defprim equality (p a b)
  ((p) (or (= p 0) (= p 1) (contradiction p)))
  (p (a b) (setc p (if (= a b) 1 0)))
  (b (p a) (and (= p 1) (setc b a)))
  (a (p b) (and (= p 1) (setc a b))))

(defprim gate (p a b)
  ((p) (or (= p 0) (= p 1) (contradiction p)))
  (p (a b) (or (= a b) (setc p 0)))
  (b (p a) (and (= p 1) (setc b a)))
  (a (p b) (and (= p 1) (setc a b))))
```

Compare this with Table 2-7 (page 53).

TABLE 3-5. Implementation of Primitive Constraints with Dependency Information.

a new data type called `rule` with several components (one of them being the function itself), but this technique lessens the distance between the old and new code; for example, the code for `awaken` need not be altered.

With each rule is associated two lists of names of pins. The list *trigger-names* contains the names of pins which must have values in order to run the body of the rule. This is exactly the set of pins whose boundp components are checked by the preamble in each rule defined by `defprim`. The list *output-names* contains the names of pins which might (or might not) receive values when the rule is run. Thus these are the pins which are "inputs" or "outputs" for that rule. Any given invocation of the rule might not use all the inputs and might not give values to all the outputs, however, depending on the values of inputs examined.

```
(defmacro defrule (typename output-names trigger-names . body)
  (let ((rulename (gen-name typename 'rule)))
    `(progn 'compile
            (push ',rulename (ctype-rules ,typename))
            (defun ,rulename (*me*) (let ((*rule* ',rulename)) ,@body))
            (defprop ,rulename ,trigger-names trigger-names)
            (defprop ,rulename ,output-names output-names)
            '(,typename rule))))

(defmacro defprim (name vars . rules)
  `(progn 'compile
          (declare (special ,name))
          (setq ,name (make-constraint-type))
          (setf (ctype-name ,name) ',name)
          (setf (ctype-vars ,name) ',vars)
          ,@(forlist (rule rules)
              (do ((r rule (cdr r))
                   (output-names '() (cons (car r) output-names)))
                  ((or (null (car r)) (not (atom (car r))))
                   (let ((trigger-names (car r))
                         (body (cdr r)))
                     `(defrule ,name ,output-names ,trigger-names
                        (let ,(forlist (var vars)
                                `(,(symbolconc var "-CELL") (the ,var *me*)))
                          (and ,@(forlist (var trigger-names)
                                   `(node-boundp ,(symbolconc var "-CELL")))
                               (let ,(forlist (var trigger-names)
                                       `(,var (node-contents
                                                ,(symbolconc var "-CELL"))))
                                 ,@body)))))))))
          '(,name primitive)))
```

Compare this with Table 2-8 (page 55).

TABLE 3-6. Definition of `defprim` Which Saves Rule Information.

These lists could be computed automatically by analyzing the code of the rule-body, and a "real" constraint language system ought to do this. To save work here, however, that information will be represented redundantly (just as in the last chapter the set of trigger names was written redunadantly). The format of `defprim` is redefined such that the output names are written before the list of input names in each rule clause. New definitions of the primitive constraint boxes are in Table 3-5; new definitions of `defprim` and `defrule` appear in Table 3-6. (Only the most important changes in the code are indicated by vertical lines to the left—for example, the substitutions of "`trigger-names`" for "`(car rule)`" in several places in `defprim` are not marked.) One change to `defrule` which is used by `process-setc` is that the variable `*rule*` is bound to the name of the rule when the rule-body is executed.

If the primary supplier for a value is a pin of a constraint, then the repository for that value also contains the name of the rule which derived that value. Given that, the names of the triggers

```
(defun ancestor (cell1 cell2)
  (let ((r1 (cell-repository cell1))
        (r2 (cell-repository cell2)))
    (or (eq r1 r2)
        (and (rep-boundp r2)
             (cell-owner (rep-supplier r2))
             (do ((tns (get (node-rule (rep-supplier r2)) 'trigger-names)
                       (cdr tns)))
                 ((null tns) ())
               (and (ancestor cell1 (*the (car tns) (cell-owner (rep-supplier r2))))
                    (return t)))))))
```

TABLE 3-7. Definition of the Ancestor Relationship between Cells with Values.

for that rule can be obtained, and from that and the owner of the pin the trigger cells themselves and their values can be located. This is all that is needed to define the `ancestor` predicate (see Table 3-7). One cell is an ancestor of another if they have the same repository, or if the second is bound and the first is an ancestor of one of the triggers for the rule used to compute the second. (Another way to compute this would be: the first is an ancestor of the second if they have the same repository, or if any pin for which the first had been a trigger is an ancestor of the second. This searches from the top down rather than the bottom up. However, a value may be a trigger for arbitrarily many other values, but any given value is computed from only as many trigger values as are required by the rule needing the greatest number of triggers (among all rules in the system). Intuitively, then, the fanout of the search procedure in Table 3-7 is guaranteed to be bounded, wile that of the other is not. On the other hand, perhaps in typical use the typical value is a trigger for only one or two other values. I have not yet made measurements to determine which procedure is better in practice.)

No change is necessary for handling the `contradiction` construct. On the other hand, `setc` and `process-setc` must be changed to install supplier and rule information (see Table 3-8). With this requirement, it is just as easy *not* to make up a fresh cell and equate it to the pin; instead, one might as well just do the relevant tests and install the new value (if indeed it is new) in the existing repository along with the supplier and rule information. If the pin to be set does not have a value, then the value and dependency information is installed and all interested parties awakened. If it does have a value, then it had better be the same as the one we wish to install. Technically `merge-values` should be used here, but for now we omit this for the sake of giving a more precise error message. Similarly, a side benefit of having `setc` do some case analysis is that less `ctrace` output is generated; this version of `setc` only calls `ctrace` when a new value has been computed.

```
(defmacro setc (cellname value)
  `(process-setc *me* ',cellname ,(symbolconc cellname "-CELL") ,value *rule*))

(defun process-setc (*me* name cell value rule)
  (require-constraint *me*)
  (require-cell cell)
  (let ((sources (get rule 'trigger-names)))
    (cond ((not (node-boundp cell))
           (ctrace "~S computed ~S for its part ~S~:[~2*~; from pin~P ~{~S~^, ~}~]."
                   *me* value name sources (length sources) sources)
           (setf (node-contents cell) value)
           (setf (node-boundp cell) t)
           (setf (node-supplier cell) cell)
           (setf (node-rule cell) rule)
           (awaken-all (node-cells cell)))
          ((not (equal (node-contents cell) value))
           (lose "Contradictory values at ~S: ~S says ~S, but ~S says ~S."
                 (cell-id cell)
                 (node-supplier cell)
                 (node-contents cell)
                 *me*
                 value)))))
```

Compare this with Table 2-11 (page 57).

TABLE 3-8. Definition of `setc` for Handling Dependencies.

3.3. Producing Explanations

All the machinery for maintaining dependency information in the constraint network is now in place. The remaining new code uses this information to generate explanations.

The code for `why` appears in Table 3-9. It is complicated only because there are several cases, and because each case tries to format the output neatly. If the cell has no value, then this fact is stated; then all the possible ways of computing it in one step are found and printed, or if none are found this is stated. If the cell has a value, then the value is printed, and if the given cell is not the supplier the connection to the supplier is mentioned; then the supplier may be a constant or may be a pin of a constraint, and in the latter case the relevant rule and its triggers are printed. (The function `cell-goodname` constructs a "good" name for the cell, one the user is most likely to find useful.)

The information printed by `why-ultimately` includes the *premises* of the value asked about. These are all the values used to compute the given value which do not have any ancestors; that is, the premises are the ultimate ancestor values of the given value.

Table 3-10 gives a straightforward but potentially inefficient algorithm for computing the set of premises, given a cell. If the cell has no value, it has no premises. If it is a constant, then it is its own premise. Otherwise the set of premises is the union of the sets of premises for the triggers of the rule used to compute the value in the cell. This algorithm is recursive, and performs a tree walk

```
(defun why (cell)
  (require-cell cell)
  (cond ((not (node-boundp cell))
         (format t "~%;~S has no value." (cell-id cell))
         (let ((flag ()))
           (dolist (c (node-cells cell))
             (and (cell-owner c)
                  (dolist (rule (ctype-rules (con-ctype (cell-owner c))))
                    (let ((trigger-names (get rule 'trigger-names))
                          (output-names (get rule 'output-names)))
                      (cond ((memq (cell-name c) output-names)
                             (format t "~:[  I could compute it~;~
                                        ; or~]"
                                     flag)
                             (setq flag t)
                             (format t "~%;   from ~:[~2*~;pin~P ~{~S~^, ~} of ~]~
                                          ~S by rule ~S"
                                     trigger-names
                                     (length trigger-names)
                                     trigger-names
                                     (con-name (cell-owner c))
                                     rule)))))))
           (format t "~:[  I don't have any way to compute it.~;.~]" flag)))
        (t (format t "~%;The value ~S is in ~S because "
                   (node-contents cell) (cell-id cell))
           (let ((s (node-supplier cell)))
             (or (eq s cell)
                 (format t "that is connected to ~S,~%;   and " (cell-goodname s)))
             (if (null (cell-owner s))
                 (format t "that is a constant.")
                 (format t "~:[~;~%;   ~]~S computed it using rule ~S~
                            ~@[~%;   from: ~:{~S (~S)~:[~*~; = ~S~]~:^, ~}~]."
                         (eq s cell)
                         (cell-owner s)
                         (node-rule s)
                         (forlist (trigger-name (get (node-rule s) 'trigger-names))
                           (let ((cell (*the trigger-name (cell-owner s))))
                             (list (cell-id cell)
                                   trigger-name
                                   (node-boundp cell)
                                   (node-contents cell)))))))))
  'q.e.d.)

(defun cell-goodname (cell)
  (require-cell cell)
  (cond ((globalp cell) (cell-name cell))
        ((constantp cell) (cell-id cell))
        (t (list 'the (cell-name cell) (con-name (cell-owner cell))))))
```

TABLE 3-9. Code for **why**: Generating a One-Step Explanation.

on the dependency structure. Inefficiency arises when the dependency graph is not strictly a tree, but contains many shared subtrees; in the worst case it may take exponential time in the size of the network to compute that set, for example on the network of Figure 3-2.

```
(defun premises (cell)
  (require-cell cell)
  (cond ((not (node-boundp cell)) '())
        (t (let ((s (node-supplier cell)))
             (if (null (cell-owner s))
                 (list s)
                 (do ((tns (get (node-rule s) 'trigger-names) (cdr tns))
                      (p '() (unionq (premises (*the (car tns) (cell-owner s))) p)))
                     ((null tns) p)))))))
```

TABLE 3-10. Calculation of the Premises Supporting a Value.

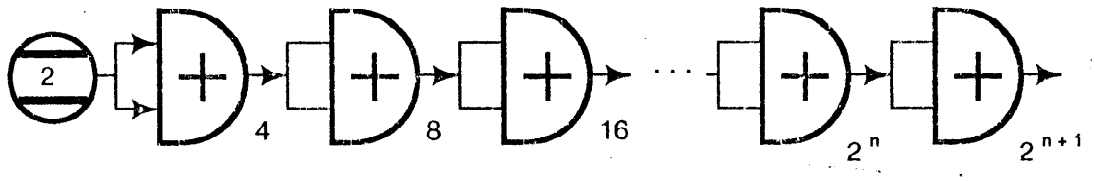


FIGURE 3-2. A Dependency Structure for Which `premises` Takes Exponential Time.

Table 3-11 gives an algorithm that avoids such exponential behavior by marking nodes as they are visited. The macros `mark-node`, `unmark-node`, and `markp` are used to set, clear, and test a (normally clear) mark bit associated with each node. The structure of the algorithm is much the same, except that every node visited is marked, and when a marked node is encountered that branch of the search is cut off. Moreover, since every premise will be counted exactly once, the set-union operation `unionq` (which eliminates duplicate entries) can be replaced by the faster list-concatenation operation `nconc`. After the set of premises has been collected, however, another pass is needed to clear all the marks again (because it is assumed that all marks are initially clear). Thus for dependency structures with no sharing of sub-trees this algorithm may be slower by a constant factor due to overhead (but remember that the first algorithm may be exponentially slower than the fast one in other cases!).

Digression. Another possible approach depends heavily on an underlying garbage collector (which in fact exists in this LISP-based implementation). Rather than using a mark bit, a mark object is used which is uniquely generated for each application of `fast-premises`. When `fast-premises` is called, it generates a new storage object, stores this object in the mark component of every visited node. Thus `markp` merely tests whether the mark component is this object. The generated object cannot be confused with one generated for either an earlier or a later call to `fast-premises`. Confusion could only arise if such an object were re-used while a pointer to it resided in some node; but the garbage collector guarantees that this cannot occur. (This idea is due to Gerald Jay Sussman.)

One can say that a global process is needed so as not to confuse one marking with another. The function `fast-premises-unmark` constitutes one such process. The technique discussed here pushed that work onto the garbage collector, an already existing global process.

```
(defmacro mark-node (cell) `(setf (node-mark ,cell) t))
(defmacro unmark-node (cell) `(setf (node-mark ,cell) ()))
(defmacro markp (cell) `(node-mark ,cell))

(defun fast-premises (cell)
  (require-cell cell)
  (prog1 (fast-premises-mark cell) (fast-premises-unmark cell)))

(defun fast-premises-mark (cell)
  (require-cell cell)
  (and (node-boundp cell)
       (let ((s (node-supplier cell)))
         (cond ((markp s) '())
               (t (mark-node s)
                  (if (null (cell-owner s))
                      (list s)
                      (do ((tns (get (node-rule s) 'trigger-names) (cdr tns))
                           (p '() (nconc (fast-premises-mark
                                          (*the (car tns) (cell-owner s)))
                                         p)))
                          ((null tns) p))))))))

(defun fast-premises-unmark (cell)
  (require-cell cell)
  (let ((s (node-supplier cell)))
    (cond ((markp s)
           (unmark-node s)
           (or (null (cell-owner s))
               (dolist (trigger-name (get (node-rule s) 'trigger-names))
                 (fast-premises-unmark (*the trigger-name (cell-owner s)))))))))
```

TABLE 3-11. Fast Calculation of Premises.

Note that simply generating a number (the "bakery ticket" method) to use for a mark object doesn't quite work—if the size of a number is finite eventually some will be re-used. Only a global process keeping track of which numbers still reside in nodes can avoid confusion.

The number of distinct mark objects simultaneously in existence need not exceed the number of nodes, plus one.

(*End of digression.*)

If a cell has no value, it is still possible to determine the set of *potential* premises of the cell: cells which, if they only had values, might eventually become premises because their values might contribute to a value for the cell of interest. The function `desired-premises` in Table 3-12 computes this set. It uses a graph-marking technique in the same manner as `fast-premises`. In this case cells which have *no* value are of interest. The search is more complicated because at each step, if there are several constraints attached to a node, no one of them is distinguished as the supplier, and no one rule distinguished as the generating rule; instead, all rules of all attached constraints which might possibly compute a value for that node must be considered and recursively searched.

```
(defun desired-premises (cell)
  (require-cell cell)
  (prog1 (desired-premises-mark cell) (desired-premises-unmark cell)))

(defun desired-premises-mark (cell)
  (require-cell cell)
  (cond ((and (not (node-boundp cell))
              (not (markp cell)))
         (mark-node cell)
         (do ((c (node-cells cell) (cdr c))
              (p '() (nconc (if (null (cell-owner (car c)))
                                (and (globalp (car c)) (list (car c)))
                                (desired-premises-constraint (car c)))
                            p)))
             ((null c) p)))))

(defun desired-premises-constraint (cell)
  (require-cell cell)
  (let ((p '()))
    (dolist (rule (ctype-rules (con-ctype (cell-owner cell))))
      (and (memq (cell-name cell) (get rule 'output-names))
           (dolist (trigger (get rule 'trigger-names))
             (setq p (nconc (desired-premises-mark
                             (*the trigger (cell-owner cell)))
                            p)))))
    p))

(defun desired-premises-unmark (cell)
  (require-cell cell)
  (cond ((and (not (node-boundp cell))
              (markp cell))
         (unmark-node cell)
         (dolist (c (node-cells cell))
           (and (cell-owner c)
                (dolist (pin (con-values (cell-owner c)))
                  (desired-premises-unmark pin)))))))

(defun globalp (cell)
  (require-cell cell)
  (and (null (cell-owner cell)) (not (eq (cell-name cell) '?))))
```

TABLE 3-12. Determining Potential Premises for a Cell with No Value.

(The function `globalp` is a predicate which is true of cells which are neither pins nor constants.)

The code for `why-ultimately` (Table 3-13), like that for `why`, divides into two cases. If the given cell is has no value, then that fact is stated, and if the set of desired premises is not empty its elements are listed. If the cell has a value, then the possibilities that it is not the supplier and that the supplier is a constant are considered, exactly as they are for `why`. Then the set of premises is printed; all premises are constants, of course, and so to help distinguish them any global names

```
(defun why-ultimately (cell)
  (require-cell cell)
  (cond ((not (node-boundp cell))
         (format t "~%;~S has no value." (cell-id cell))
         (format t "~@[  Perhaps knowing the value of ~
                    ~{~<~%;  ~:15;~S ~>~^or ~}would help.~]"
                 (forlist (c (delq cell (desired-premises cell))) (cell-name c))))
        (t (format t "~%;The value ~S is in ~S because "
                   (node-contents cell) (cell-id cell))
           (let ((s (node-supplier cell)))
             (or (eq s cell)
                 (format t "that is connected to ~S,~%;   and " (cell-goodname s)))
             (if (null (cell-owner s))
                 (format t "that is a constant.")
                 (format t "it was ultimately derived~
                            ~@[ from:~:{~%;   ~S~@{ == ~S~}~:^,~}~]."
                         (forlist (p (premises s))
                           (cons p (mapcan #'(lambda (c)
                                               (and (globalp c)
                                                    (list (cell-name c))))
                                           (node-cells p)))))))))
  'q.e.d.)
```

TABLE 3-13. Implementation of `why-ultimately`.

attached to a premise are also printed, preceded by `==`.[4]

That's all there is to `why` and `why-ultimately`. Simple, is it not?

The only difficulty with these explanation mechanisms is that one provides very local information, and the other the most global possible information; neither provides any sense of how the local situation is related to its surroundings. Of course, the user can use `why` to chase down the computation step-by-step, but that can produce a *very* long explanation full of trivial details. A long linear explanation at the lowest level is much less useful than a short one mentioning high-level goals.

4. If the set of premises is empty, then the output will say "The value 43 is in CELL-27 because it was ultimately derived." which is admittedly cryptic. This shouldn't happen with the particular primitive constraints shown so far, but could with more bizarre constraint-types. Part of the art of designing `format` messages is arranging for the boundary cases and conditional cases always to be grammatical!

Consider the following subtraction problem ...

342 — 173

Now, remember how we used to do that: three from two is nine, carry the one; and if you're under 35 or went to a private school you say seven from three is six, but if you're over 35 and went to a public school you say eight from four is six; and carry the one, so you have **169.**

But in the new approach, as you know, the important thing is to understand what you're doing rather than to get the right answer. Here's how they do it now:

You can't take three from two,
Two is less than three.
So you look at the four
In the tens place.
Now, that's really four tens.
So you make it three tens.
Regroup.
And you change a ten to ten ones.
Then you add to the two and get twelve
And you take away three—that's nine.

(Is that clear?)

Now instead of four in the tens place.
You've got three.
'Cause you added one—
That is to say, ten—
To the two, but you can't
Take seven from three
So you look in the hundreds place.
From the three
You then use one
To make ten ones
And you know why four
Plus minus-one plus ten
Is fourteen minus one:
'Cause addition is commutative! Right!
And so you have thirteen tens
And you take away seven
and that leaves five!

(Well ... six, actually, ...
but the idea is the important thing.)

Now you go back
To the hundreds place,
You're left with two
And you take away one
From two, and that leaves ...

(Everybody get one? Not bad for the first day.)

Hooray for New Math,
New-ew-ew Math.
It won't do you a bit of good to review math:
It's so simple,
So very simple,
That only a child can do it!

—Tom Lehrer (1965)
"New Math"
That Was The Year That Was

3.4. Representing Symbolic Results in the Network

We return now to the problem encountered at the end of §3.1: what explanation can be given when the computation fails to make progress? More generally, what explanation can be given that is less local than the one-step explanation of `why` but less distant than the leaves of the tree searched by `why-ultimately`? The ultimate explanation of whatever computation did or did not occur certainly lies in the network itself, but it is not necessarily helpful simply to print the entire network, partly because the network may be huge, and partly because it may be that most of the network is irrelevant to the needed explanation. (By way of comparison, it is not of direct help to answer a question like, "What are the colors of the rainbow?" by handing the inquirer an encyclopedia—particularly if the encyclopedia is not alphabetized!—expecting him to read it all to get an answer to his question.)

3.4.1. Subgraphs of the Network May Be Printed as Algebraic Expressions

Here we present a new function `what` which produces an explanation for a cell by copying a carefully chosen part of the network structure, with directions assigned to edges such that the chosen part is an acyclic directed graph and the cell of interest is at the root, and printing that part as nested algebraic expressions. Suppose once again that we have created a fresh temperature conversion network.

```
(what fahrenheit)
;CELL-99 has no value.  I can express it in this way:
;   FAHRENHEIT = (+ (// (* 9 CENTIGRADE) 5) 32)
OKAY?
(what centigrade)
;CELL-100 has no value.  I can express it in this way:
;   CENTIGRADE = (// (* (- FAHRENHEIT 32) 5) 9)
OKAY?
```

The function `what` is not performing transformations on algebraic expressions in the usual general sense. Each the algebraic expressions above is merely a way of printing (a part of—in this case the whole of) the network.

```
(what (the c mult))
;CELL-94 has no value.  I can express it in this way:
;   (THE C MULT) = (* (- FAHRENHEIT 32) 5)
OKAY?
(what (the c othermult))
```

```
;CELL-98 has no value.  I can express it in this way:
;   (THE C OTHERMULT) = (* 9 CENTIGRADE)
OKAY?
```

The c pins of the two multipliers are connected together. However, what prints two different expressions for them because it avoids (as a heuristic) using a constraint as part of the explanation for a valueless pin of that constraint.

```
(variable foo)
<CELL-104 (FOO): no value>
(== (the c mult) foo)
DONE
(what fahrenheit)
;CELL-99 has no value.  I can express it in this way:
;   FAHRENHEIT = (+ (// FOO 5) 32)
OKAY?
(what centigrade)
;CELL-100 has no value.  I can express it in this way:
;   CENTIGRADE = (// FOO 9)
OKAY?
```

Another heuristic is that nodes with explicit global names are a good stopping place. It is not necessary to print the entire network—just to indicate relationships to the "nearest neighbors" which have "meaning" to the user. If the meaning of foo is not clear, one can ask what that is.

```
(what foo)
;CELL-104 has no value.  I can express it in this way:
;   FOO = (* (- FAHRENHEIT 32) 5)
OKAY?
```

Now foo could be expressed in terms of either fahrenheit or centigrade; what happened to choose the former. There may be many equivalent expressions to use; it is not desirable to print them all, and not easy to choose the best. [McAllester 1980] The version of what presented here has only a few heuristics, and to simplify the presentation, no way has been provided to change them. It will serve as a modest example of what can be done, however.

Let centigrade be given the value —40.

```
(== centigrade (constant -40))
;|Awakening <MULT:MULTIPLIER-91> because its B got the value -40.
;|<MULT:MULTIPLIER-91> computed -360 for its part C from pins A, B.
;|Awakening <OTHERMULT:MULTIPLIER-95> because its C got the value -360.
;|<OTHERMULT:MULTIPLIER-95> computed -72 for its part A from pins B, C.
;|Awakening <ADD:ADDER-87> because its A got the value -72.
;|<ADD:ADDER-87> computed -40 for its part C from pins A, B.
;|Awakening <ADD:ADDER-87> because its C got the value -40.
;|Awakening <OTHERMULT:MULTIPLIER-95> because its A got the value -72.
```

```
;|Awakening <MULT:MULTIPLIER-91> because its C got the value -360.
DONE
```

Note that much less `ctrace` output is generated this time (thanks to the changes to `process-setc` in Table 3-8).

```
(what fahrenheit)
;The value -40 in CELL-99 was computed in this way:
;   FAHRENHEIT ← (+ (// (* 9 CENTIGRADE) 5) 32)
;   CENTIGRADE ← -40
OKAY?
```

In this case the heuristic is to explain the value completely. The entire computation is printed. In order to take advantage of nested expression notation, `what` avoids using intermediate names like `foo`. However, it does use the name `centigrade` to identify the constant —40 to distinguish it from the other constants. The variable name `foo` still exists, of course; `what` has merely chosen not to use it.

```
(what foo)
;The value -360 in CELL-104 was computed in this way:
;   FOO ← (* 9 CENTIGRADE)
;   CENTIGRADE ← -40
OKAY?
(what centigrade)
;The value -40 in CELL-100 was computed in this way:
;   CENTIGRADE ← -40
OKAY?
```

Explanations of intermediate stages are also easily generated. This time `foo` is explained in terms of `centigrade` rather than `fahrenheit`, since the value was derived from `centigrade`.

Now suppose that instead of —40, the value 37 is given to `centigrade` (in a fresh temperature conversion network). Recall that this computation will "fail" because of inexact division.

```
(== centigrade (constant 37))
;|Awakening <MULT:MULTIPLIER-110> because its B got the value 37.
;|<MULT:MULTIPLIER-110> computed 333 for its part C from pins A, B.
;|Awakening <OTHERMULT:MULTIPLIER-114> because its C got the value 333.
;|Awakening <MULT:MULTIPLIER-110> because its C got the value 333.
DONE
(what fahrenheit)
;CELL-118 has no value.  I can express it in this way:
;   FAHRENHEIT = (+ (// 333 5) 32)
OKAY?
```

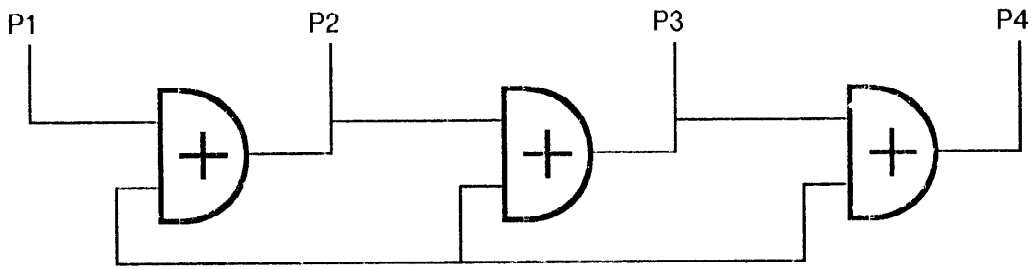


FIGURE 3-3. Constraining Four Points to be Equally Spaced.

Here `what` claims that `fahrenheit` "has no value". However, it is able to print out an expression for it entirely in terms of constants. Except that it is not in reduced form, this is the form of answer we might expect anyway: a mixed number.[5]

The ability of `what` to deal reasonably with networks containing cycles has not been demonstrated, as the temperature conversion network has no cycles. For another example let us use an extension of the network of Figure 2-10 (page 64) for spacing *four* points equally (see Figure 3-3).

```
(defun test ()
  (variable p1)
  (variable p2)
  (variable p3)
  (variable p4)
  (create a12 adder)
  (create a23 adder)
  (create a34 adder)
  (== (the a a12) p1)
  (== (the c a12) p2)
  (== (the b a12) (the b a23))
  (== (the a a23) p2)
  (== (the c a23) p3)
  (== (the b a23) (the b a34))
  (== (the a a34) p3)
  (== (the c a34) p4))
TEST
```

5. Production of a reduced form cannot be done purely by local propagation anyway: it requires algebra. The steps are: $32+333/5 \rightarrow 32+(330+3)/5 \rightarrow 32+(330/5+3/5) \rightarrow 32+(66+3/5) \rightarrow (32+66)+3/5 \rightarrow 98+3/5$, which requires (among other things) distribution of division over addition and associativity of addition, as well as two simple local arithmetic operations and one "non-deterministic" (actually guided by the requirements of "reduced form") reverse-addition splitting of 333 into $330 + 3$.

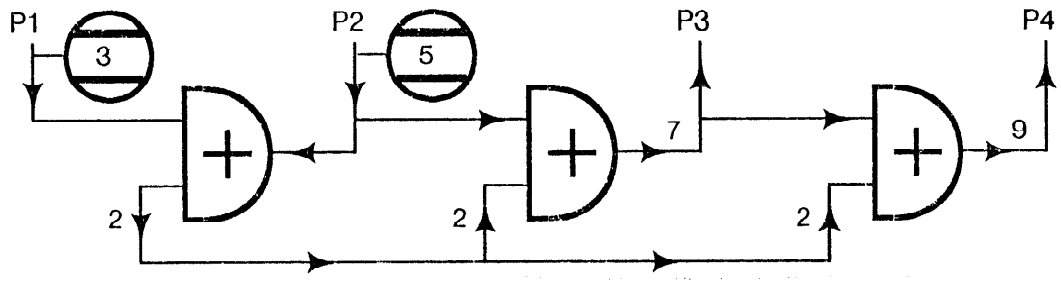


FIGURE 3-4. Computing Equal Spacing for Four Points.

These are all the statements for constructing the four-point equal-spacing network, packaged up as a LISP function called `test`. Executing this function will thus construct an instance of the equal-spacing network.[6]

```
(TEST)
DONE
(what p1)
;CELL-151 has no value.  I can express it in this way:
;   P1 = (- P2 (- P3 P2))
OKAY?
(what p2)
;CELL-152 has no value.  I can express it in this way:
;   P2 = (- P3 (- P2 P1))
OKAY?
(what p3)
;CELL-153 has no value.  I can express it in this way:
;   P3 = (- P4 (- P2 P1))
OKAY?
(what p4)
;CELL-154 has no value.  I can express it in this way:
;   P4 = (+ P3 (- P2 P1))
OKAY?
```

In each case, `what` doesn't run off and print the entire network, but just enough to give a feel for the local connections. Note that three of the equations are not circular; this is somewhat accidental. However, in describing `p1` the expression `(- p3 p2)` was used rather than `(- p2 p1)` because the adder needed for the latter had already been used for the outer subtraction. In this limited (and locally defined!) sense, circularity is avoided in the explanations.

6. Use of a LISP function definition in this way is of course also outside the defined constraint language. This is yet another example of how the facilities of the meta-language can be used to augment the usability of a toy language until the latter grows to the point of providing such facilties itself. Defining and using a function like `test` is much easier than typing a dozen or two statements each time the network is needed. Eventually an equivalent facility will be provided in the constraint language itself.

```
(== p1 (constant 3))
;|Awakening <A12:ADDER-155> because its A got the value 3.
DONE
(== p2 (constant 5))
;|Awakening <A23:ADDER-159> because its A got the value 5.
;|Awakening <A12:ADDER-155> because its C got the value 5.
;|<A12:ADDER-155> computed 2 for its part B from pins A, C.
;|Awakening <A12:ADDER-155> because its B got the value 2.
;|Awakening <A23:ADDER-159> because its B got the value 2.
;|<A23:ADDER-159> computed 7 for its part C from pins A, B.
;|Awakening <A34:ADDER-163> because its A got the value 7.
;|<A34:ADDER-163> computed 9 for its part C from pins A, B.
;|Awakening <A34:ADDER-163> because its C got the value 9.
;|Awakening <A23:ADDER-159> because its C got the value 7.
;|Awakening <A34:ADDER-163> because its B got the value 2.
DONE
```

Presumably now p3 = 7 and p4 = 9. (See Figure 3-4.)

```
(what p3)
;The value 7 in CELL-153 was computed in this way:
;   P3 ← (+ P2 (- P2 P1))
;   P2 ← 5
;   P1 ← 3
OKAY?
```

Okay! This is a reasonable explanation. Again, note that if a constant has a global name associated with it, that name is used.

```
(what p4)
;The value 9 in CELL-154 was computed in this way:
;   P4 ← (+ (+ P2 (THE B A12)) (THE B A12))
;   (THE B A12) ← (- P2 P1)
;   P2 ← 5
;   P1 ← 3
OKAY?
```

Here the result of the computation `(- p2 p1)` must be used twice in the expression describing `p4` because in fact the value was used in two different ways during the course of the computation. However, `what` avoids giving the impression that parts of the network are duplicated. Since there is no global name for the intermediate quantity, the name `(the b a12)` of the supplying pin is used to name the quantity.

```
(variable spacing)
<CELL-169 (SPACING): no value>
(== spacing (the b a12))
DONE
```

```
(defun what (cell)
  (require-cell cell)
  (cond ((not (node-boundp cell))
         (format t "~%;~S has no value.  I can express it in this way:~
                    ~:{~%;    ~S = ~S~}"
                 (cell-id cell) (tree-form cell t)))
        (t (format t "~%;The value ~S in ~S was computed in this way:~
                    ~:{~%;    ~S ← ~S~}"
                   (node-contents cell) (cell-id cell) (tree-form cell))))
  'okay?)
```

TABLE 3-14. Definition of the `what` Explanation Function.

```
(what p4)
;The value 9 in CELL-154 was computed in this way:
;    P4 ← (+ (+ P2 SPACING) SPACING)
;    SPACING ← (- P2 P1)
;    P2 ← 5
;    P1 ← 3
OKAY?
```

Once a global name has been supplied, however, `what` is happy to use it instead.

```
(what p3)
;The value 7 in CELL-153 was computed in this way:
;    P3 ← (+ P2 (- P2 P1))
;    P2 ← 5
;    P1 ← 3
OKAY?
```

On the other hand, the extra name is not used if it is not necessary to avoid duplicating expressions.

The general aim of the `what` function is to print a relevant portion of the network. If the portion contains no values, as little as possible is printed to relate the cell of interest to globally named cells and constants. If the cell of interest contains a value, then the entire computation of that value is displayed (which for a large network might still be too much, actually); nested algebraic expressions are used as much as possible, but intermediate names are introduced to denote constants and quantities which must be mentioned more than once. Whenever an intermediate quantity must be named an attempt is made to locally determine the "best" name for it; but no more global criterion is used to choose one expression over another. [McAllester 1980]

```
(defmacro nummark (cell)
  `(setf (node-mark ,cell)
         (if (numberp (node-mark ,cell)) (+ (node-mark ,cell) 1) 1)))
(defmacro unnummark (cell) `(setf (node-mark ,cell) ()))
(defmacro nummarkp (cell) `(numberp (node-mark ,cell)))
(defmacro singlenummarkp (cell) `(equal (node-mark ,cell) 1))

(defprop adder ((c (+ a b)) (b (- c a)) (a (- c b))) treeforms)
(defprop multiplier ((c (* a b)) (b (// c a)) (a (// c b))) treeforms)
(defprop maxer ((c (max a b)) (b (arcmax c a)) (a (arcmax c b))) treeforms)
(defprop minner ((c (min a b)) (b (arcmin c a)) (a (arcmin c b))) treeforms)
(defprop equality ((p (= a b)) (b (arc= p a)) (a (arc= p b))) treeforms)
(defprop gate ((p (0-if-unequal a b)) (b (-> p a)) (a (-> p b))) treeforms)

(defun tree-form (cell &optional (shallow ()))
  (require-cell cell)
  (nummark cell)
  (prog2 (tree-form-trace cell shallow)
         (tree-form-gather cell shallow)
         (tree-form-unmark cell)))
```

TABLE 3-15. The `tree-form` Function and Macros for Numerical Marks.

3.4.2. Choosing a Subgraph is Guided by Dependencies and Better-Name Heuristics

The code for `what` itself is fairly simple (Table 3-14), but it uses a rather complicated marking routine `tree-form`. This routine traces out an appropriate subgraph of the network graph, marking it out as it goes, then copies the subgraph in the form of a LISP s-expression (actually a list of equations), and finally cleans up by resetting all mark bits.

The `tree-form` function, like `fast-premises` and `desired-premises`, uses graph-marking techniques. In this case, however, a three-state mark bit is required, because it is of interest to know whether a node has been visited not at all, once, or more than once. Thus the mark may be thought of as a reference count, possibly with a ceiling at 2. The macros `nummark`, `unnummark`, and `nummarkp` (Table 3-15) implement such a reference count (with no ceiling) using the mark component of the node's repository. The normal value of such a counter should be zero, but for compatibility with the other graph-tracing routines these macros arrange to treat false as zero.

Also in Table 3-15 is a little data base of algebraic forms to use when copying portions of the network. Associated with each constraint-type is a table which, for each pin of a constraint, illustrates how to represent that pin in terms of other pins (not necessarily all the others, though that is so for our example constraint-types) in LISP prefix form. Thus for an adder the `c` pin can be represented as the sum of `a` and `b`, the `b` pin as the difference of `c` and `a`, and so on.[7]

7. Actually, these formats might more usefully be associated with individual rules to handle special cases—for example, in the case of multiplication by zero, the value of the other input need not enter into the expression. However, it was done this way (the "kludge it in quickly by hanging it from a property list" technique) so as not to have to again revise the format of `defrule`; after all, this data base is a specialized one purely for the benefit of `what`.)

The function `tree-form` first marks the given node by using `nummark`; then it calls `tree-form-trace` to trace out a subgraph of interest; next it uses `tree-form-gather` to copy the traced-out subgraph as a LISP list of equations; and finally it asks `tree-form-unmark` to clean up the marker counts.

The function `tree-form-trace` (Table 3-16) recursively marks out a subgraph explaining the value or non-value of the given node. The general idea is that if the node has a value, then we are tracing out, by following the supplier chain, the computation which produced the value; but if the node has no value, then we trace out any single potential computation.

The node given to `tree-form-trace` must already have been marked by the caller, and determined by the caller to have been the first time that node was marked. (This is trivially the case for the top-level invocation of `tree-form-trace` within `tree-form`.) The `shallow` flag indicates whether or not a full tracing out is desired: if it is set, the tracing may stop when a node with a value or a global name is encountered; but if not, then it must proceed until it can go no farther. If the node has a value, then the supplier is examined. If it has an owner, then it is a pin of that owner, and the computations for the sources (the input pins for the rule that computed the value for the supplier) are recursively traced, unless the `shallow` flag is true. If the supplier has no owner, it is a constant, and as a special kludge it is marked again; the effect of this is to make it appear to be visited more than once, which will cause `tree-form-gather` to try to find a name for it (see below). If the node has no value, then a supplier is artificially and arbitrarily chosen for the cell. If the shallow flag is set, then the artificial supplier will preferably be a global cell; if not, then preferably a pin of a constraint. If no other cell of the node will do, then as a last resort the given cell is deemed to be its own supplier; if it is a pin, then `tree-form-deep-trace` is called to trace its sources.

The function `tree-form-trace-set` takes a constraint and a list of pin names and traces from all the pins named. It first marks each of the pins, and adds those visited for the first time to a queue; then all the nodes on the queue are traced. It is very important that all pins be marked before any are traced—otherwise circular explanations can arise. The function `tree-form-tag` marks a node, then returns a list of the node (i.e., of its representative cell) if the node had previously been unmarked, and otherwise returns an empty list.

Digression. Looking back on it, `tree-form-tag` might have been written more simply as:

```
(defun tree-form-tag (cell)
  (nummark cell)
  (and (singlenummarkp cell) (list cell)))
```

I wonder why I did it the more complicated way? Probably because I was thinking of the cell as being unmarked before the visit, rather than as begin singly marked after the visit. I have decided to show both versions here to indicate how one's point of view can affect the way a program is written.

(*End of digression.*)

```
(defun tree-form-trace (cell shallow)
  (require-cell cell)
  (cond ((node-boundp cell)
         (let ((s (node-supplier cell)))
           (cond ((cell-owner s)
                  (or shallow
                      (tree-form-trace-set (cell-owner s)
                                           (get (node-rule s) 'trigger-names)
                                           shallow)))
                 (t (nummark cell))))))                                    ;crock
        (t (let ((cells (node-cells cell)))
             (setf (node-supplier cell)
                   (or (if shallow
                           (or (tree-form-shallow cell cells)
                               (tree-form-deep cell cells shallow))
                           (or (tree-form-deep cell cells shallow)
                               (tree-form-shallow cell cells)))
                       (if (cell-owner cell)
                           (tree-form-deep-trace cell shallow)
                           cell)))))))

(defun tree-form-trace-set (owner names shallow)
  (do ((n names (cdr n))
       (queue '() (nconc (tree-form-tag (*the (car n) owner)) queue)))
      ((null n) (dolist (c queue) (tree-form-trace c shallow)))))

(defun tree-form-tag (cell)
  (and (not (prog1 (nummarkp cell) (nummark cell)))
       (list cell)))

(defun tree-form-shallow (cell cells)
  (do ((c cells (cdr c)))
      ((null c) ())
    (and (not (eq (car c) cell))
         (globalp (car c))
         (return (car c)))))

(defun tree-form-deep (cell cells shallow)
  (do ((z cells (cdr z)))
      ((null z) ())
    (and (not (eq (car z) cell))
         (cell-owner (car z))
         (return (tree-form-deep-trace (car z) shallow)))))

(defun tree-form-deep-trace (cell shallow)
  (let ((treeform
          (cadr (assq (cell-name cell)
                      (get (ctype-name (con-ctype (cell-owner cell)))
                           'treeforms)))))
    (tree-form-trace-set (cell-owner cell) (cdr treeform) shallow)
    cell))
```

TABLE 3-16. Tracing Out a Subgraph of Interest for `what`.

The function `tree-form-shallow` tries to find a global cell in the current node which is not the given cell. Similarly, `tree-form-deep` tries to find a pin in the current node other than

the given cell (and if it finds one, it recursively traces the sources, using `tree-form-deep-trace`, which determines the sources by looking at the `treeforms` data base.) Each of these functions returns the desirable pseudo-supplier cell, or false if no desirable cell is found. In this way they signal success or failure, and the LISP `or` construct can be used to try one method and then another in `tree-form-trace`.

The function `tree-form-gather` (Table 3-17) retraces the subgraph, starting from the same cell that `tree-form-trace` did, and copies the traced subgraph. It returns a list of equations. Each equation is represented as a list of the left-hand side and the right-hand side. The left-hand side is always the name of a cell; the right-hand side is a formula. Hence, taken in the correct order (roughly last to first, though this property is not guaranteed), these equations represent a set of FORTRAN-style assignment statements for the computation.

The general idea is that the tree traced is copied as an algebraic expression. Whenever a node is encountered which has been marked more than once, then the computation for that node must be expressed as a separate equation defining a name for that node; then the name for that node can be used in other equations to represent that node. This is necessary to avoid duplication of shared sub-computations. Nodes which have been marked more than once are called *cuts*, because they divide the traced subgraph into portions which are trees in the strict sense. Each of these strict trees can be represented by a nested algebraic expression, but each cut requires a new equation. (The kludge in `tree-form-trace` mentioned above, where a constant is purposely marked an extra time, is to delude `tree-form-gather` (actually `tree-form-chase`) into thinking that the constant is a cut; this will force it to try to create an extra equation in order to give a name to the constant.)

A queuing mechanism is used within `tree-form-gather`. The variable `*cuts*` contains a queue of nodes which are cuts and which have yet to have equations computed for them. At each iteration one cut node is dequeued and the strict tree it heads is recursively copied by `tree-form-chase`, which when it reaches the leaves of the struct tree may enqueue other cut nodes. The variable `*allcuts*` contains the set of all nodes ever enqueued onto `*cuts*`; this is used to prevent the same cut node from being enqueued more than once. The list `*extra-equations*` is a list of equations added to by `tree-form-chase` when an equation to name a constant must be created. This is kept as a separate list rather than adding these extra equations directly to the list in `equations` purely so that all such named constants will appear at the end of the final list of equations.

The first thing `tree-form-chase` does is check the supplier of the given node; this may be a true supplier (if the node has a value), or an artificial supplier (if it does not). If the node has a value and the `shallow` flag is set, then the value itself represents the node. Now `top` is a flag indicating whether or not this call to `tree-form-chase` is on the root of a strict tree (which may occur if the node is the node given to `what`, or if the node is a cut node). Only if this is not the top

```
(declare (special *cuts* *allcuts* *extra-equations*))

(defun tree-form-gather (cell shallow)
  (require-cell cell)
  (do ((*cuts* (list cell))
       (*allcuts* (list cell))
       (equations '())
       (*extra-equations* '()))
      ((null *cuts*) (nreverse (append *extra-equations* equations)))
    (let ((cut (pop *cuts*)))
      (push (list (cell-goodname cut) (tree-form-chase cut shallow t))
            equations))))

(defun tree-form-chase (cell shallow top)
  (require-cell cell)
  (let ((s (node-supplier cell)))
    (cond ((and shallow (node-boundp cell)) (node-contents cell))
          ((and (not top) (not (singlenummarkp s)))
           (cond ((constantp s)
                  (do ((c (node-cells s) (cdr c)))
                      ((null c) (node-contents s))
                    (cond ((globalp (car c))
                           (cond ((not (memq (car c) *allcuts*))
                                  (push (car c) *allcuts*)
                                  (push (list (cell-name (car c)) (node-contents s))
                                        *extra-equations*)))
                           (return (cell-name (car c)))))))
                 (t (let ((best (do ((c (node-cells s) (cdr c)))
                                    ((null c) s)
                                  (and (not (eq (car c) s))
                                       (globalp (car c))
                                       (return (car c))))))
                      (cond ((and (not (and (eq best s) (globalp s)))
                                  (not (memq best *allcuts*)))
                             (push best *allcuts*)
                             (push best *cuts*)))
                      (cell-goodname best)))))
          ((cell-owner s)
           (let ((treeform (cadr (assq (cell-name s)
                                       (get (ctype-name (con-ctype (cell-owner s)))
                                            'treeforms)))))
             (cons (car treeform)
                   (forlist (n (cdr treeform))
                     (cond ((and (node-boundp s)
                                 (not (memq n (get (node-rule s) 'trigger-names))))
                            '?)
                           (t (tree-form-chase (*the n (cell-owner s))
                                               shallow
                                               ()))))))))
          ((globalp s) (cell-name s))
          (t (node-contents s)))))
```

TABLE 3-17. Copying a Traced Subgraph as a Set of Equations.

node does `tree-form-chase` want to check for its being a cut node (visited more than once by `tree-form-trace`). If it is a cut node other than the top, then the supplier may be a constant or

something else. If a constant, then an attempt is made to find a global cell within the same node, to serve as a name for the constant. If one is found, that is returned, and an equation identifying the name with the constant is added to `*extra-equations*` (if it has not already been added); but if one is not found, the constant itself is returned. If the cut node's supplier is not a constant, then a similar search for a global name is made; either such a name, or else the supplier's name (which if the supplier is a pin looks like `(the foo bar)`), is chosen to name the cut node. The node is queued as a cut node if it has not already ever been queued *and* if the supplier is not a global cell artificially chosen to be the supplier (in which case there is no way to express that name in terms of something else, so no equation is needed).

If the node is not to be treated as a cut node, then there are three cases: the supplier may be a constant, a global cell, or a pin. For a constant the value is returned; because `tree-form-trace` always marks nodes with constant suppliers as cut nodes, this case can only arise when the `top` flag is set. For a global cell, its name is returned; this can only occur when the global cell has been chosen as an artifical supplier. For a pin, the associated expression form is fetched from the `treeforms` data base; then each input pin named in the expression form is recursively chased and the resulting expression filled in as a sub-expression of the treeform for this node. An exception is that if this node has a value, then the rule used to compute it is examined, and if an input pin named in the expression form was not actually a trigger for the rule, then "?" is filled in for that pin in the form. This is a crude attempt to take into account things like the multiplication-by-zero rule. For example:

```
(create m multiplier)
<M:MULTIPLIER-44>
(what (the c m))
;CELL-47 has no value.  I can express it in this way:
;   (THE C M) = (* (THE A M) (THE B M))
;   (THE B M) = (// (THE C M) (THE A M))
;   (THE A M) = (// (THE C M) (THE B M))
OKAY?
```

(This is an example of a strange case that occurs when a node is a "dead end" (a leaf of the computation tree) for the `tree-form-trace` search, and the node's sole cell is a pin. The pin must serve as its own supplier, and so `tree-form-chase` believes that this supplier must be expressed as an expression. If global names were given to `(the a m)` and `(the b m)` then this anomaly would not occur. Alternatively, the `tree-form` code could be made more complex, but I didn't feel like doing this.)

```
(== (the a m) (constant 0))
;|Awakening <M:MULTIPLIER-44> because its A got the value 0.
;|<M:MULTIPLIER-44> computed 0 for its part C from pin A.
;|Awakening <M:MULTIPLIER-44> because its C got the value 0.
DONE
```

```
(defun tree-form-unmark (cell)
  (require-cell cell)
  (cond ((nummarkp cell)
         (unnummark cell)
         (let ((s (node-supplier cell)))
           (and (cell-owner s)
                (dolist (pin (con-values (cell-owner s)))
                  (tree-form-unmark pin))))
         (or (node-boundp cell) (setf (node-supplier cell) ()))))
```

TABLE 3-18. Resetting the Mark Components for `tree-form`.

```
(what (the c m))
;The value 0 in CELL-47 was computed in this way:
;   (THE C M) ← (* 0 ?)
OKAY?
```

The explanation indicates that `(the c m)` is the product of zero and something we don't much care about (because the rule didn't).

```
(what (the b m))
;CELL-46 has no value.  I can express it in this way:
;   (THE B M) = (// 0 0)
OKAY?
```

Certainly `(the b m)` = `0/0`; of course, this defines no particular value, but then again, `(the b m)` indeed has no particular value.

The function `tree-form-unmark` (Table 3-18) is similar to `fast-premises-unmark` (Table 3-11) and `desired-premises-unmark` (Table 3-12), with the additional feature that if a marked node is encountered which has no value, then that node's supplier is reset to the null supplier; this removes the artificial suppliers introduced by `tree-form-trace`.

3.5. Summary of Some Uses for Dependencies

The facilities described in this chapter illustrate the recording and some the uses of dependency information. Recording dependencies amounts to remembering the history of the local propagation. All such histories must be embedded within the structure of the constraint network; the computation history can be represented as the directions of information flow within the network, plus the computational rules used to compute new values.

Even if a computation has not been performed, or has propagated values to only as part of the network, the structure of the network can be used to advantage, because it describes all possible potential histories. These potential histories, if few in number or carefully chosen among, may be useful for analyzing the situation.

Three procedures for examining a constraint network have been exhibited. The function `why` traces a single actual computation step or all potential single computation steps. The function `why-ultimately` traces through an entire actual computation tree (actually a dag—a tree, possibly with shared subtrees) or through all possible potential trees, in effect, and exhibits the leaves of the trees. The function `what` traces through an entire actual computation tree or a single, carefully chosen potential tree, and exhibits that tree as an algebraic expression, indicating shared subtrees by naming them and then defining the name once to be the shared sub-expression.

There are other ways of using the dependency information. For example, the notion dual to that of the set of premises is known as the set of *repercussions*—it is the set of nodes which have a given node as an ancestor and which are not ancestors of any other nodes. In other words, the repercussions are the ultimate consequences of a node. Functions for tracing through the network and locating the repercussions and then printing the findings in a manner similar to `why-ultimately` or `what` would be useful.

Little fishies in the brook.
All they do is look and look.
Christmas comes but once a year.
My dad drives a peanut wagon!

—Cordon Ruthven Kerns

Chapter Four

Retraction

IN THE CONSTRAINT SYSTEM developed in Chapters Two and Three, if a constant is mistakenly connected to the wrong node, that's too bad; the user must start over from scratch. If the user wants to use a single network to explore several cases, to tinker with parameters to see the resulting effects, that also is too bad. Once a value has been computed for a node, it is fixed for all time. Trying to change it will only produce a contradiction, causing the signalling of an irrecoverable error.

Mechanisms for retracting values will be developed in this chapter. This is not simply a matter of throwing away old values and installing new ones. Values associated with other nodes may have been computed from the retracted one, and these must also be retracted. Moreover, when a contradiction occurs, it may be "far from the scene of the crime"; the contradiction may involve not premises but derived values. Retracting a derived value does no good—the same value will be recomputed from the premises. Instead, one of the premises of one of the contradictory values must be retracted. Of course, the premise-tracing machinery developed in Chapter Three will be of use for this.

4.1. Forgiving Systems

Many computer systems require a great deal of forgiving (though few deserve it), but here I mean that the system *is* forgiving—of mistakes, changes of mind, and so on. Ideally one would like to be able to invert any action with an appropriate counter-action, and have the state of the system

be as if the action had never taken place. Here follow examples of the system allowing the user to change his mind when some action causes a contradiction.

4.1.1. Connecting Conflicting Cells Can Cause Contradictions

Let us re-enact the example of §2.3, where we constructed a temperature conversion network, equated `centigrade` to −40 (which of course computed −40 for `fahrenheit`), and then equated `fahrenheit` to 32.

```
(defun temp-converter ()
    (create add adder)
    (create mult multiplier)
    (create othermult multiplier)
    (variable fahrenheit)
    (variable centigrade)
    (== fahrenheit (the c add))
    (== (the b add) (constant 32))
    (== (the a add) (the a othermult))
    (== (the c othermult) (the c mult))
    (== (the b othermult) (constant 5))
    (== centigrade (the b mult))
    (== (the a mult) (constant 9)))
TEMP-CONVERTER
```

This is the same definition for a temperature converter we used in Chapters Two and Three, packaged up as a single LISP function.

```
(temp-converter)
;|Awakening <ADD:ADDER-23> because its B got the value 32.
;|Awakening <OTHERMULT:MULTIPLIER-31> because its B got the value 5.
;|Awakening <MULT:MULTIPLIER-27> because its A got the value 9.
DONE
```

Now that the network has been instantiated, we equate `centigrade` to −40. Rather than using the `constant` construct, however, we shall use `default` instead. This has roughly the same effect; the only difference is that a default is tentative, while a constant is relatively fixed (but only relatively).

```
(== centigrade (default -40))
;|Awakening <MULT:MULTIPLIER-27> because its B got the value -40.
;|<MULT:MULTIPLIER-27> computed -360 for its part C from pins A, B.
;|Awakening <OTHERMULT:MULTIPLIER-31> because its C got the value -360.
;|<OTHERMULT:MULTIPLIER-31> computed -72 for its part A from pins B, C.
;|Awakening <ADD:ADDER-23> because its A got the value -72.
;|<ADD:ADDER-23> computed -40 for its part C from pins A, B.
```

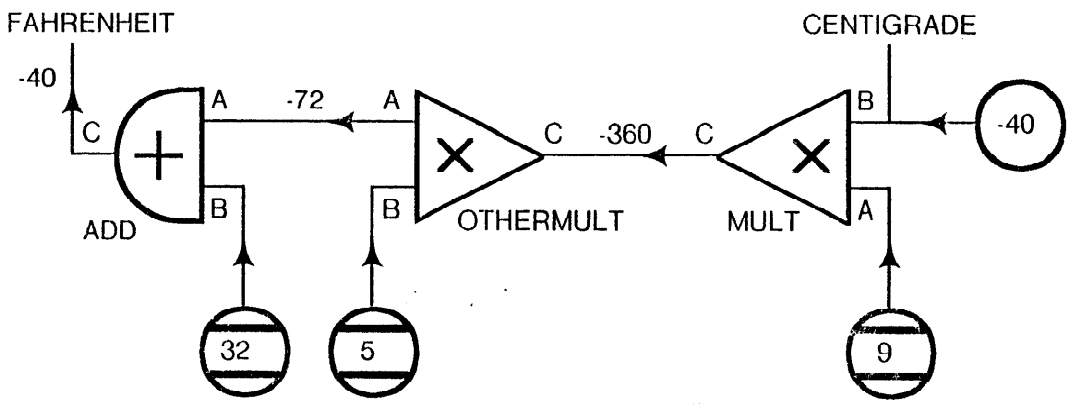


FIGURE 4-1. Computation of a Temperature Conversion, Using a Default Value.

```
;|Awakening <ADD:ADDER-23> because its C got the value -40.
;|Awakening <OTHERMULT:MULTIPLIER-31> because its A got the value -72.
;|Awakening <MULT:MULTIPLIER-27> because its C got the value -360.
DONE
```

This process of course has computed —40 for `fahrenheit`. (See Figure 4-1. As before, circles with horizontal bars indicate `constant` values; in addition, `default` values are drawn as plain circles.)

```
(what fahrenheit)
;The value -40 in CELL-35 was computed in this way:
;   FAHRENHEIT ← (+ (// (* 9 CENTIGRADE) 5) 32)
;   CENTIGRADE ← -40
OKAY?
```

Now we come to the critical point we left off at in §2.3: what happens if `fahrenheit` is not equated to 32?

```
(== fahrenheit (default 32))

;;; Contradiction when merging the cells
;         <CELL-35 (FAHRENHEIT): -40> and <CELL-41 (DEFAULT): 32>.
;;; These are the premises that seem to be at fault:
;        <CELL-40 (DEFAULT): -40> == CENTIGRADE,
;        <CELL-41 (DEFAULT): 32>.
;;; Choose one of these to retract and RETURN it.
;BKPT Choose Culprit
```

As before, the system has detected a conflict between the given value and the value previously computed for `fahrenheit`. This time, however, the system has determined the premises of the conflicting values and listed them, and asked us to choose which one to retract. (Note that the name of a `default` cell is "`default`", not "?". Similarly, later we shall see that the name of a `constant` cell is "`constant`".)

Now we are within a "breakpoint", within which we may interact with the LISP system in the usual manner. In particular, we could invoke the `why` and `what` functions to explore the network before choosing a culprit.[1] When eventually a form `(return form)` is typed, then the breakpoint level is exited, and the value of *form* is returned and made available to the program which caused the breakpoint. We will do this now: let us choose to retract the value for `centigrade`.

```
(return centigrade)
;|Retracting the premise <CELL-40 (DEFAULT): -40>.
;|Removing -40 from CELL-40.
;|Removing -360 from (THE C MULT) because (THE B MULT)==CELL-40.
;|Removing -72 from (THE A OTHERMULT) because (THE C OTHERMULT)==(THE C MULT).
;|Removing -40 from (THE C ADD) because (THE A ADD)==(THE A OTHERMULT).
;|Awakening <ADD:ADDER-23> because its C lost its value.
;|Awakening <ADD:ADDER-23> because its A lost its value.
;|Awakening <OTHERMULT:MULTIPLIER-31> because its A lost its value.
;|Awakening <OTHERMULT:MULTIPLIER-31> because its C lost its value.
;|Awakening <MULT:MULTIPLIER-27> because its C lost its value.
;|Awakening <MULT:MULTIPLIER-27> because its B lost its value.
;|Awakening <ADD:ADDER-23> because its C got the value 32.
;|<ADD:ADDER-23> computed 0 for its part A from pins B, C.
;|Awakening <ADD:ADDER-23> because its A got the value 0.
;|Awakening <OTHERMULT:MULTIPLIER-31> because its A got the value 0.
;|<OTHERMULT:MULTIPLIER-31> computed 0 for its part C from pins A, B.
;|Awakening <OTHERMULT:MULTIPLIER-31> because its C got the value 0.
;|Awakening <MULT:MULTIPLIER-27> because its C got the value 0.
;|<MULT:MULTIPLIER-27> computed 0 for its part B from pins A, C.
;|Awakening <MULT:MULTIPLIER-27> because its B got the value 0.
DONE
```

Whew! After the value —40 was retracted for `centigrade`, all of the values which were computed from that value were recursively removed. Then all the constraint devices which had values retracted from their pins were awakened, in the hope that they could provide a value for the

1. This is yet another example of how pleasant it is to implement a toy system by embedding it in a more complete system such as LISP. Rather than having the interaction limited to merely a choice from a menu, the user can be given the opportunity to perform any computation before making the decision.

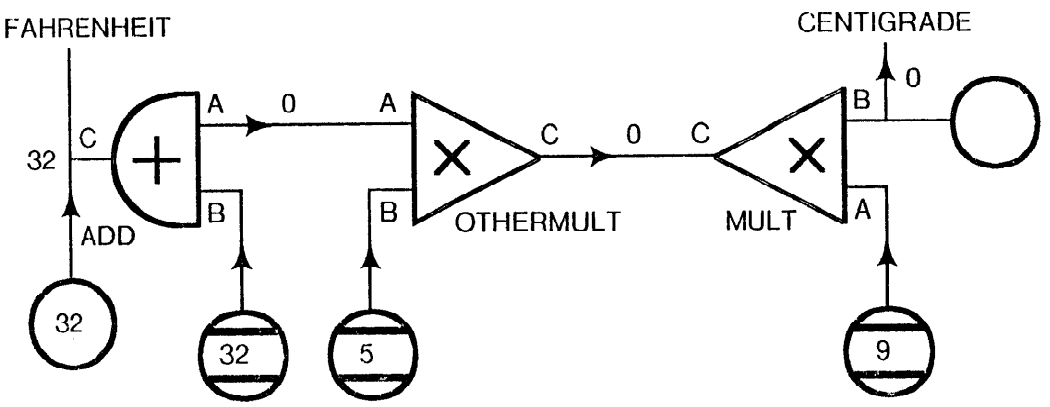


FIGURE 4-2. Recomputation of a Temperature Conversion.

retracted-from pin.[2] Next the new value 32 for `fahrenheit` was installed, and the usual process of local propagation proceeded from there. The result is pictured in Figure 4-2. We can ask `what` about the value in `fahrenheit`.

```
(what fahrenheit)
;The value 32 in CELL-35 was computed in this way:
;    FAHRENHEIT ← 32
OKAY?
(what centigrade)
;The value 0 in CELL-36 was computed in this way:
;    CENTIGRADE ← (// (* (- FAHRENHEIT 32) 5) 9)
;    FAHRENHEIT ← 32
OKAY?
```

Everything is just as if we had originally given the value 32 to `fahrenheit`, and had never given the value −40 to `centigrade` in the first place.

When the contradiction occurred above, the computed value for `fahrenheit` had been derived not only from the value for `centigrade`, but also from the constants 5, 9, and 32. However, these constants were not listed among the choices for retraction. This illustrates the one difference between the `constant` and `default` constructs—if a contradiction is derived from at least one `default` value, then only `default` values are considered for retraction. If the

2. Presumably this would be the same value that was retracted. However, a constraint device might be an subsidiary supplier of the value rather than the primary supplier. When the primary supplier is then retracted, subsidiary suppliers then have a chance to become the primary supplier. Thus, rather than recording multiple justifications for a value, as in [Stallman 1977], [Doyle 1978a], [Doyle 1978b], [McAllester 1978], and [Doyle 1979], this system merely recomputes value when necessary (presumably involving only a single computation step in each case).

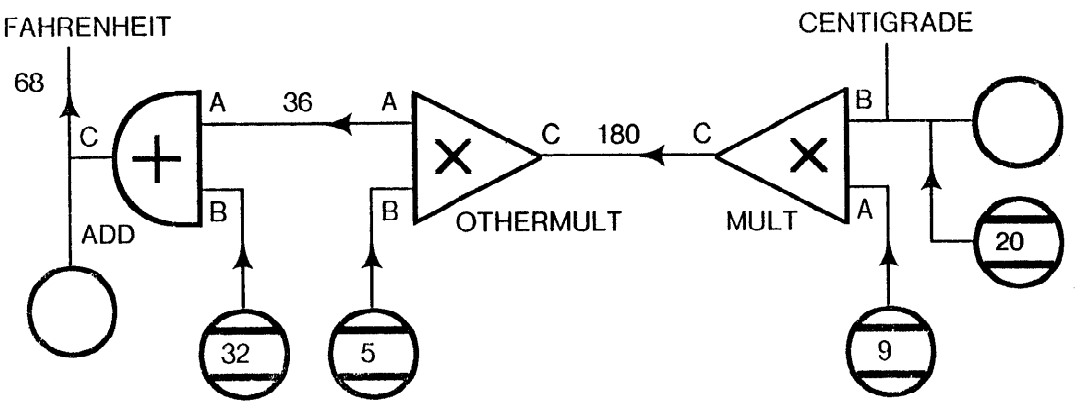


FIGURE 4-3. Another Recomputation of a Temperature Conversion.

contradiction rests solely on `constant` values, however, then the system will consider retracting a constant.[3] To further illustrate the distinction, let us now fix `centigrade` as the constant 20.

```
(== centigrade (constant 20))

;;; Contradiction when merging the cells
;         <CELL-36 (CENTIGRADE): 0> and <CELL-42 (CONSTANT): 20>.
;|Retracting the premise <CELL-41 (DEFAULT): 32>.
;|Removing 32 from CELL-41.
;|Removing 0 from (THE A ADD) because (THE C ADD)==CELL-41.
;|Removing 0 from (THE C OTHERMULT) because (THE A OTHERMULT)==(THE A ADD).
;|Removing 0 from (THE B MULT) because (THE C MULT)==(THE C OTHERMULT).
;|Awakening <MULT:MULTIPLIER-27> because its B lost its value.
;|Awakening <OTHERMULT:MULTIPLIER-31> because its C lost its value.
;|Awakening <MULT:MULTIPLIER-27> because its C lost its value.
;|Awakening <ADD:ADDER-23> because its A lost its value.
;|Awakening <OTHERMULT:MULTIPLIER-31> because its A lost its value.
;|Awakening <ADD:ADDER-23> because its C lost its value.
;|Awakening <MULT:MULTIPLIER-27> because its B got the value 20.
;|<MULT:MULTIPLIER-27> computed 180 for its part C from pins A, B.
;|Awakening <OTHERMULT:MULTIPLIER-31> because its C got the value 180.
;|<OTHERMULT:MULTIPLIER-31> computed 36 for its part A from pins B, C.
;|Awakening <ADD:ADDER-23> because its A got the value 36.
;|<ADD:ADDER-23> computed 68 for its part C from pins A, B.
;|Awakening <ADD:ADDER-23> because its C got the value 68.
;|Awakening <OTHERMULT:MULTIPLIER-31> because its A got the value 36.
;|Awakening <MULT:MULTIPLIER-27> because its C got the value 180.
DONE
```

3. One might argue that this should constitute a hard error as before. However, the ability to retract a constant is not difficult to provide, and the system can inform the user of the situation and let him decide whether or not to tamper with a "constant of the universe".

It doesn't show clearly here, but in fact the conflict is between the new value 20, a constant, and the old value 0, which was derived from various constants and a single `default` value. The system concluded that as there was precisely one `default` value involved, it might as well retract it automatically and not even bother us with it.

```
(what fahrenheit)
;The value 68 in CELL-35 was computed in this way:
;   FAHRENHEIT ← (+ (// (* 9 CENTIGRADE) 5) 32)
;   CENTIGRADE ← 20
OKAY?
```

Now `fahrenheit` = 68, computed from `centigrade` = 20 (Figure 4-3).

If we now try to equate `fahrenheit` to a `default` value, the system "bounces back". The default value comes into conflict with hard constants, and so is immediately rejected.

```
(== fahrenheit (default 41))

;;; Contradiction when merging the cells
;         <CELL-35 (FAHRENHEIT): 68> and <CELL-43 (DEFAULT): 41>.
;|Retracting the premise <CELL-43 (DEFAULT): 41>.
;|Removing 41 from CELL-43.
DONE
```

If we really want `fahrenheit` to be 41, we had better fight constants with constants!

```
(== fahrenheit (constant 41))

;;; Contradiction when merging the cells
;        <CELL-35 (FAHRENHEIT): 68> and <CELL-44 (CONSTANT): 41>.
;;; These are the premises that seem to be at fault:
;        <CELL-44 (CONSTANT): 41>,
;        <CELL-37 (CONSTANT): 32>,
;        <CELL-42 (CONSTANT): 20> == CENTIGRADE,
;        <CELL-39 (CONSTANT): 9>,
;        <CELL-38 (CONSTANT): 5>.
;;; Choose one of these to retract and RETURN it.
;BKPT Choose Culprit
```

This contradiction involves only `constant` values, and so one of them must be chosen for retraction.

```
(return centigrade)
;|Retracting the premise <CELL-42 (CONSTANT): 20>.
;|Removing 20 from CELL-42.
;|Removing 180 from (THE C MULT) because (THE B MULT)==CELL-42.
;|Removing 36 from (THE A OTHERMULT) because (THE C OTHERMULT)==(THE C MULT).
;|Removing 68 from (THE C ADD) because (THE A ADD)==(THE A OTHERMULT).
```

```
;|Awakening <ADD:ADDER-23> because its C lost its value.
;|Awakening <ADD:ADDER-23> because its A lost its value.
;|Awakening <OTHERMULT:MULTIPLIER-31> because its A lost its value.
;|Awakening <OTHERMULT:MULTIPLIER-31> because its C lost its value.
;|Awakening <MULT:MULTIPLIER-27> because its C lost its value.
;|Awakening <MULT:MULTIPLIER-27> because its B lost its value.
;|Awakening <ADD:ADDER-23> because its C got the value 41.
;|<ADD:ADDER-23> computed 9 for its part A from pins B, C.
;|Awakening <ADD:ADDER-23> because its A got the value 9.
;|Awakening <OTHERMULT:MULTIPLIER-31> because its A got the value 9.
;|<OTHERMULT:MULTIPLIER-31> computed 45 for its part C from pins A, B.
;|Awakening <OTHERMULT:MULTIPLIER-31> because its C got the value 45.
;|Awakening <MULT:MULTIPLIER-27> because its C got the value 45.
;|<MULT:MULTIPLIER-27> computed 5 for its part B from pins A, C.
;|Awakening <MULT:MULTIPLIER-27> because its B got the value 5.
DONE
```

The constant 20 equated to `centigrade` has been retracted, and a new value derived from `fahrenheit`.

```
(what centigrade)
;The value 5 in CELL-36 was computed in this way:
;   CENTIGRADE ← (// (* (- FAHRENHEIT 32) 5) 9)
;   FAHRENHEIT ← 41
OKAY?
```

4.1.2. Propagation Potentially Poses Problems for Predefined Pins

Contradictions can arise not only when equating two nodes, but also when a constraint device calculates a value for a pin in conflict with an existing value. As an example of this, consider the network of Figure 3-3 (page 92) for constraining four points to be equally spaced. Assume that one has been constructed now. Then we perform the equatings $p1 = 0$, $p3 = 6$, and $p2 = 4$, which of course are not consistent.

```
(== p1 (default 0))
;|Awakening <A12:ADDER-89> because its A got the value 0.
DONE
(== p3 (default 6))
;|Awakening <A34:ADDER-97> because its A got the value 6.
;|Awakening <A23:ADDER-93> because its C got the value 6.
DONE
```

So far so good . . .

```
(== p2 (default 4))
```

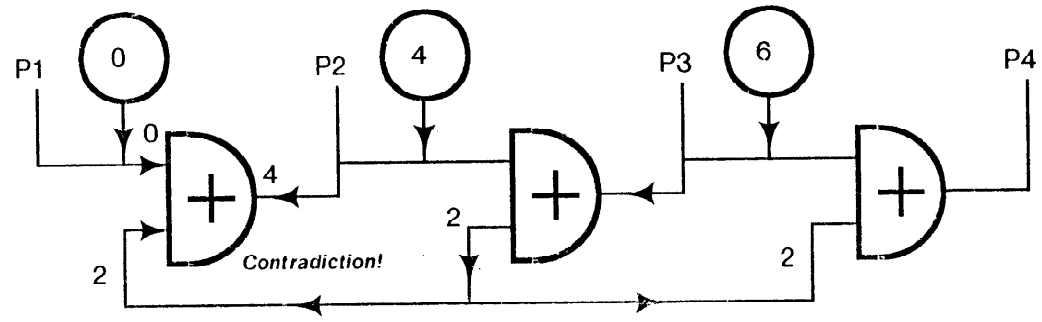


FIGURE 4-4. A Contradiction in a Four-Point Spacing Network.

```
;|Awakening <A23:ADDER-93> because its A got the value 4.
;|<A23:ADDER-93> computed 2 for its part B from pins A, C.
;|Awakening <A12:ADDER-89> because its B got the value 2.

;;; Contradiction in <A12:ADDER-89> among these parts: A=0, B=2, C=4;
;;;    it calculated 2 for A from the others by rule ADDER-RULE-3.
;;; These are the premises that seem to be at fault:
;        <CELL-101 (DEFAULT): 0> == P1,
;        <CELL-103 (DEFAULT): 4> == P2,
;        <CELL-102 (DEFAULT): 6> == P3.
;;; Choose one of these to retract and RETURN it.
```

This situation is pictured in Figure 4-4. The contradiction was caught when adder `a12` computed a value for `p1` which was not consistent with the existing default value. The system has indicated that the adder computed the value 2. Of course, the value 0 might be the correct one, and one of the premises for `p2` or `p3` might be at fault. We should examine the computations of all the pins involved.

```
(what (the a a12))
;The value 0 in CELL-90 was computed in this way:
;   (THE A A12) ← 0
OKAY?
```

Actually, in this case we would have preferred that `what` show us a global name for the constant; but it thinks that `(the a a12)` is the preferred name for the root node because that is what we gave it. Fixing this is left as an exercise for the reader.[4]

```
(what (the b a12))
;The value 2 in CELL-91 was computed in this way:
```

4. Don't you just hate it when an author leaves something as an "exercise for the reader"?

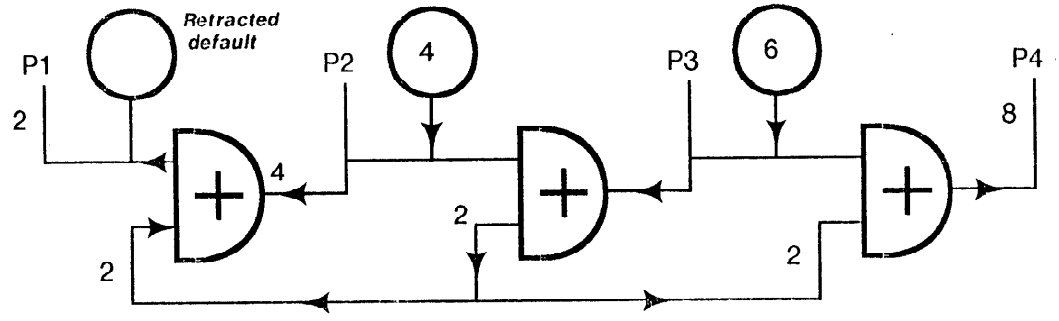


FIGURE 4-5. Defeating Two Defaults in a Four-Point Spacing Network.

```
;    (THE B A12) ← (- P3 P2)
;    P3 ← 6
;    P2 ← 4
OKAY?
```

Now, this tells us something: `(the b a12)` is the difference between `p3` and `p2`.

```
(what (the c a12))
;The value 4 in CELL-92 was computed in this way:
;    (THE C A12) ← 4
OKAY?
```

We could choose to retract any of the premises; let us in fact choose `p1`.

```
(return p1)
;|Retracting the premise <CELL-101 (DEFAULT): 0>.
...
;|Awakening <A12:ADDER-89> because its C got the value 4.
DONE
```

(Much of the tedious trace output has been omitted here; sixteen lines were deleted. Henceforth most `ctrace` output will be similarly condensed, and omissions indicated by ellipses.)

```
(what p1)
;The value 2 in CELL-85 was computed in this way:
;    P1 ← (- P2 (- P3 P2))
;    P2 ← 4
;    P3 ← 6
OKAY?
(what p4)
;The value 8 in CELL-88 was computed in this way:
;    P4 ← (+ P3 (- P3 P2))
```

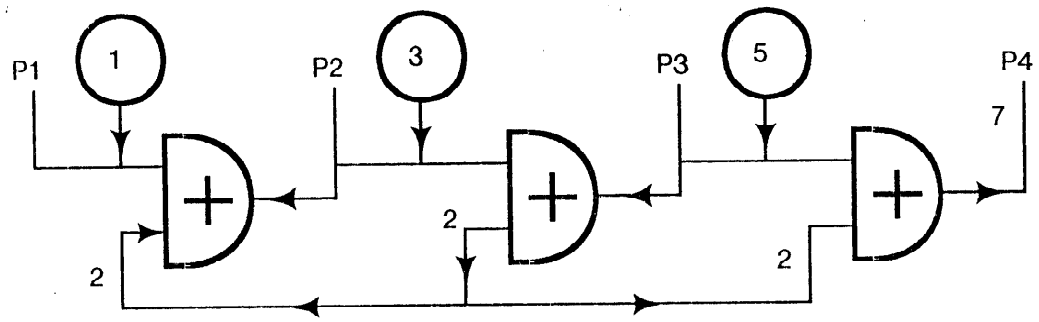


FIGURE 4-6. Redundant Premises for a Four-Point Spacing Network.

```
;    P3 ← 6
;    P2 ← 4
OKAY?
```

New values for p1 and p4 have been computed in terms of p2 and p3. (See Figure 4-5.)

Multiple contradictions can arise in a single interaction, if the network contains redundant reasons for current values. For example, in a fresh four-point-spacing network, this might occur:

```
(== p1 (default 1))
;|Awakening <A12:ADDER-69> because its A got the value 1.
DONE
(== p3 (default 5))
...
DONE
(== p2 (default 3))
...
DONE
```

The three values for p1, p2, and p3 constitute redundant information for the network. Given p1 and p2, p3 could have been deduced; given p2 and p3, p1 could have been deduced. (However, p2 was not deduced given p1 and p3 because that cannot be done by local propagation; algebra is required. Once a correct value for p2 was supplied, however, the system verified it.)

```
(what p4)
;The value 7 in CELL-68 was computed in this way:
;    P4 ← (+ P3 (- P3 P2))
;    P3 ← 5
;    P2 ← 3
OKAY?
```

The value 7 was computed for p4 (Figure 4-6). If now p4 is equated to 6, a contradiction must occur.

```
(== p4 (default 6))

;;; Contradiction when merging the cells
;       <CELL-68 (P4): 7> and <CELL-84 (DEFAULT): 6>.
;;; These are the premises that seem to be at fault:
;       <CELL-82 (DEFAULT): 5> == P3,
;       <CELL-83 (DEFAULT): 3> == P2,
;       <CELL-84 (DEFAULT): 6>.
;;; Choose one of these to retract and RETURN it.
```

Indeed, as what indicated, the old value of p4 depended on p2 and p3. We choose to retract p2.

```
(return p2)
;|Retracting the premise <CELL-83 (DEFAULT): 3>.
;|Removing 3 from CELL-83.
;|Removing 2 from (THE B A23) because (THE A A23)==CELL-83.
;|Removing 7 from (THE C A34) because (THE B A34)==(THE B A23).
;|Awakening <A34:ADDER-77> because its C lost its value.
...
;|Awakening <A34:ADDER-77> because its C got the value 6.
;|<A34:ADDER-77> computed 1 for its part B from pins A, C.
;|Awakening <A12:ADDER-69> because its B got the value 1.
;|<A12:ADDER-69> computed 2 for its part C from pins A, B.
;|Awakening <A23:ADDER-73> because its A got the value 2.

;;; Contradiction in <A23:ADDER-73> among these parts: A=2, B=1, C=5;
;;;     it calculated 4 for A from the others by rule ADDER-RULE-3.
;;; These are the premises that seem to be at fault:
;       <CELL-81 (DEFAULT): 1> == P1,
;       <CELL-82 (DEFAULT): 5> == P3,
;       <CELL-84 (DEFAULT): 6> == P4.
;;; Choose one of these to retract and RETURN it.
```

While p2 supported the value for p4, the redundant value in p1 also supported it indirectly, and now the computation has run afoul again. The system is not satisfied until the network is totally consistent. We could now change our minds and retract the assignment of 6 to p4, but here we will proceed to retract p1.

```
(return p1)
;|Retracting the premise <CELL-81 (DEFAULT): 1>.
;|Removing 1 from CELL-81.
;|Removing 1 from (THE B A12) because (THE A A12)==CELL-81.
;|Removing 2 from (THE C A12) because NIL==(THE B A12).
```

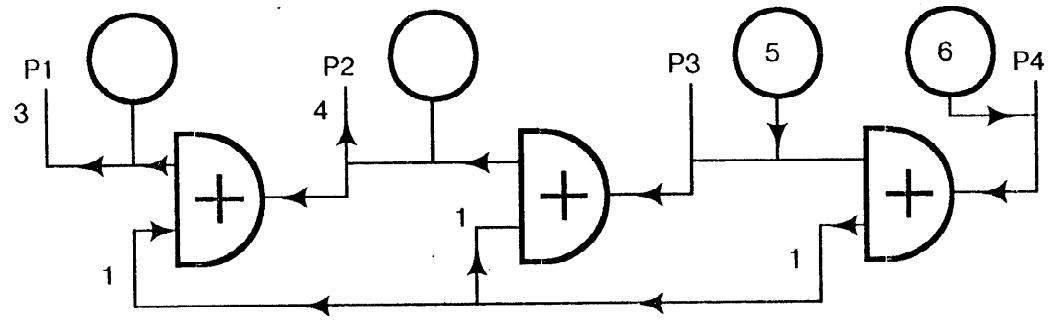


FIGURE 4-7. Surviving Two Contradictions a Four-Point Spacing Network.

```
;|Awakening <A23:ADDER-73> because its A lost its value.
...
;|Awakening <A34:ADDER-77> because its B got the value 1.
DONE
```

The end result is pictured in Figure 4-7.

4.1.3. Erroneous Equatings Elicit Execution Exceptions Equally Easily

Until now it has been implicitly assumed that the constraint network, once constructed, is fixed. All computations are relative to this fixed structure, and any errors are attributed to the chosen premises rather than to the network structure. Premises can be retracted, but not connections. The retraction mechanism exhibited above operates not by disconnecting a rejected constant cell from the rest of the network, but by "denaturing" the constant, forcibly removing its value but leaving it connected as a useless appendage (useless because it has no value and no name).

Here are introduced two functions `dissolve` and `disconnect` for undoing connections. When given a cell, `dissolve` will undo the connections among *all* the cells of the node to which the given cell belongs. On the other hand, `disconnect` causes a specified cell to be unhooked from its node, leaving any others connected together. Neither of these is a true inverse for the `==` construction; for example, if one says `(== a b)` and then `(disconnect a)`, the new situation will be identical to the original only if `a` had not previously been connected to any other cells. If it had, then those cells will now all be connected to `b` . It is impossible to provide a way to provide a true inverse for `==` given the current data structures used in the implementation, because not enough information is retained (for example, no information whatever is recorded for redundant equatings). Later we will see ways of providing for this.

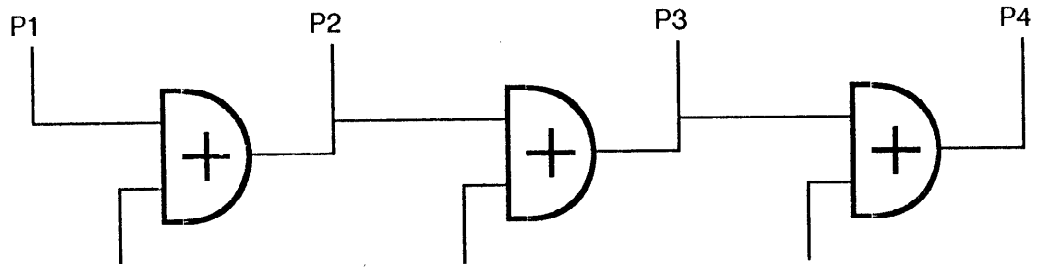


FIGURE 4-8. A Partially Dissolved Four-Point Spacing Network.

To illustrate the use of `dissolve` and `disconnect`, consider yet another fresh four-point spacing network. Initially `p4` has no value, and can be expressed in terms of `p1`, `p2`, and `p3`.

```
(what p4)
;CELL-92 has no value.  I can express it in this way:
;   P4 = (+ P3 (- P2 P1))
OKAY?
```

If the connection between the `b` pins of the three adders is now dissolved, then of course `p4` can no longer be expressed in terms of the difference between `p2` and `p1`.

```
(dissolve (the b a12))
;|Dissolving (THE B A12), (THE B A23), (THE B A34).
DONE
```

See Figure 4-8.

```
(what p4)
;CELL-92 has no value.  I can express it in this way:
;   P4 = (+ P3 (THE B A34))
OKAY?
```

If now a `default` value is given to `(the b a12)`, it will not affect `(the b a23)` because they are no longer connected.

```
(== (the b a12) (default 3))
;|Awakening <A12:ADDER-93> because its B got the value 3.
DONE
(what (the b a23))
;CELL-99 has no value.  I can express it in this way:
;   (THE B A23) = (- P3 P2)
OKAY?
```

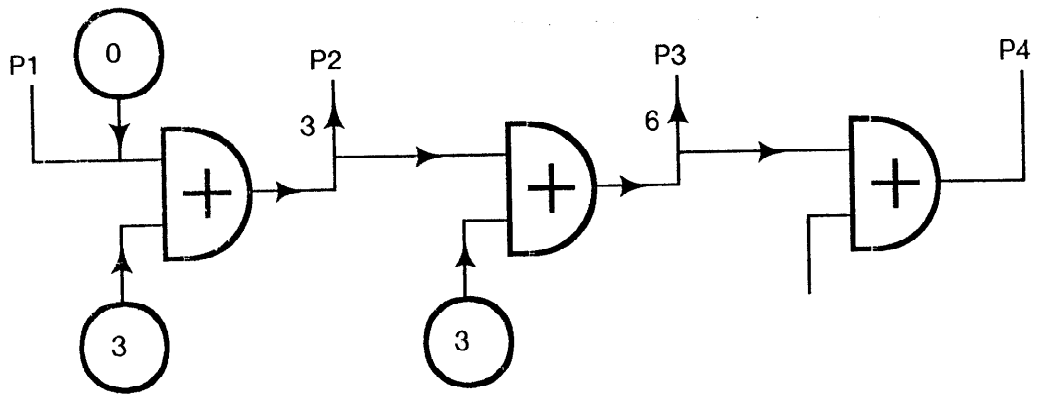


FIGURE 4-9. Computation in a Partially Dissolved Spacing Network.

Let us now also equate `(the b a23)` to 3, and `p1` to 0. This will result in the computation of 3 for `p2` and 6 for `p3`.

```
(== (the b a23) (default 3))
;|Awakening <A23:ADDER-97> because its B got the value 3.
DONE
(== p1 (default 0))
;|Awakening <A12:ADDER-93> because its A got the value 0.
...
DONE
```

The result is shown in Figure 4-9.

```
(what p3)
;The value 6 in CELL-91 was computed in this way:
;   P3 ← (+ (+ P1 3) 3)
;   P1 ← 0
OKAY?
(what p4)
;CELL-92 has no value.  I can express it in this way:
;   P4 = (+ 6 (THE B A34))
OKAY?
```

No value was computed for `p4` because `(the b a34)` still has no value. Suppose we were to connect `(the b a34)` to `p2` (which produces something other than a four-point spacing network!).

```
(== (the b a34) p2)
;|Awakening <A34:ADDER-101> because its B got the value 3.
;|<A34:ADDER-101> computed 9 for its part C from pins A, B.
```

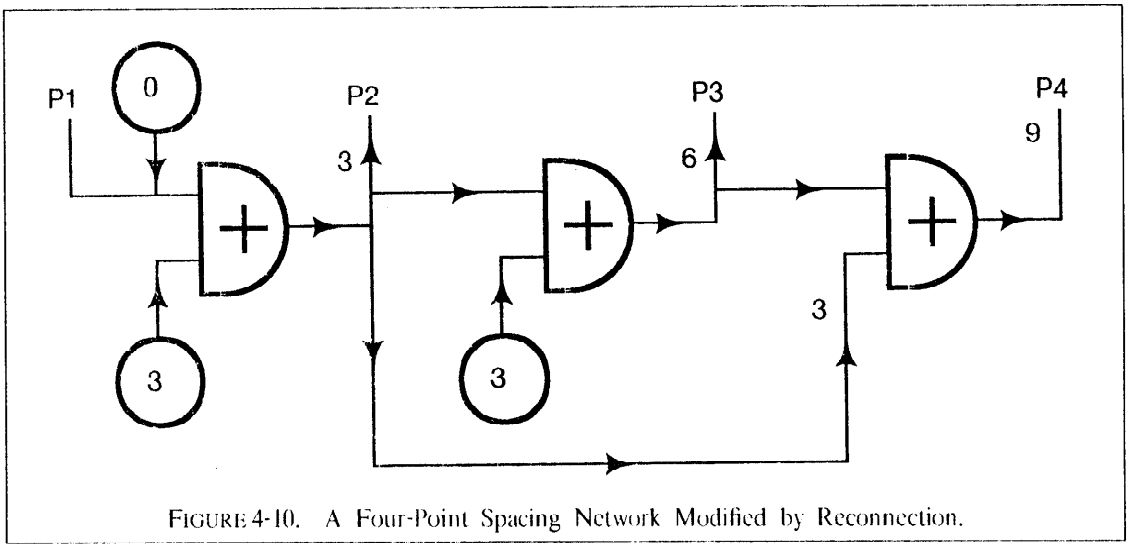


FIGURE 4-10. A Four-Point Spacing Network Modified by Reconnection.

```
;|Awakening <A34:ADDER-101> because its C got the value 9.
DONE
```

See Figure 4-10.

```
(what p4)
;The value 9 in CELL-92 was computed in this way:
;    P4 ← (+ (+ P2 3) P2)
;    P2 ← (+ P1 3)
;    P1 ← 0
OKAY?
```

Now `p4` has the value 9 expected for equal spacing, but that value was computed in a rather unorthodox fashion! Let us undo the connection.

```
(disconnect (the b a34))
;|Disconnecting (THE B A34) from (THE A A23), (THE C A12), P2.
;|Removing 3 from (THE B A34).
;|Removing 9 from (THE C A34) because of NIL.
;|Awakening <A34:ADDER-101> because its C lost its value.
;|Awakening <A34:ADDER-101> because its B lost its value.
DONE
```

When `(the b a34)` was disconnected from the `p2` node, it was disconnected from the source of its value. Hence the value 3 was removed from it, and thus the value 9 derived from it was also removed from `(the c a34)==p4`. The network has now been restored to the situation of Figure 4-9.

```
(== (the b a34) (the b a23))
```

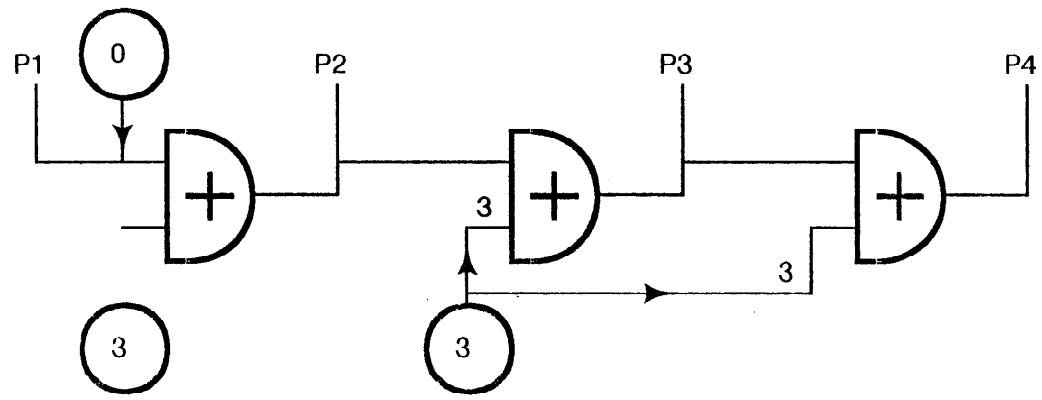


FIGURE 4-11. A Usefully Modified Four-Point Spacing Network.

```
;|Awakening <A34:ADDER-101> because its B got the value 3.
;|<A34:ADDER-101> computed 9 for its part C from pins A, B.
;|Awakening <A34:ADDER-101> because its C got the value 9.
DONE
```

Now `(the b a34)` has been connected to a more legitimate source of 3. This network makes some sense: it spaces `p2`, `p3`, and `p4` equally, and allows a different spacing to be specified between `p1` and `p2`. Thus the disconnection facility can be used to make useful modifications to an existing network.

Disconnecting `(the b a12)` will sever the connection with the default value 3. This will cause retraction of many computed values.

```
(disconnect (the b a12))
;|Disconnecting (THE B A12) from CELL-105.
;|Removing 3 from (THE B A12).
;|Removing 3 from (THE C A12) because of NIL.
;|Removing 6 from (THE C A23) because (THE A A23)==(THE C A12).
;|Removing 9 from (THE C A34) because (THE A A34)==(THE C A23).
;|Awakening <A34:ADDER-101> because its C lost its value.
...
DONE
```

This situation is shown in Figure 4-11.

If now `(the b a12)` and `(the b a23)` are connected, the spacing network will have been completely restored, and `p4` can be computed from `p1` = 0 and the given spacing 3.

```
(== (the b a12) (the b a23))
```

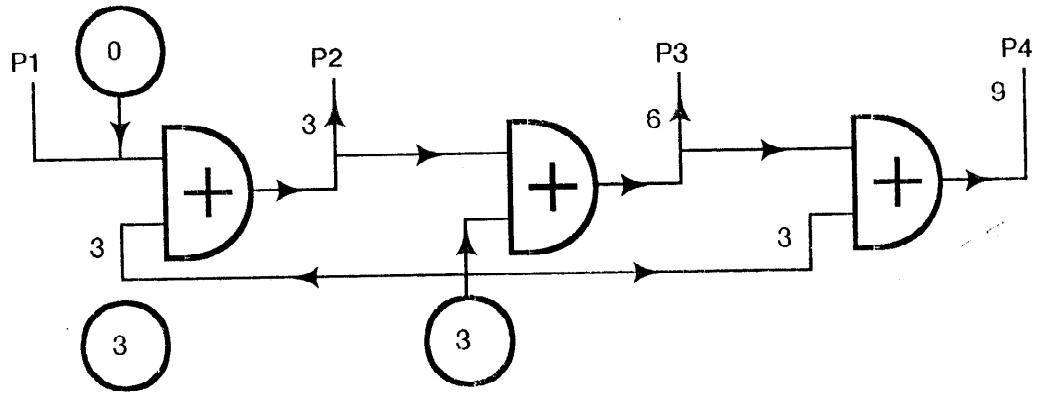


FIGURE 4-12. A Usefully Modified Four-Point Spacing Network.

```
...
DONE
(what p4)
;The value 9 in CELL-92 was computed in this way:
;   P4 ← (+ (+ (+ P1 3) 3) 3)
;   P1 ← 0
OKAY?
```

This final happy circumstance appears in Figure 4-12.

4.2. Implementation of Retraction Mechanisms

The primary visible difference in the language, aside from the addition of extra capabilities such as `dissolve`, is the new distinction between `default` and `constant` values. Internally they are the same, except that each cell is tagged as to which type it is, for those routines which care to check for the distinction.

The changes for `constant`, `default`, `constantp`, and `globalp` appear in Table 4-1. The function formerly called `constant` has been renamed `initialized-cell`, and given an extra parameter `reason`, which will be the symbol `constant` or `default`. The functions of those names simply call `initialized-cell`. When the cell is generated by `gen-cell`, the name given to the cell is not " ? " as before, but rather the symbol `constant` or `default`; this was originally intended to make the distinction visible when such a cell is printed, but turned out to have an important application in the retraction process. The predicates `constantp` and `globalp`, which operate by checking the name of the cell, require changes for the new naming convention. Finally, recall that the rule component of a node was formerly only used if the supplier

```
(defun constant (value)
  (initialized-cell value 'constant))

(defun default (value)
  (initialized-cell value 'default))

(defun initialized-cell (value reason)
  (let ((cell (gen-cell reason)))
    (setf (node-contents cell) value)
    (setf (node-boundp cell) t)
    (setf (node-supplier cell) cell)
    (setf (node-rule cell) reason)
    cell))

(defun constantp (cell)
  (require-cell cell)
  (and (null (cell-owner cell)) (memq (cell-name cell) '(constant default))))

(defun globalp (cell)
  (require-cell cell)
  (and (null (cell-owner cell)) (not (memq (cell-name cell) '(constant default)))))
```

Compare this with Table 3-2 (page 76).

TABLE 4-1. Implementation of Constant and Default Cells.

for the node was a pin, because if the supplier was a constant then the "rule" for the value was self-evident. Now, when the supplier is a `constant` or `default` cell, which kind is recorded as the rule. (This information could still be derived from the cell name, but it seems cleaner to express it as the rule.)

The handling of contradictions must be changed to allow for retraction. Contradiction handling is now centralized in the function `process-contradiction`. The function `merge-values` is changed to call `process-contradiction` on discovering a conflict (after printing a message). However, this is no longer considered to be a fatal error that brings the system to a grinding halt. It is assumed that `process-contradiction` may have fixed the problem (but only maybe!—if the conflicting values depended on redundant premises, then removing one premise may only have caused the retraction and recomputation of the conflicting value from other premises). Thus, when `process-contradiction` returns, the merge must be retried.

If a merge discovers a contradiction, then the resolution of that contradiction will require changing the values of the cells involved. Hence it is important in `==` not to decide which of the cells is bound until after `merge-values` has been called. (This is a subtle interaction which I missed at first!) Thus `==` has been rearranged to call `merge-values` first thing, and save the result in `newval`. Also, the decision as to which repository to use (and thus which cell of the merged node should become the supplier) has become more complicated: as a heuristic, if one supplier is a `constant` (as determined by looking at the rule), then that is preferred to anything

```
(defun == (cell1 cell2)
  (require-cell cell1)
  (require-cell cell2)
  (or (eq (cell-repository cell1) (cell-repository cell2))
      (let ((newval (merge-values cell1 cell2)))
        (let ((r1 (cell-repository cell1))
              (r2 (cell-repository cell2))
              (cb1 (node-boundp cell1))
              (cb2 (node-boundp cell2)))
          (let ((r (cond ((eq (rep-rule r1) 'constant) r1)
                         ((eq (rep-rule r2) 'constant) r2)
                         ((or (not cb2) (and cb1 (ancestor cell1 cell2))) r1)
                         (t r2)))
                (rcells (append (rep-cells r1) (rep-cells r2))))
            (setf (rep-contents r) newval)
            (let ((newcomers (if cb1 (if cb2 '() (rep-cells r2))
                                 (if cb2 (rep-cells r1) '()))))
              (setf (rep-cells r) rcells)
              (dolist (cell (rep-cells (if (eq r r1) r2 r1)))
                      (setf (cell-repository cell) r))
              (awaken-all newcomers)
              'done))))))

(defun merge-values (cell1 cell2)
  (require-cell cell1)
  (require-cell cell2)
  (let ((val1 (node-contents cell1))
        (val2 (node-contents cell2)))
    (cond ((not (node-boundp cell1)) val2)
          ((not (node-boundp cell2)) val1)
          ((equal val1 val2) val1)
          (t (ctrace "Contradiction when merging ~S and ~S." cell1 cell2)
             (process-contradiction (list cell1 cell2))
             (merge-values cell1 cell2)))))

(defun awaken-all (cells)
  (dolist (cell cells)
    (require-cell cell)
    (cond ((cell-owner cell)
           (ctrace "Awakening ~S because its ~S ~
                    ~:[lost its value~*~;got the value ~S~]."
                   (cell-owner cell)
                   (cell-name cell)
                   (node-boundp cell)
                   (node-contents cell))
           (awaken (cell-owner cell))))))
```

Compare this with Table 3-3 (page 77).

TABLE 4-2. Delaying Equating Decisions Until after the Merge.

else. This cannot cause circularities, because nothing is an ancestor of a `constant`. The reason for this heuristic will be discussed below.

```
(defmacro setc (cellname value)
  `(process-setc *me* ',cellname ,(symbolconc cellname "-CELL") ,value *rule*))

(defun process-setc (*me* name cell value rule)
  (require-constraint *me*)
  (require-cell cell)
  (let ((sources (get rule 'trigger-names)))
    (cond ((not (node-boundp cell))
           (ctrace "~S computed ~S for its part ~S~:[~2*~; from pin~P ~{~S~^, ~}~]."
                   *me* value name sources (length sources) sources)
           (setf (node-contents cell) value)
           (setf (node-boundp cell) t)
           (setf (node-supplier cell) cell)
           (setf (node-rule cell) rule)
           (awaken-all (node-cells cell)))
          ((not (equal (node-contents cell) value))
           (let ((triggers (forlist (pinname sources) (*the pinname *me*))))
             (ctrace "Contradiction in ~S~@[ among these parts: ~
                                ~:~S=~S~:^, ~~];~
                       ~%;|  it calculated ~S for ~S ~
                                from the others by rule ~S."
                     *me*
                     (forlist (cell (cons cell triggers))
                       (require-cell cell)
                       (list (cell-name cell) (node-contents cell)))
                     value
                     (cell-name cell)
                     rule)
             (process-contradiction (cons cell triggers))
             (do ((x triggers (cdr x)))
                 ((null x) (process-setc *me* name cell value rule))
               (or (node-boundp (car x)) (return)))))))))
```

Compare this with Table 3-8 (page 82).

TABLE 4-3. Handling Contradictions in `setc`.

The `ctrace` message printed by `awaken-all` is changed because owners may now be awakened not only because a pin has newly received a value, but because a pin has lost a value (in which case the awakening is a request to recompute it if possible).

```
(defmacro contradiction vars
  `(signal-contradiction (list ,@(forlist (v vars) (symbolconc v "-CELL"))) *me*))

(defun signal-contradiction (cells constraint)
  (require-constraint constraint)
  (ctrace "Contradiction in ~S~@[ among these parts: ~:{~S=~S~:^, ~}~]."
          constraint
          (forlist (cell cells)
            (require-cell cell)
            (list (cell-name cell) (node-contents cell))))
  (process-contradiction cells))

(defun process-contradiction (cells)
  (let ((premises (premises* cells)))
    (let ((losers (do ((p premises (cdr p))
                       (z '() (if (eq (node-rule (car p)) 'default)
                                  (cons (car p) z)
                                  z)))
                      ((null p) (or z premises)))))
      (cond ((null losers) (lose "Hard-core contradiction!"))
            ((null (cdr losers))
             (retract (car losers)))
            (t (retract (choose-culprit losers)))))))

(defun choose-culprit (losers)
  (format t "~%;;; These are the premises that seem to be at fault:~
             ~:{~%;~8X~S~@{ == ~S~}~:^,~}."
          (forlist (p losers)
            (cons p (mapcan #'(lambda (c)
                                (and (globalp c)
                                     (list (cell-name c))))
                            (node-cells p)))))
  (format t "~%;;; Choose one of these to retract and RETURN it.")
  (let ((culprit (break "Choose Culprit")))
    (do ((z losers (cdr z)))
        ((null z) (choose-culprit losers))
      (and (eq (cell-repository (car z)) (cell-repository culprit))
           (return (car z))))))
```

Compare this with Table 2-11 (page 57).

TABLE 4-4. Processing and Recovering from Contradictions.

The processing for the `setc` construct remains the same in the usual case (see Table 4-3). When a contradiction is detected, however, because the value computed by a constraint conflicts with a value already on the pin, then `process-contradiction` is called, giving it a list of the conflicting cells. When `process-contradiction` returns, then there is a question as to whether to install the value in the pin after all—the processing of the contradiction may have removed the support for that value. Hence `process-setc` checks all of the trigger pins for the rule, and only retries the `setc` operation if they all still have values.

(There are serious problems remaining, however. The assumption here is that no new values will be asserted within `process-contradiction`, but only old ones retracted. This requires the assumption that the user will not assert new equatings within the breakpoint provided by `signal-contradiction`, but only use probing functions like `what`. If the user were to not simply retract a value from a trigger pin but were also to provide a new value, then the test in `process-setc` would be incorrect; a value would be supported, but not the one in hand! A better thing to do would be to restart the rule which invoked `setc`; this involves a non-local escape. This issue will be addressed in the next chapter.)

The `contradiction` construct is implemented in the same way as before (Table 4-4). The function `signal-contradiction` does not signal a fatal error, however, but simply prints a message and then calls `process-contradiction`.[5] This function takes a list of conflicting cells, All of the places in the system which detect contradictions (`merge-values`, `process-setc`, and `contradiction`) handle them by calling `process-contradiction`.

A set of "losers" (constant cells deemed to be collectively at fault for the contradiction) is computed. The function `premises*` computes the set of joint premises for the list of cells. Then those premises which are `default` cells are extracted. If there are any `default` cells, then those are considered to be the losers; otherwise all the premises (which must then all be `constant` cells) are taken as the losers. If there are no losers at all (this shouldn't ever happen?), it is fatal. If there is just one loser, it is automatically retracted. Otherwise, `choose-culprit` is called to decide which one to retract.

(Suppose there are two losers, and one is a `default node` specified explicitly in a `==` request which the user just typed, causing the contradiction. Should the system in this case automatically retract the value just specified (thus making the network resistant to obvious inconsistent changes)? This would make it hard to vary parameters. Or should the system automatically retract the other loser, allowing the one just explicitly specified to hold? Consider the two cases where the cell to which the default was explicitly equated already had a value or did not. Exercise for the reader: determine what is "the right thing".)

In principle, `choose-culprit` could be an automatic routine using various heuristics to decide which loser to retract. This version defers the problem to the user (not necessarily the best thing to do). It prints the list of losers (for each one printing also any global names connected to it), and then calls `break` to enter a LISP breakpoint. The `return` function causes the specified value to be returned from the call to `break`, so this is bound to the variable `culprit` . This returned value is then tested to ensure that it is a valid culprit; if it is not, then the question is repeated. The user need not return an actual loser cell, but may return any cell of that node. This is for convenience, so that the user may refer to a value by an equivalent global name.

5. As with `setc`, there is an assumption that within `process-contradiction` there will be only retractions, not any newly computed values. Therefore it is not necessary to retry the contradiction test.

```
(defun retract (cell)
  (ctrace "Retracting the premise ~S." cell)
  (awaken-all (forget cell)))

(defun forget (cell &optional (source () sourcep)  (via () viap))
  (require-cell cell)
  (and sourcep (require-cell source))
  (and viap (require-cell via))
  (ctrace "Removing ~S from ~S~:[~3*~; because ~:[of ~;~S==~]~S~]."
          (node-contents cell)
          (cell-goodname cell)
          sourcep
          (and viap (not (eq via source)))
          (and viap (not (eq via source)) (cell-goodname via))
          (and sourcep (cell-goodname source)))
  (setf (node-boundp cell) ())
  (setf (node-contents cell) ())
  (setf (node-supplier cell) ())
  (setf (node-rule cell) ())
  (let ((fcells (append (rep-cells (cell-repository cell)) '())))
    (dolist (c (rep-cells (cell-repository cell)))
      (and (cell-owner c)
           (dolist (value (con-values (cell-owner c)))
             (require-cell value)
             (and (node-boundp value)
                  (eq value (node-supplier value))
                  (memq (cell-name c)
                        (get (node-rule value) 'trigger-names))
                  (setq fcells (nconc (forget value cell c) fcells))))))
    fcells))
```

TABLE 4-5. Retracting Values from the Network.

```
(defun premises (cell)
  (require-cell cell)
  (cond ((not (node-boundp cell)) '())
        (t (let ((s (node-supplier cell)))
             (if (null (cell-owner s))
                 (list s)
                 (premises* (forlist (name (get (node-rule s) 'trigger-names))
                                     (*the name (cell-owner s)))))))))

(defun premises* (cells)
  (do ((c cells  (cdr c))
       (p '() (unionq (premises (car c)) p)))
      ((null c) p)))
```

Compare this with Table 3-10 (page 84).

TABLE 4-6. A Rewriting of the `premises` Function.

The function `retract` in Table 4-5 takes care of removing the value from a cell and awakening the relevant constraints to request recomputation. The recursive function `forget` takes a cell

whose value should be removed and returns a list of cells whose owners should be awakened. (The optional arguments `source` and `via` are used only internally to produce better `ctrace` output, so that the chain of recursive forgetting is more easily followed.)

Once the `ctrace` output has been produced in `forget`, the value is removed from the repository and the boundp, supplier, and rule components reset. Then the variable `fcells` is used to accumulate a set of cells whose owners should be awakened. This is initially all the cells of the current node. In addition, if any cell of the current node has an owner, and any of the pins of that owner was computed by a rule using that cell as a trigger, then those pins must also be forgotten, and the set of cells returned by the recursive call to `forget` is added to the current `fcells` set. Finally the complete `fcells` list is returned. (The `nconc` function can be used instead of the set-union operation `unionq` because we know that any node can be forgotten at most once; if such a node is encountered again by another path, its cells will not be returned by `forget` . A LISP trick: the call to `append` in the initialization of `fcells` copies the list of cells so that `nconc` may be used later and not destroy the list used by the repository.)

In summary, `forget` removes the value from the given node and all nodes whose values recursively depend on it. All the cells of all the nodes which lost values are returned. The owners of these cells must be awakened; this process is known as "begging", because a node that forgets its value must beg connected devices to re-supply (and re-support) the value. This is necessary because the device might, after all, have provided a value as a subsidiary supplier, only to have the value discarded because it agreed with the value provided by the main supplier.

The function `premises*`, which is simply a part of `premises` which formerly was not a separate function, is given in Table 4-6. This illustrates a common situation in dealing with recursive data structures: one part of the recursion involves mapping over a list of sub-structures. Writing two mutually recursive functions allows two entry points, one taking a single structure, one taking a list of them. A similar transformation for `fast-premises` (which the reader will recall does the same thing as `premises`, but with a better worst case, by using graph-marking techniques) appears in Table 4-7.

Counterintuitively, it is actually easier to dissolve a node than to disconnect a single cell from a node. One might think that dissolution involves more work, and indeed it may; but disconnecting requires more special cases, because one cell is treated differently from the rest, and so requires more code.

When a node is dissolved (see the code for `dissolve` in Table 4-8), each cell must become a node unto itself, and so each must have its own repository. To avoid some work and to avoid wasting a repository, the existing repository is left attached to the supplier for the original node. If the node has no value, then an artificial supplier is chosen arbitrarily by using the cell received as an argument. For every cell other than the supplier, a repository is created and hooked up to the cell. If the cell is a constant (in which case the node must have a value, and that value must be the

```
(defun fast-premises (cell)
  (require-cell cell)
  (prog1 (fast-premises-mark cell) (fast-premises-unmark cell)))

(defun fast-premises* (cells)
  (prog1 (fast-premises-mark* cells) (fast-premises-unmark* cells)))

(defun fast-premises-mark (cell)
  (require-cell cell)
  (and (node-boundp cell)
       (let ((s (node-supplier cell)))
         (cond ((markp s) '())
               (t (mark-node s)
                  (if (null (cell-owner s))
                      (list s)
                      (fast-premises-mark*
                          (forlist (name (get (node-rule s) 'trigger-names))
                            (*the name (cell-owner s))))))))))

(defun fast-premises-mark* (cells)
  (do ((c cells (cdr c))
       (p '() (nconc (fast-premises-mark (car c)) p)))
      ((null c) p)))

(defun fast-premises-unmark (cell)
  (require-cell cell)
  (let ((s (node-supplier cell)))
    (cond ((markp s)
           (unmark-node s)
           (or (null (cell-owner s))
               (fast-premises-unmark
                   (forlist (trigger-name (get (node-rule s) 'trigger-names))
                     (*the trigger-name (cell-owner s)))))))))

(defun fast-premises-unmark* (cells)
  (dolist (cell cells) (fast-premises-unmark cell)))
```

Compare this with Table 3-11 (page 85).

TABLE 4-7. A Rewriting of the `fast-premises` Function.

constant's value, even though the constant is not the supplier—the assumption is that the network is consistent), then the new repository should bear the value. The constant cell becomes its own supplier, and the name of the cell (`constant` or `default`) is used as the rule name. (This is the situation alluded to earlier where the cell name is used other than for printing.) If the cell is not a constant, then it becomes a valueless node, as it has become disconnected from its supplier (if any).

Once each cell has gotten its own repository, then the original repository remains with the supplier cell alone; its cells component is updated to reflect this fact.

Finally, if the node had had a value, then disconnecting some cells has the effect of retraction of a value, and so the `forget` function must be applied. Every cell which is not bound and has an owner is given to `forget` to recursively remove values which depended on the connection; once

```
(defun dissolve (cell)
  (require-cell cell)
  (let ((supplier (if (node-boundp cell) (node-supplier cell) cell))
        (cells (node-cells cell)))
    (ctrace "Dissolving ~{~S~^, ~}." (forlist (c cells) (cell-goodname c)))
    (dolist (c cells)
      (or (eq c supplier)
          (let ((r (make-repository)))
            (cond ((and (node-boundp supplier) (constantp c))
                   (setf (rep-contents r) (node-contents supplier))
                   (setf (rep-boundp r) t)
                   (setf (rep-supplier r) c)
                   (setf (rep-rule r) (cell-name c))))
            (setf (cell-repository c) r)
            (push c (rep-cells r)))))
    (setf (node-cells supplier) (list supplier))
    (and (node-boundp supplier)
         (let ((queue '()))
           (dolist (c cells)
             (cond ((and (not (node-boundp c)) (cell-owner c))
                    (setf (node-contents c) (node-contents supplier))        ;kludge
                    (setq queue (nconc (forget c) queue)))))
           (awaken-all queue))))
  'done)
```

TABLE 4-8. Dissolving a Node—Carefully!.

this has been done, all the appropriate awakenings are performed. (The line marked "kludge" puts the old value back into the repository solely so that `forget` can print its trace message correctly before removing the value again.)

Observe that `dissolve` works correctly in the limiting case of a single-cell node. The first loop does nothing; the setting of the node-cells changes nothing; and the second loop only executes when the supplier has a value, but its body only works for cells with no value. Hence the node remains effectively unchanged.

Disconnecting a cell requires some special cases (Table 4-9). The disconnected cell must acquire a new repository, while the old one remains with all the other cells of the node. The cell is deleted from the cells list of the old repository, and hooked up to the new one. The contents, boundp, supplier, and rule components are copied from the old repository to the new one. There follow several cases.

(a) If the node had had a value and was the supplier for the node, then there is great upheaval. First there is a search for other cells of the node which might immediately become a new supplier for the node; such cells must be constants. However, `constant` cells are preferred to `default` cells, and so there are two identical loops, one looking for `constant` cells, and the other for `default` cells if the first one fails. In either case, the found cell is installed as the new supplier. If no such new supplier can be found, then a third loop applies `forget` to all

```
(defun disconnect (cell)
  (require-cell cell)
  (let ((oldr (cell-repository cell))
        (newr (make-repository)))
    (setf (rep-cells oldr) (delq cell (rep-cells oldr)))
    (ctrace "Disconnecting ~S from ~{~S~^, ~}."
            (cell-goodname cell)
            (forlist (c (rep-cells oldr)) (cell-goodname c)))
    (setf (cell-repository cell) newr)
    (push cell (rep-cells newr))
    (setf (rep-contents newr) (rep-contents oldr))
    (setf (rep-boundp newr) (rep-boundp oldr))
    (setf (rep-supplier newr) (rep-supplier oldr))
    (setf (rep-rule newr) (rep-rule oldr))
    (cond ((and (rep-boundp oldr) (eq cell (rep-supplier oldr)))
           (do ((c (rep-cells oldr) (cdr c)))
               ((null c)
                (do ((cc (rep-cells oldr) (cdr cc)))
                    ((null cc)
                     (do ((ccc (rep-cells oldr) (cdr ccc))
                          (z '() (nconc (forget (car ccc)) z)))
                         ((null ccc) (awaken-all z))))
                  (cond ((and (constantp (car cc))
                              (eq (cell-name (car cc)) 'default))
                         (ctrace "~S becomes the new supplier for the node."
                                 (cell-id (car cc)))
                         (setf (rep-supplier oldr) (car cc))
                         (return)))))
             (cond ((and (constantp (car c))
                         (eq (cell-name (car c)) 'constant))
                    (ctrace "~S becomes the new supplier for the node."
                            (cell-id (car c)))
                    (setf (rep-supplier oldr) (car c))
                    (return)))))
          ((constantp cell)
           (setf (rep-supplier newr) cell)
           (setf (rep-rule newr) (cell-name cell)))
          (t (awaken-all (forget cell)))))
  'done)
```

TABLE 4-9. Disconnecting a Cell from a Node.

of the cells of the node, and then gives the collective nodes to `awaken-all` in an attempt to recompute a value.

(b) If the cell had been a constant but not the supplier for the node, then its value must have been the same as that of the node. It retains its value, but becomes its own supplier again, and the rule is copied from the cell name (`constant` or `default`), just as for `dissolve` in Table 4-8.

(c) Otherwise the cell has been disconnected from its supplier, and its value must be forgotten. The usual `forget-awaken-all` sequence is applied to it.

4.3. Summary of the Retraction Mechanisms

The retraction capability exhibits the first traces of an automatic deduction facility. When a contradiction is observed, the system automatically traces the problem to its origin, and then makes a decision (sometimes automatically, but often by asking the user) as to how to solve the problem. Once the decision is made, the system will remove one premise from the network and "anti-propagate" the value—that is, propagate the removal by removing values which were computed from it, and then try to find ways to recompute removed values. Thus the system tries in whatever way it can to compute values for as many nodes as possible,

The distinction introduced between `constant` and `default` cells allows the user to express a level of confidence in the value. For example, constants used within a program that are part of the intended algorithmic structure can be expressed as `constant` cells, while input data can be expressed as `default` cells. The distinction is used to decide what premises to retract in case of contradiction. As a secondary benefit, the distinction can also be used to choose heuristically the "best" supplier for a node. In the general case one might want to have many types of constant cells, with some partial order among the types to determine which ones are losers; even more generally, a user procedure could be allowed to step in and choose among the premises; but this gets very complicated.

The ability to disconnect portions of the computation network also is a kind of retraction capability. When a computation goes awry, the fault may be with the input data, but it may also be that the program was misconstructed. However, we have not yet provided for automatic retraction of network connections! Such a facility might be useful, however—certainly the network is suspect if a contradiction cannot be traced to any premise! If `constant` cells are considered to be part of the algorithmic structure expressed by the network, then perhaps suspicion of network connections should be on a par with suspicion of `constant` values.

The `default` mechanism is not quite like that in [Doyle 1978a] and [McAllester 1978]; it gives preference to the value for retraction, but makes no attempt to re-assert the value if the situation causing the contradiction is altered. This ability is treated in the next chapter.

Once you were two.
Dear birthday friend.
In spite of purple weather.

But now you are three
And near the end
As we grewsome together.

How fourthful thou.
Forsooth for you.
For soon you will be more!

But—'fore
One can be three be two:
Before be five before!

—Walt Kelly (1951)

Chapter Five

Assumptions

THE DEFAULT VALUE MECHANISM presented in Chapter Four allows one to say, "in the absence of any other information, assume that a certain value is thus-and-so—but feel free to ignore the value if necessary." In our constraint language, however, where computations can be undone *and redone*, it is useful to draw a distinction between a default value which, once retracted, does not re-appear, and one which has a certain persistence. We will call the latter kind an *assumption.*

There are applications for constraints where it is vital that a node always have *some* value. For example, if a node represents the x-position of some graphical object being displayed on a screen, then if the object is to appear it must have *some* x-position. An assumption might be that the x-position is zero unless otherwise constrained.

Another use of assumptions is in case analysis. If it is known (or assumed!) that a node must take on one of a specific set of values, then one element of that set can be arbitrarily assumed to be the value of the node; another can always be chosen if this leads to a contradiction. Such assumptions lead to conclusions which are *permissible*, rather than *required.* If, however, one goes a step further and arranges to assume all of the values, one by one, then any conclusions which come out the same for all the choices must be the case independent of the choice, leading to the deduction that such conclusions are required *independent* of the choice of value for the node. (Here we shall not make use of this extra step, but will make use of its contrapositive form: if every assumption of a set leads to a contradiction, then the choice from the set is not itself at fault for the contradiction, but rather the sets of other premises for the respecitive contradictions, taken collectively. This will lead to the resolution principle.)

5.1. Definition of Assumption Constructs

We will introduce two new constructs to the language: `assume` and `oneof`. Each one will represent a cell in the same manner that the `constant` and `default` constructs do. However, `assume` and `oneof` cells will have an associated mechanism for persistently giving the cell a value.

More precisely, `(assume` *n*`)` should generate a cell which has the value *n* provided that the cell can take on that value consistently with the rest of the network. If having the value *n* would conflict with `constant` or `default` values in the network, then the value *n* is gracefully withdrawn.[1] If it would conflict with other assumptions, then one assumption is chosen arbitrarily and withdrawn.

Similarly, `(oneof` *list*`)` takes a LISP list of values and generates a cell which takes on one of the values in *list*. If taking on one value leads to a contradiction, it is withdrawn as for `assume` and a new value is tried. For example, `(oneof '(0 1 2))` generates a cell that tries to take on one of the values 0, 1, or 2. It might appear that

```
(== x (oneof '(0 3 5 6 9)))
```

is exactly the same as (and therefore simply shorthand for)

```
(== x (assume 0))
(== x (assume 3))
(== x (assume 5))
(== x (assume 6))
(== x (assume 9))
```

but this is not so. The latter says that `x` tries to take on one of the values 0, 3, 5, 6, or 9, other things being equal. If, however, some external constraint on `x` requires `x` to be 4, then all these assumptions quietly bow out. On the other hand, `oneof` imposes the constraint that `x` *must* take on some one of these values. If some external constraint on `x` requires `x` to be 4 when the `oneof` construct has been used as above, then a contradiction occurs.

1. It is thus arbitrarily assumed that assumptions are less important than defaults or constants. One might want to have something like a default which was less important than an assumption. Indeed, the question of persistence for a cell and the question of which is chosen for retraction in case of conflict are orthogonal.

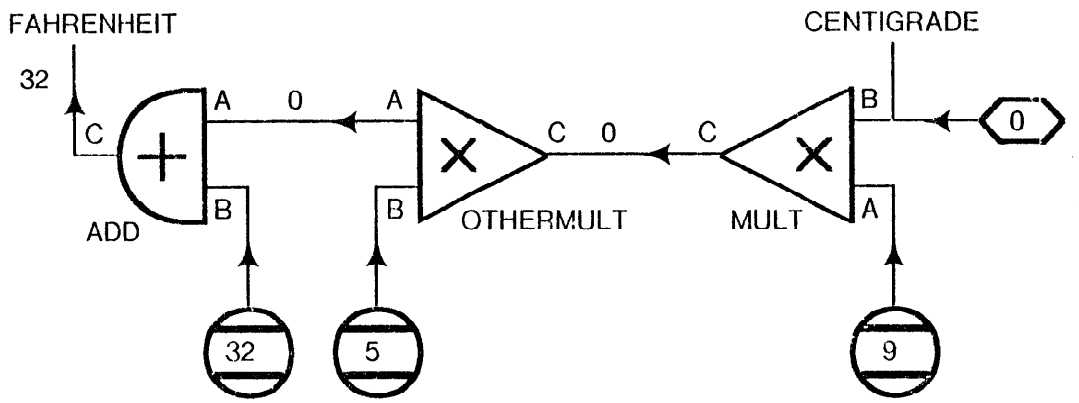


FIGURE 5-1. A Temperature Conversion Network with an Assumption.

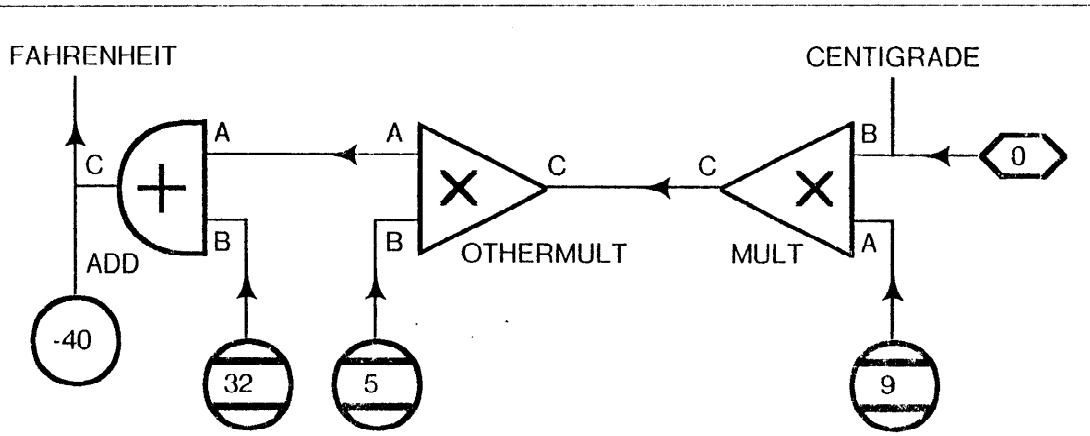


FIGURE 5-2. A Temperature Conversion Network, after Retracting an Assumption.

5.2. Implementation Problems

There are some difficulties with implementing these constructs. The primary problem is that they cannot operate using purely local information. This will be solved by recording extra information about the structure of the network so that assumption cells will have the information they need immediately to hand.

5.2.1. Nogood Sets Can Be Used to Locally Record Contradictions

Consider first the `assume` mechanism. Suppose, in a temperature conversion network, that `centigrade` is assumed to be zero:

```
(== centigrade (assume 0))
```

This causes the computation (on the basis of this assumption) of the value 32 for `fahrenheit` (see Figure 5-1; the assumption is indicated by a hexagonal shape). Suppose then that `fahrenheit` is equated to the default value —40. This of course causes immediate detection of a contradiction. The contradiction mechanism, tracing the premises of the contradictory values, finds that the premises are three `constant` cells, a `default` cell, and an `assume` cell. The last is chosen as the culprit and retracted. At this instant, just after the retraction of the assumption and the forgetting of the consequences, the situation is as shown in Figure 5-2. Once values have been forgotten, then every the owner (if any) of every retracted cell is awakened, to request it to compute a value if it can.

Here, then, is the problem. If the assumption cell is awakened in the "obvious" way, it will gladly supply a value for its cell. (It cannot tell at this point that this value is contradictory. All it can tell locally is that its cell has no value, and it has been asked to supply a value.) From this value new deductions may proceed. Indeed, before you know it the value 32 might be re-deduced for `fahrenheit`! This would trigger a new contradiction, and the result is that the system might oscillate, forever thrashing. Even if deductions from `fahrenheit` made any headway (say through the first multiplication device), deductions from the assumption might continue to beat against them at intermediate points.

One approach to solving this problem would be to assign priorities to propagation possibilities: values not depending on assumptions should be propagated in preference to values not depending on assumptions. Of course, this in effect implies carrying around information with each value describing its origin. This amounts to carrying around extra non-local information about distant parts of the network. Moreover, so far it has not been necessary to place any restrictions on the order in which propagation step are carried out by the system; indeed, we wish to preserve as much as possible the property that propagations may be performed in parallel.

If we are committed to recording some kind of non-local information, we might as well do it right, in a straightforward way. We will introduce the mechanism of *nogood sets* [Stallman 1977]. A nogood set records a set of premises which have been found to be inconsistent. When a contradiction occurs at least one of whose premises is an assumption, then a list of the premises is made up. This list is recorded in *each* premise's node. When an assumption cell considers trying to assert an assumed value for its node, it can first check all of the nogood sets associated with that node. If the assumed value would eventually cause the reoccurrence of a contradiction which has already been noted once and recorded in the form of a nogood set, then the assumption cell can avoid asserting the value.

Nogood sets record value information about the network, information gained at some computational cost, concerning sets of incompatible values. They serve as a cache, so that blind alleys need not be re-explored over and over again. Instead, assumption cells can perhaps determine

locally and immediately that its value will eventually cause a contradiction in the current situation. Nogood sets therefore provide a semi-predicate for the safety of the assumed value: the absence of a relevant nogood set does not guarantee that trying a value will succeed, but the presence of one can immediately guarantee that it must fail.

The formation of nogood sets of course constitutes a kind of algebra on the network. Each set summarizes some computation tree in terms of a set of values for its leaves known to be incompatible. Indeed, we might want to think of a nogood set as another kind of constraint: a redundant constraint that a certain set of nodes may not all simultaneously take on certain associated values, and that assumption cells know about specially. For efficiency (?), however, we will not actually implement them as constraint devices. (Another reason is that the language does not have sets as data objects.)

5.2.2. Resolution Can Derive New Nogood Sets from Old Ones

We turn now to the `oneof` construct. Suppose that the node to which a `oneof` cell is connected has no value, and the `oneof` cell is asked to supply a value. It can examine its set of possibilities, possibly filter out some of them by consulting the nogood sets recorded in the node, and then arbitrarily choose one of the remainder to assume.

Further suppose, however, that the recorded nogood sets rule out *all* of the possible choices; that is, for each choice there is a nogood set which rules out that choice. What then can be done? Let us refer to the nogood set that rules out a choice as a "killer" of that choice. Now a nogood set is a mapping of nodes to particular values, and asserts that not all the nodes may take on the associated values, because that has been previously determined to be a contradictory state. For a nogood set to be a killer for a choice for a node, it must be the case that every other node in the killer must currently bear its associated value. By an abuse of terminology let us call all these other nodes the premises of the killer (they are the grounds for assuming that the choice cannot hold). Since the `oneof` construction indicates that it is a contradiction not to be able to choose any of the values, then all the premises of all the killers must be collectively responsible for this contradiction. It follows that these collective premises themselves constitute a nogood set, which can be duly recorded. The result is that from several nogood sets sharing a common node, each forbidding one value for that node, a new nogood set can be derived not containing that node.

As an example, consider the network of Figure 5-3. The node named `confusion` has attached to it three little sub-networks and a `oneof` choice. One network states that whatever `confusion` is,

$$\texttt{confusion} + \texttt{confusion} = \texttt{confusion} \times \texttt{confusion}$$

must hold. This is a quadratic equation with roots 0 and 2; however, this subnetwork cannot compute a value for `confusion` by local propagation. The second subnetwork states that

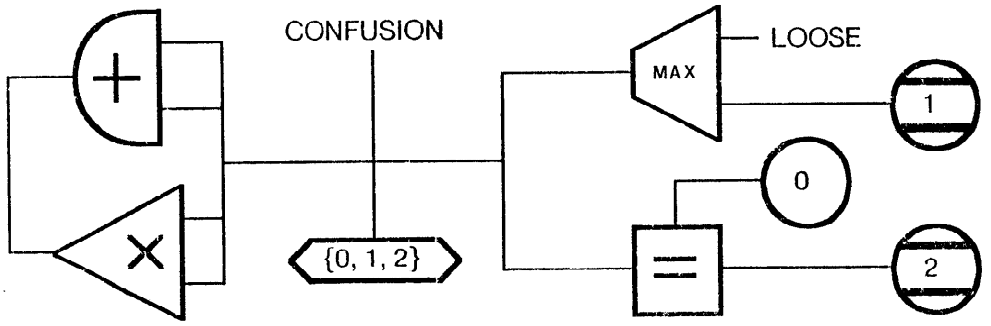


FIGURE 5-3. A `oneof` Cell for which No Alternative Works.

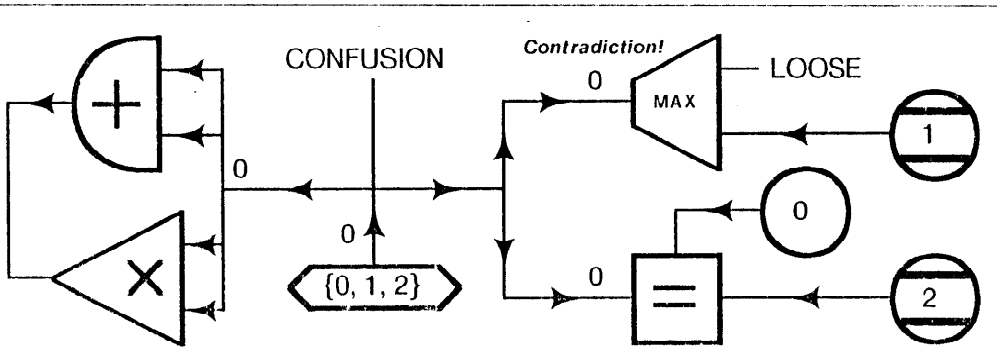


FIGURE 5-4. Assuming Zero Does Not Work.

`confusion` is the maximum of 1 and something else. The third says that `confusion` cannot be 2. Neither of these can compute a value for `confusion` by local propagation, either.

The `oneof` cell (indicated by a hexagonal shape with a set inside it) will assume some value of its set. Suppose that it assumes 0 is the value. Then a contradiction will occur in the `maxer` device, for 0 cannot be the maximum of 1 and anything else (Figure 5-4). The set of premises causing this contradiction is the assumption 0 and the constant 1. Thus a nogood set is created:

{(assumption-cell, 0), (constant-1, 1)}

(Here we notate a nogood set as a mathematical set of ordered pairs (cell, value).) The assumed value 0 is retracted, and all consequent deductions forgotten. This leaves us back where we started, except for the newly created nogood set.

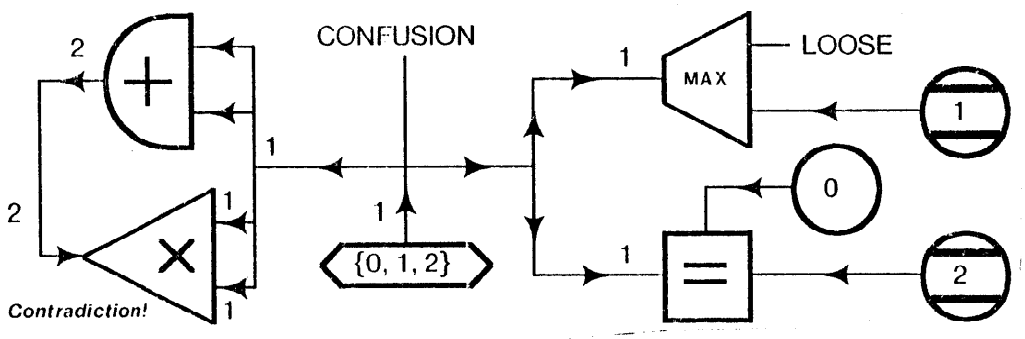


FIGURE 5-5. Assuming One Does Not Work.

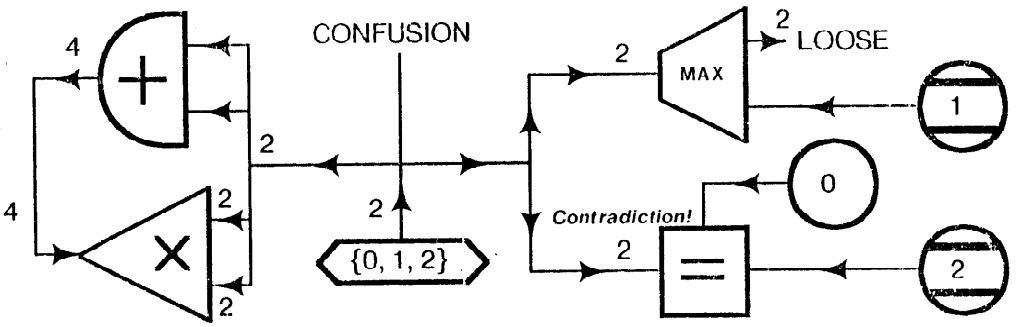


FIGURE 5-6. Assuming Two Does Not Work.

The `oneof` cell, true to its nature, would still like to assume some value. Consulting the available nogood sets, it discovers that 0 is currently forbidden. Suppose that it chooses 1. Then a contradiction will occur somewhere in the little quadratic equation, for 1 is not a root of the equation (Figure 5-5). The set of premises causing this contradiction is the assumption 1 (the quadratic equation contains no constants!). Thus another nogood set is created:

{(assumption-cell, 1)}

In other words, 1 simply can never work in the current network, even if constants or defaults are retracted. The assumed value 1 is retracted, and all consequent deductions forgotten. This leaves us once more back where we started, except for another nogood set.

The `oneof` cell *still* tries to assume some value. The existing nogood sets now rule out both 0 and 1, and so 2 is chosen. This causes a contradiction in the equality device (Figure 5-6). (It also happens to compute the value 2 for the variable `loose`, which hangs loose from the `maxer` device.) The set of premises causing this contradiction is the assumption 2, the constant 2, and the

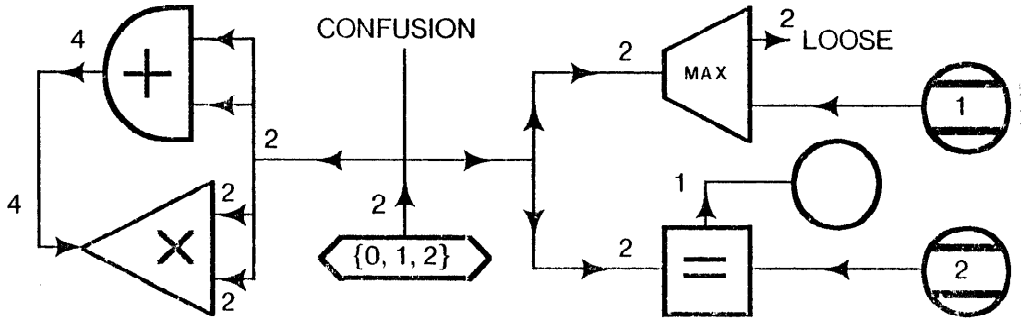


FIGURE 5-7. Causing a Contradiction and Retraction Eventually Works.

default value 0. Thus another nogood set is created:

{(assumption-cell, 2), (constant-2, 2), (default-0, 0)}

The assumed value 2 is retracted, and all consequent deductions forgotten. This leaves us yet again back where we started, except for yet another nogood set.

Once again the `oneof` cell tries to assume a value. Now it discovers that every possibility is ruled out. The constant 1 prevents the choice of 0; the choice of 1 is flatly forbidden; and the constant 2 and the default 0 rule out the choice of 2. Since one of the three choices must hold, this constitutes a contradiction. The three nogood sets are merged, eliminating the assumption-cell entries, to form a fourth nogood set:

{(constant-1, 1), (constant-2, 2), (default-0, 0)}

This constitutes a resolution step on the nogood sets. Now the contradiction mechanism of Chapter Four goes to work, and finding that there is exactly one `default` cell involved, automatically retracts that value.

One last time the `oneof` cell contemplates its situation. There are three nogood sets to consider. The value 0 is still ruled out, because the `constant` cell 1 still has its value. The value 1 is still ruled out. The value 2, however, is *not* now ruled out, because one cell of that nogood set (the `default` cell) now has no value. Hence the `oneof` cell is again free to choose 2 for its value. This eventually propagates throughout the network, and computes the value 1 for the `default` cell which was formerly 0 (Figure 5-7).

5.3. Implementation of Assumption Mechanisms

To implement the assumption mechanisms we need a way to represent the "persistence" of an assumed value, and also a data structure for representing nogood sets. We will treat an assumption as a funny kind of constraint, one which (sometimes) computes a value without requiring any inputs. The constraint needs to know what value (for `assume`) or values (for `oneof`) to choose from. To this end a new component *info* is added to every constraint. Not every constraint will use it (indeed most will not); it is a catch-all component for sticking extra things into. This component will find yet other uses in later chapters.

A nogood set will be represented as a header plus a sorted list of pairs, each pair being a cons of a repository (used to uniquely represent a node) and a value (an integer). Each repository will bear an identification (an *id* component similar to that for cells and constraints), and the pairs of a nogood set are sorted by alphabetical ordering on this repository identification. (The only reason for the id component in each repository is for sorting purposes, and the only reason for sorting is so that certain linear algorithms can be used on nogood sets.) The header will be simply the symbol `nogood`. This is present purely so that a nogood set can be altered by side-effect; if every place that knows about the nogood set points only to the header, then alterations of the nogood set will be visible to all. Thus a nogood set might look like this:

```
(NOGOOD (<REP-12> . 5) (<REP-15> . -3) (<REP-23> . 6))
```

This is a nogood set with three pairs.

Every repository mentioned in a nogood set needs to know about that nogood set. Hence another new component, *nogoods*, is added to each repository. This could be simply a list of all nogood sets mentioning that repository, but to speed up searching it will be divided into "buckets" according to the value associated with that repository in the nogood. Thus, for a given repository,all the nogoods associating value 0 with that repository will be in bucket 0; for the value 1, bucket 1; for the value -43, the bucket -43, etc. Thus, to check whether a certain value n is assumable for a repository, only nogood sets in bucket n need be checked. For fastest access to a bucket, the buckets could be kept in a hash array. We will not be that complicated here; instead, the nogoods component will simply be an a-list, associating buckets with values. However, the a-list pairs will be kept sorted by values, again for speed. In all, a repository's nogoods component might look like this:

```
((-3 (NOGOOD (<REP-12> . 5) (<REP-15> . -3) (<REP-23> . 6))
     (NOGOOD (<REP-15> . -3) (<REP-43> . -20)))
 (0 (NOGOOD (<REP-11> . -4) (<REP-15> . 0)))
 (7 (NOGOOD (<REP-14> . 2) (<REP-15> . 7) (<REP-23> . -7) (<REP-43> . 27))
    (NOGOOD (<REP-15> . 7) (<REP-24> . 0) (<REP-43> . 0))
    (NOGOOD (<REP-6> . 3) (<REP-14> . -7) (<REP-15> . 7) (<REP-43> . 27)))
 (9 (NOGOOD (<REP-15> . 9))))
```

```
(deftype constraint (con-id con-name con-ctype con-values con-info)
  (format stream "<~@[~S:~]~S>" (con-name constraint) (con-id constraint)))

(deftype repository ((rep-contents ()) (rep-boundp ()) (rep-cells ())
                     (rep-supplier ()) (rep-rule ()) (rep-mark ())
                     rep-id (rep-nogoods '()))
  (format stream "<Repository~:[~*~;: ~S~]~@[ for ~{~S~^,~/]~^]>"
          (rep-boundp repository)
          (rep-contents repository)
          (cell-ids repository)))

(defmacro node-contents (cell) `(rep-contents (cell-repository ,cell)))
(defmacro node-boundp (cell) `(rep-boundp (cell-repository ,cell)))
(defmacro node-cells (cell) `(rep-cells (cell-repository ,cell)))
(defmacro node-supplier (cell) `(rep-supplier (cell-repository ,cell)))
(defmacro node-rule (cell) `(rep-rule (cell-repository ,cell)))
(defmacro node-mark (cell) `(rep-mark (cell-repository ,cell)))
(defmacro node-nogoods (cell) `(rep-nogoods (cell-repository ,cell)))

(defun gen-repository ()
  (let ((r (make-repository))
        (n (gen-name 'rep)))
    (setf (rep-id r) n)
    (set n r)
    r))

(defun node-lessp (x y)
  (require-cell x)
  (require-cell y)
  (alphalessp (rep-id (cell-repository x)) (rep-id (cell-repository y))))
```

Compare this with Table 3-1 (page 75) and Table 2-3 (page 49).

TABLE 5-1. Data Structure Modifications for Assumptions.

This nogoods component has four buckets, which are sorted according to the values —3, 0, 7, and 9. These buckets have 2, 1, 3, and 1 nogood sets, respectively. This is evidently the nogoods component of repository number 15. The buckets are not sorted (they could be sorted by a lexicographic order, but this did not seem to be worthwhile for the present purposes). Each entry of each bucket (i.e., each nogood set) is sorted by repository id.

Table 5-1 shows the necessary changes to the `constraint` and `repository` data structures. As usual, a macro `node-nogoods` is defined to access the nogoods given a representative cell of a node. The function `gen-repository` generates a repository and associates a unique LISP variable name with it, also in the usual manner. Everyplace that used to call `make-repository` (these places are in `gen-cell`, `disconnect`, and `dissolve`) are changed to call `gen-repository`. (The new definition of `gen-cell` is not shown here, as that is the only change to that function.) The predicate `node-lessp` orders two nodes according to the alphabetical order of the id's of their respective repositories.

```
(defprim assumption (pin))

(progn 'compile
       (defun assumption-rule (*me*)
         (let ((*rule* 'assumption-rule)
               (pin-cell (the pin *me*)))
           (or (node-boundp pin-cell)
               (let ((value (con-info *me*)))
                 (do-named outer-loop
                           ((x (cdr (assoc value (node-nogoods pin-cell))) (cdr x)))
                           ((null x) (setc pin value))
                   (do-named inner-loop
                             ((c (cdar x) (cdr c)))
                             ((null c) (return-from outer-loop))
                     (and (not (eq (caar c) (cell-repository pin-cell)))
                          (or (not (rep-boundp (caar c)))
                              (not (equal (rep-contents (caar c)) (cdar c))))
                          (return-from inner-loop)))))))))
       (push 'assumption-rule (ctype-rules assumption))
       (defprop assumption-rule () trigger-names)
       (defprop assumption-rule (pin) output-names)
       (defprop assumption-rule assumption tentative)
       '(assumption rule))

(defun assume (value)
  (let ((a (gen-constraint assumption ())))
    (setf (con-name a) (con-id a))
    (setf (con-info a) value)
    (awaken a)
    (the pin a)))
```

TABLE 5-2. Implementation of the `assume` Construct..

The implementation of the `assume` construct is shown in Table 5-2. A special kind of primitive constraint called an `assumption` is first defined. It has a single pin called `pin`, and no rules of the usual kind. The function `assumption-rule` implements a special rule for assumptions, which unlike other rules has no triggers. The function's argument is called `*me*`, and the first thing it does is to bind the variables `*rule*` and `pin-cell`; this is in accordance with convention so that the `setc` construct can be used within the rule (see Table 4-3 (page 124)).

If the pin is *not* bound, then the assumption rule considers asserting an assumed value. The relevant value is stored in the info component of the constraint (which is passed in as `*me*`). The assumption rule performs a set of two nested loops. The outer loop fetches the bucket associated with the value from the node's nogoods component, then iterates over the contents of the bucket (each element is a nogood set). If each nogood set passes a test (that it not currently forbid the value), then the pin is set to the value, using `setc`.

The inner loop implements the nogood test. All the repositories in the nogood set are checked. If any repository other than the one for the pin-cell is either unbound or had a different value from the one associated with it in the nogood set, then that nogood set does not forbid the value; hence

```
(defprim oneof (pin))

(progn 'compile
       (push 'oneof-rule (ctype-rules oneof))
       (defprop oneof-rule () trigger-names)
       (defprop oneof-rule (pin) output-names)
       (defprop oneof-rule oneof tentative)
       '(oneof rule))

(defun oneof (valuelist)
  (let ((a (gen-constraint oneof ())))
    (setf (con-name a) (con-id a))
    (setf (con-info a) valuelist)
    (awaken a)
    (the pin a)))
```

TABLE 5-3. Implementation of the `oneof` Construct..

the inner loop may be exited, and the next nogood set tested. If the inner loop checks all the pairs of a nogood set without exiting, however, then the nogood set must forbid the value, and so the outer loop is exited. In other words, if p is the repository for the pin of the assumption, and b is the bucket of nogoods for the value, then the value is forbidden if

$$\exists n \in b\,(\forall (r, v) \in n\,(r \neq p \;\wedge\; \text{rep-boundp}(r) \;\wedge\; \text{rep-contents}(r) = v))$$

The function `assume` generates an assumption constraint. It makes the name of the constraint be the same as its id, installs the assumed value in the info field, and then—very important!—awakens the constraint. (Since the rule has no triggers, it is *always* triggerable. If the assumption is not awakened now, it probably never will be, so it better be done now.) This will cause the assumed value to be asserted in the pin. Finally, the pin is returned.

The `push` construct adds the rule to the set of rules for contraint-type `assumption`. The first two `defprop` forms define the set of triggers (empty) and outputs (the pin). The third `defprop` form defines the rule to be *tentative*; that is, a value computed using that rule is very "weak", and subject to automatic retraction.[2] Also, nogood sets should always be recorded for tentative values. This property will be used in `process-contradiction`.

The implementation of `oneof` (Table 5-3) is similar to that for `assume`. There is a kind of constraint called a `oneof`, and the info component of the constraint holds the list of possibilities. There is a function `oneof` which is analogous to the function `assume`—indeed, they are almost identical.

2. Compare this with Brown's "weak rules". [Brown 1980]

```
(defun oneof-rule (*me*)
  (let ((*rule* 'oneof-rule)
        (pin-cell (the pin *me*)))
    (let ((values (con-info *me*)))
      (cond ((node-boundp pin-cell)
             (or (member (node-contents pin-cell) values)
                 (contradiction pin)))
            (t (do-named loop-over-possibilities
                         ((v values (cdr v))
                          (killers '()))
                         ((null v)
                          (ctrace "All of the values ~S for ~S are no good."
                                  values
                                  (cell-goodname pin-cell))
                          (let ((losers '()))
                            (dolist (killer killers)
                              (dolist (x (cdr killer))
                                (or (eq (car x) (cell-repository pin-cell))
                                    (let ((cell (if (rep-boundp (car x))
                                                    (rep-supplier (car x))
                                                    (car (rep-cells
                                                          (car x))))))) ;??
                                      (or (memq cell losers)
                                          (push cell losers))))))
                            (process-contradiction losers))
                          (oneof-rule *me*))
                   (do-named outer-loop
                             ((x (cdr (assoc (car v) (node-nogoods pin-cell)))
                                 (cdr x)))
                             ((null x)
                              (setc pin (car v))
                              (return-from loop-over-possibilities))
                     (do-named inner-loop
                               ((c (cdar x) (cdr c)))
                               ((null c)
                                (push (car x) killers)
                                (return-from outer-loop))
                       (and (not (eq (caar c) (cell-repository pin-cell)))
                            (or (not (rep-boundp (caar c)))
                                (not (equal (rep-contents (caar c)) (cdar c))))
                            (return-from inner-loop)))))))))))
```

TABLE 5-4. The Rule for `oneof`.

The difference between `assume` and `oneof` is expressed in the function `oneof-rule` (Table 5-4). If the pin has a value, then it must be in the permitted set of values, or else a contradiction is signalled. If the pin has no value, then a more complicated search is performed. There is a third loop nested outside the other two, which loops over the possible choices. For each choice the same test used by `assumption-rule` is performed, trying to find a nogood set that will forbid the value. If none is found, then the value is installed in the pin, and the loop over the possibilities is exited, as a valid choice has been found. If, however, for a given possibility a nogood set is found which does forbid that choice, then there is no hope for that value. The nogood set is a killer for the value, and is remembered by pushing it onto the list `killers`.

```
(defun process-contradiction (cells)
  (let ((premises (premises* cells)))
    (do ((x premises (cdr x)))
        ((null x)
         (let ((losers (do ((p premises (cdr p))
                            (z '() (if (eq (node-rule (car p)) 'default)
                                       (cons (car p) z)
                                       z)))
                           ((null p) (or z premises)))))
           (cond ((null losers) (lose "Hard-core contradiction!"))
                 ((null (cdr losers))
                  (retract (car losers)))
                 (t (retract (choose-culprit losers))))))
      (cond ((get (node-rule (car x)) 'tentative)
             (ctrace "Deeming ~S in ~S (computed by rule ~S) to be the culprit."
                     (node-contents (car x))
                     (cell-id (car x))
                     (node-rule (car x)))
             (form-nogood-set premises)
             (retract (car x))
             (return))))))
```

Compare this with Table 4-4 (page 125).

TABLE 5-5. Looking for Tentative Values for Use as Culprits.

If any valid choice is found, then it is installed as described above. If no valid choice is found, then a killer nogood set has been found for each choice. In this case `oneof-rule` announces (via `ctrace`) that all the possibilities have been ruled out. It then takes the union of all the repositories in all the killers, other than the repository for the pin itself, accumulating them in the list `losers` (actually, for each repository a representative cell is found; if the repository has a value, then its supplier is used, and otherwise one is chosen arbitrarily [3]).

The list `losers` is eventually a set of cells in contradiction produced by resolution of the set of killers. These cells are given to `process-contradiction`. When contradiction processing has ended, `oneof-rule` re-invokes itself to try choosing again.

When a contradiction occurs, the central handler `process-contradiction` is called. This function is changed (Table 5-5) to have three priority levels for culprits: just as `default` values are preferred to `constant` values, so values computed by a tentative rule are preferred to either. Thus there is an extra search loop, which first checks all the premises for a tentative value. If any is found, it is immediately deemed to be the culprit, and a nogood set is constructed and recorded for this contradiction. The culprit is then retracted in the usual manner.

3. The latter case should of course never occur, but coding it this way allows for general non-monotonic rules later which are triggered by the lack of a value in the same way that `assumption-rule` and `oneof-rule` are. In this case the "unbound value" might usefully appear in a nogood set.

```
(defun form-nogood-set (cells)
  (setq cells (sort (append cells '()) #'node-lessp))
  (ctrace "The set~:{~<~%;|~8X~:15; ~S=~S~>~:t,~/]~<~%;|~8X~:15; is no good.~>"
          (forlist (c cells) (list (cell-goodname c) (node-contents c))))
  (let ((nogood (cons 'nogood
                      (forlist (c cells)
                        (cons (cell-repository c) (node-contents c))))))
    (dolist (cell cells)
      (let ((slot (assoc (node-contents cell) (node-nogoods cell))))
        (cond (slot (or (member nogood (cdr slot)) (push nogood (cdr slot))))
              ((or (null (node-nogoods cell))
                   (< (node-contents cell) (caar (node-nogoods cell))))
               (push (list (node-contents cell) nogood) (node-nogoods cell)))
              (t (do ((ng (node-nogoods cell) (cdr ng)))
                     ((or (null (cdr ng))
                          (< (node-contents cell) (caar (cdr ng))))
                      (setf (cdr ng)
                            (cons (list (node-contents cell) nogood)
                                  (cdr ng))))))))))))
```

TABLE 5-6. Constructing and Recording a Nogood Set.

It is essential that a nogood set be recorded if a tentative rule is involved, because the rule will depend on the existence of that set not to keep making the same poor choice over and over. It is not necessary to record a nogood set if only constant and default values are involved. It might be useful, of course; the ordinary propagation mechanism could check nogood sets in order to detect contradictions earlier. This might be particularly useful if the user is trying one default value after another while twiddling some parameter: the `==` mechanism (in `merge-values`, perhaps) could check nogood sets before attaching a new value in order to detect a bad value quickly. There is a trade-off between the space and time needed to record a nogood set and the time needed to check them, and the overhead of repeatedly rediscovering the same contradictory situation if premises are being varied rapidly. However, it is unclear whether this is worth it; it is a good subject for future statistical research.

The function `form-nogood-set` (Table 5-6) takes a list of nodes (i.e., representative cells), and constructs and records a nogood set for their current values. First the nodes are sorted according to the `node-lessp` predicate, to ensure that the nogood set will be properly sorted. (The call to `append` is intended to copy the list of nodes, because the `sort` primitive is destructive.) After a trace message is printed, the nogood a-list is constructed. Then for every node, the new nogood set is installed in that node. This involves using `assoc` to get the relevant bucket. If the necessary bucket exists, the nogood set is added to the bucket. Otherwise a new bucket must be created and inserted in the correct place to keep the list of buckets properly sorted. This involves some tedious special cases.

The trouble with adding an interesting new feature is always that it interacts with everything else. Nogood sets are no exception. What should happen to the nogood sets when two nodes are

```
(defun == (cell1 cell2)
  (require-cell cell1)
  (require-cell cell2)
  (or (eq (cell-repository cell1) (cell-repository cell2))
      (let ((newval (merge-values cell1 cell2)))
        (let ((r1 (cell-repository cell1))
              (r2 (cell-repository cell2))
              (cb1 (node-boundp cell1))
              (cb2 (node-boundp cell2)))
          (let ((r (cond ((eq (rep-rule r1) 'constant) r1)
                         ((eq (rep-rule r2) 'constant) r2)
                         ((or (not cb2) (and cb1 (ancestor cell1 cell2))) r1)
                         (t r2)))
                (rcells (append (rep-cells r1) (rep-cells r2))))
            (setf (rep-contents r) newval)
            (let ((newcomers (if cb1 (if cb2 '() (rep-cells r2))
                                 (if cb2 (rep-cells r1) '())))
                  (xr (if (eq r r1) r2 r1)))
              (setf (rep-cells r) rcells)
              (dolist (cell (rep-cells xr)) (setf (cell-repository cell) r))
              (let ((fcells (alter-nogoods-rep xr r)))
                (setf (rep-nogoods r)
                      (merge-nogood-sets (rep-nogoods r) (rep-nogoods xr)))
                (awaken-all fcells))
              (awaken-all newcomers)
              'done))))))
```

Compare this with Table 4-2 (page 123).

TABLE 5-7. Merging Nogood Sets When Equating Cells.

equated? According to our principle of order-independence, everything ought be be just as if the equating had happened first, followed by creation of the nogood sets. This is not simple.

The necessary changes to `==` are shown in Table 5-7. A variable `xr` has been introduced to stand for the repository which will be thrown away; thus `r` and `xr` are `r1` and `r2` or vice versa. Now if a value was no good for `xr` before, then it will certainly be no good for `r`, because they are to be the same. Hence all the nogoods for `xr` must be carried over to `r`. The function `alter-nogoods-rep` causes all the nogoods in `xr` to be modified to apply to `r`. Then the two collections of nogoods must be merged; in the process any duplicates are eliminated for searching efficiency later. (After all, there may have been two nogood sets that were identical except that one mentioned `r` and one mentioned `xr`.)

The function `alter-nogoods-rep` returns a list of cells whose owners should be awakened after everything else has been done. These cells are awakened by `==` after the nogood collections have been merged.

The function `alter-nogoods-rep` (Table 5-8) must handle lots of special cases. It iterates over the nogoods component of `xr`. For each bucket it iterates over the nogood sets in that bucket. For each nogood set it uses `assq` to find the pair mentioning `xr` (which must be present—if it is

```
(defun alter-nogoods-rep (xr r)
  (let ((fcells '()))
    (dolist (bucket (rep-nogoods xr))
      (dolist (nogood (cdr bucket))
        (let ((z (assq r (cdr nogood)))
              (xz (assq xr (cdr nogood))))
          (cond ((null xz)
                 (lose "Funny nogood set ~S for bucket ~S of repository ~S."
                       xr (car bucket) nogood))
                ((null z)
                 (setf (cdr nogood)
                       (add-nogood-pair r (cdr xz) (delassq xr (cdr nogood)))))
                ((equal (cdr z) (cdr xz))
                 (setf (cdr nogood) (delassq xr (cdr nogood))))
                (t (dolist (pair (cdr nogood))
                     (setq fcells (append (rep-cells (car pair)) fcells))
                     (let ((buck (assoc (cdr pair) (rep-nogoods (car pair)))))
                       (or buck (lose "Nonexistent bucket: ~S." pair))
                       (setf (cdr buck) (delq nogood (cdr buck)))
                       (or (cdr buck)
                           (setf (rep-nogoods (car pair))
                                 (delrassq '() (rep-nogoods (car pair))))))))))))
    fcells))

(defun add-nogood-pair (rep val nogoodlist)
  (require-repository rep)
  (cond ((null nogoodlist) (list (cons rep val)))
        ((node-lessp (car (rep-cells rep)) (car (rep-cells (caar nogoodlist))))
         (cons (cons rep val) nogoodlist))
        (t (cons (car nogoodlist) (add-nogood-pair rep val (cdr nogoodlist))))))
```

TABLE 5-8. Altering Nogood Sets for a New Repository.

not, an internal error has been detected), and possibly a pair mentioning `r`. If a pair mentioning `r` is not found, then the pair mentioning `xr` is deleted from the nogood set, and a pair mentioning `r` with the same value is added (it must be added in the correct place to keep the nogood set sorted). Now if there is a pair mentioning `r`, then there are two cases, depending on whether or not the mentions of `r` and `xr` associate the same value with each. If the values are the same, then the mention of `xr` should be deleted; the nogood relationship still holds, because once `r` and `xr` are merged, then `r` holding the value is the same as `xr` holding the value. If the values are different, then the nogood relationship can never hold (one of the two cases cannot hold), and so the entire nogood set might as well be eliminated; the nogood set must be deleted from every bucket which contains it. Every such bucket can of course be found from the repository-value pairs of the nogood set. As an extra but unnecessary space-saving twist, if deleting a nogood set from a bucket makes the bucket empty, then the bucket is removed from the list of buckets for that bucket's repository.

If a nogood set is eliminated, then all owners of cells in all the nodes whose repositories are mentioned in the nogood set must be awakened. This is because the nogood set might be the reason that some `assume` cell is not currently asserting its assumed value. Such `assume` cells must be

```
(defun merge-nogood-sets (s1 s2)
  (cond ((null s1) s2)
        ((null s2) s1)
        ((< (caar s1) (caar s2))
         (cons (car s1) (merge-nogood-sets (cdr s1) s2)))
        ((> (caar s1) (caar s2))
         (cons (car s2) (merge-nogood-sets s1 (cdr s2))))
        (t (cons (cons (caar s1) (merge-nogood-buckets (cdar s1) (cdar s2)))
                 (merge-nogood-sets (cdr s1) (cdr s2))))))

(defun merge-nogood-buckets (b1 b2)
  (cond ((null b1) b2)
        ((member (car b1) b2) (merge-nogood-buckets (cdr b1) b2))
        (t (cons (car b1) (merge-nogood-buckets (cdr b1) b2)))))
```

TABLE 5-9. Merging Two Collections of Nogood Sets.

awakened when the nogood set disappears. of them, because the network might be in a bad state until the caller has done some other clean-up first (this is the case in `==`).

The function `add-nogood-pair` simply inserts a new pair into a nogood list in the correct position for keeping it sorted.

The function `merge-nogood-sets` takes two lists of buckets of nogood sets, and merges them into a single collection. The merging of the top-level list takes linear times, because the buckets are in sorted order. However, the entries in a bucket are not sorted, and so merging two buckets with the same value can take quadratic time. On the other hand, each bucket entry (a nogood set) is kept sorted, and so is in a canonical form which can be compared by the LISP `equal` function (which is used by such primitives as `member` and `assoc`)—`equal` treats two objects of user-defined type (by `deftype`) as being equal iff they are `eq`.

When a value for a node is forgotten, then any nogood sets mentioning that value for that node might have formerly been suppressing an assumption and might now not so suppress an assumption. In the `forget` function (Table 5-10), the old value must be remembered internally before it is destroyed, and then used to fetch the relevant bucket of nogood sets. Any cells of nogoods in that bucket are added to `fcells` (the list of cells to be returned for later awakening), provided that they are not connected to the cell being forgotten and that they currently have no value. (A finer filter would first test the nogood set to see whether it actually could be suppressing a value: if too many nodes of the nogood set were unbound, or had values not matching the nogood's associated values, then the nogood would not be suppressing a value. On the other hand, it cannot hurt to awaken devices unnecessarily, except for the wasted effort involved. (On the third hand, to fail to awaken a device may be a disaster! [[]Sussman 1975]) The effort to filter the cells queued into `fcells` here should be weighed against the effort of unnecessary re-awakening here. This is purely an efficiency issue that will depend on details of a particular implementation.)

```
(defun forget (cell &optional (source () sourcep)  (via () viap))
  (require-cell cell)
  (and sourcep (require-cell source))
  (and viap (require-cell via))
  (ctrace "Removing ~S from ~S~:[~3*~; because ~:[of ~;~S==~]~S~]."
          (node-contents cell)
          (cell-goodname cell)
          sourcep
          (and viap (not (eq via source)))
          (and viap (not (eq via source)) (cell-goodname via))
          (and sourcep (cell-goodname source)))
  (let ((oldvalue (node-contents cell)))
    (setf (node-boundp cell) ())
    (setf (node-contents cell) ())
    (setf (node-supplier cell) ())
    (setf (node-rule cell) ())
    (let ((fcells (append (rep-cells (cell-repository cell)) '())))
      (dolist (c (rep-cells (cell-repository cell)))
        (and (cell-owner c)
             (dolist (value (con-values (cell-owner c)))
               (require-cell value)
               (and (node-boundp value)
                    (eq value (node-supplier value))
                    (memq (cell-name c)
                          (get (node-rule value) 'trigger-names))
                    (setq fcells (nconc (forget value cell c) fcells))))))
      (dolist (nogood (cdr (assoc oldvalue (node-nogoods cell))))
        (dolist (pair (cdr nogood))
          (and (not (eq (car pair) (cell-repository cell)))
               (not (rep-boundp (car pair)))
               (setq fcells (append (rep-cells (car pair)) fcells)))))
      fcells)))
```

Compare this with Table 4-5 (page 127).

TABLE 5-10. Forgotten Values May Re-enable Suppressed Assumptions.

If dealing with nogood sets is difficult when equating two nodes, it is nearly impossible when dissolving them. Dissolving nodes (or disconnecting single cells) disrupts network connections which had previously existed. Nogood sets implicitly contain information which is dependent on network structure, in a form which abstracts out the structure used to derive them—their very utility lies in this abstraction. When a node is dissolved, it may be very difficult to determine which nogood sets are still valid. The code here takes the easy way out—when a node is dissolved, all nodes reachable from the given node are visited, and their nogood collections destroyed. This is guaranteed to be safe; nogood sets merely redundantly encache information about the network. This information can be re-derived (at some cost, of course) for the new topology.

It would be possible to record in each node every nogood set that depended on the connections in that node; a modified `premises` function could gather together the nodes gone through, and `form-nogood-set` could use that list make the necessary records. However, this involves

```
(defun dissolve (cell)
  (require-cell cell)
  (let ((fcells (fast-expunge-nogoods cell)))
    ...
          (let ((r (gen-repository)))
            ...
            (awaken-all queue))))
    (awaken-all fcells))
  'done)

(defun disconnect (cell)
  (require-cell cell)
  (let ((fcells (fast-expunge-nogoods cell)))
    (let ((oldr (cell-repository cell))
          (newr (gen-repository)))
      ...
          (t (awaken-all (forget cell)))))
    (awaken-all fcells))
  'done)
```

Compare with Table 4-8 (page 130) and Table 4-9 (page 131).

TABLE 5-11. Disconnections Wreak Havoc with Nogood Sets.

some space and time overhead. If it is assumed that network structure changes slowly compared to changes of value, all that complexity may not be worthwhile.

The changes to the `dissolve` and `disconnect` functions are shown in Table 5-11. Each calls `fast-expunge-nogoods` before doing anything else, and each uses `gen-repository` instead of `make-repository`. After all the other work is done, then `awaken-all` is applied to the list of cells returned by `fast-expunge-nogoods`. Otherwise the code is the same as in Table 4-8 (page 130) and Table 4-9 (page 131), and so the bulk of the code is elided in Table 5-11.

Table 5-12 contains the code for `fast-expunge-nogoods`. It is a graph-marking algorithm that simply every node reachable from the given one, and destroys the nogood information in each node visited. The value of `fast-expunge-nogoods-mark` (which is returned by `fast-expunge-nogoods`) is a list of all the cells of all nodes visited which had any nogoods information. (This could be refined to return only cells with owners, or only cells owned by assumptions.) It marks nodes as they are visited, and as usual a post-pass resets the mark bits.

As with `alter-nogoods-rep` (Table 5-8), the cells are returned because the nogood information being destroyed might have formerly prevented some `assume` cell from asserting its value. Once the nogood information has been eliminated (and any changes to the network have been made by the caller of `fast-expunge-nogoods`), then such `assume` cells must be awakened, so that they may have another chance to assert their values.

There are a few trivial changes to various routines from the last chapter. The search for premises in the functions `premises` and `fast-premises` must treat assumptions as premises. Because `premises` and `fast-premises` perform the same work but the latter is faster in the

```
(defun fast-expunge-nogoods (cell)
  (require-cell cell)
  (prog1 (fast-expunge-nogoods-mark cell) (fast-expunge-nogoods-unmark cell)))

(defun fast-expunge-nogoods-mark (cell)
  (require-cell cell)
  (cond ((not (markp cell))
         (mark-node cell)
         (let ((fcells (and (not (null (node-nogoods cell)))
                            (append (node-cells cell) '()))))
           (setf (node-nogoods cell) '())
           (dolist (c (node-cells cell))
             (and (cell-owner c)
                  (dolist (v (con-values (cell-owner c)))
                    (setq fcells
                          (nconc (fast-expunge-nogoods-mark v) fcells)))))
           fcells))
        (t '())))

(defun fast-expunge-nogoods-unmark (cell)
  (require-cell cell)
  (cond ((markp cell)
         (unmark-node cell)
         (dolist (c (node-cells cell))
           (and (cell-owner c)
                (dolist (v (con-values (cell-owner c)))
                  (fast-expunge-nogoods-unmark v)))))))
```

TABLE 5-12. Rapid Destruction of Potentially Invalid Nogood Information.

```
(defun fast-premises-mark (cell)
  (require-cell cell)
  (and (node-boundp cell)
       (let ((s (node-supplier cell)))
         (cond ((markp s) '())
               (t (mark-node s)
                  (if (or (null (cell-owner s)) (get (node-rule s) 'tentative))
                      (list s)
                      (fast-premises-mark*
                          (forlist (name (get (node-rule s) 'trigger-names))
                            (*the name (cell-owner s))))))))))
```

Compare this with Table 4-7 (page 129).

TABLE 5-13. Assumptions Are Considered to be Premises.

worst case, from now on we will show the code only for `fast-premises`, the changes for which (occurring in `fast-premises-mark`) are shown in Table 5-13.

It would be nice if `what` knew how to print an `assume` or `oneof` cell in the same way it is typed. To this end a new convention is introduced whereby the occurrence of "`!`" in a treeform designates not a pin but instead the info component of a constraint. Thus the new treeform definitions in Table 5-14 specify how to print `assume` and `oneof` cells as desired.

```
(defprop adder ((c (+ a b)) (b (- c a)) (a (- c b))) treeforms)
(defprop multiplier ((c (* a b)) (b (// c a)) (a (// c b))) treeforms)
(defprop maxer ((c (max a b)) (b (arcmax c a)) (a (arcmax c b))) treeforms)
(defprop minner ((c (min a b)) (b (arcmin c a)) (a (arcmin c b))) treeforms)
(defprop equality ((p (= a b)) (b (arc= p a)) (a (arc= p b))) treeforms)
(defprop gate ((p (0-if-unequal a b)) (b (-> p a)) (a (-> p b))) treeforms)
(defprop assumption ((pin (assumption !))) treeforms)
(defprop oneof ((pin (oneof !))) treeforms)
```

Compare this with Table 3-15 (page 96).

TABLE 5-14. New `treeforms` Definitions.

```
(defun tree-form-chase (cell shallow top)
  (require-cell cell)
  (let ((s (node-supplier cell)))
    (cond ((and shallow (node-boundp cell)) (node-contents cell))
          ((and (not top) (not (singlenummarkp s)))
           ...
          ((cell-owner s)
           (cond ((and (eq s cell) (not top)) (cell-goodname s))
                 (t (let ((treeform
                           (cadr (assq (cell-name s)
                                       (get (ctype-name
                                              (con-ctype (cell-owner s)))
                                            'treeforms)))))
                      (cons (car treeform)
                            (forlist (n (cdr treeform))
                              (cond ((eq n '!) (con-info (cell-owner s)))
                                    ((and (node-boundp s)
                                          (not (memq n (get (node-rule s)
                                                            'trigger-names))))
                                     '?)
                                    (t (tree-form-chase (*the n (cell-owner s))
                                                        shallow
                                                        ()))))))))))
          ((globalp s) (cell-name s))
          (t (node-contents s)))))
```

Compare this with Table 3-17 (page 100).

TABLE 5-16. Constructing a Treeform with a !.

To make this work a few odd patches are needed. (Large systems seldom spring forth full-grown as from the forehead of Athena; they evolve by small changes.) The change to `tree-form-trace` in Table 5-15 has nothing to do with the "!" convention, but rather arranges for assumption cells to be "cuts" in the same way constant cells are. One change to `tree-form-deep-trace` causes "!" not to be treated as a pin name; the other, and also the change to `tree-form-deep` , allows a treeform not to exist for some constraints, in which case that constraint is passed by and another tried. This will be useful later for avoiding the use of certain uninteresting constraint-types in explanations.

```
(defun tree-form-trace (cell shallow)
  (require-cell cell)
  (cond ((node-boundp cell)
         (let ((s (node-supplier cell)))
           (cond ((cell-owner s)
                  (and (get (node-rule s) 'tentative) (nummark cell))   ;crock
                  (or shallow
                      (tree-form-trace-set (cell-owner s)
                                           (get (node-rule s) 'trigger-names)
                                           shallow)))
                 (t (nummark cell)))))                                          ;crock
        (t (let ((cells (node-cells cell)))
             (setf (node-supplier cell)
                   (or (if shallow
                           (or (tree-form-shallow cell cells)
                               (tree-form-deep cell cells shallow))
                           (or (tree-form-deep cell cells shallow)
                               (tree-form-shallow cell cells)))
                       (if (cell-owner cell)
                           (tree-form-deep-trace cell shallow)
                           cell)))))))

(defun tree-form-deep (cell cells shallow)
  (do ((z cells (cdr z)))
      ((null z) ())
    (and (not (eq (car z) cell))
         (cell-owner (car z))
         (let ((q (tree-form-deep-trace (car z) shallow)))
           (and q (return q))))))

(defun tree-form-deep-trace (cell shallow)
  (let ((treeform
          (cadr (assq (cell-name cell)
                      (get (ctype-name (con-ctype (cell-owner cell)))
                           'treeforms)))))
    (cond (treeform
           (tree-form-trace-set (cell-owner cell)
                                (remq '! (cdr treeform))
                                shallow)
           cell))))
```

Compare this with Table 3-16 (page 98).

TABLE 5-15. Tracing Missing Treeforms and Treeforms with !.

Finally, `tree-form-chase` must fill in the info component when it sees a "`!`" in the treeform; this is shown in Table 5-16 (part of the code has been omitted; it is the same as in Table 3-17 (page 100)).

The code in Table 5-17 has nothing whatsoever to do with assumptions; it just patches a bug described in §4.2, where a solution had been promised. The patch is that `process-setc`, before signalling a contradiction, remembers the *values* which triggered the rule which invoked `process-setc`. The `setc` is retried only if all the triggers still have those values.

```
(defun process-setc (*me* name cell value rule)
  ...
                    rule)
              (let ((values (forlist (tr triggers) (node-contents tr))))
                (process-contradiction (cons cell triggers))
                (do ((x triggers (cdr x))
                     (v values (cdr v)))
                    ((null x) (process-setc *me* name cell value rule))
                  (or (and (node-boundp (car x))
                           (equal (node-contents (car x)) (car v)))
                      (return)))))))))
```

This code patches a problem in the code in Table 4-3 (page 124).

TABLE 5-17. A More Reliable Version of `process-setc`.

The solution was delayed until this chapter, rather than being given in Chapter Four, because now we are in a position to poke a hole in this solution. With the advent of such strange rules as `assumption-rule`, which in effect trigger on the *absence* of a value rather than the presence of one, it is not clear that this patch is adequate. It will work for presently defined rules, but may not be general enough for other types of rules.

5.4. Examples of the Use of Assumptions

To illustrate the uses of assumptions, two examples are given here. One illustrates the special cases needed to awaken `assume` cells; the other uses `oneof` cells and some additional constraints to solve the four queens problem.

5.4.1. Simple Assumptions Are Persistent

To exhibit the behavior of simple assumptions, we will ring the changes on a simple `maxer` device. In the following example, all `ctrace` output concerning the awakening of devices has been suppressed without trace (pun intended). All other trace output is shown here.

```
(create u maxer)
<U:MAXER-61>
```

If the `a` is assumed to be 1 and the `b` is assumed to be 2, then from these assumed values the maximum 2 can be computed.

```
(== (the a u) (assume 1))
;|<ASSUMPTION-68:ASSUMPTION-68> computed 1 for its part PIN.
DONE
(== (the b u) (assume 2))
;|<ASSUMPTION-71:ASSUMPTION-71> computed 2 for its part PIN.
;|<U:MAXER-61> computed 2 for its part C from pins A, B.
DONE
```

Interrogation indicates that indeed the `c` was computed in this way.

```
(what (the c u))
;The value 2 in CELL-67 was computed in this way:
;   (THE C U) ← (MAX (ASSUMPTION 1) (ASSUMPTION 2))
OKAY?
```

Now we shall insist (by a `default` statement) that the `c` really ought to be 3.

```
(== (the c u) (default 3))
;|Contradiction when merging <CELL-67 (C of U): 2> and <CELL-75 (DEFAULT): 3>.
;|Deeming 2 in CELL-73 (computed by rule ASSUMPTION-RULE) to be the culprit.
;|The set (THE PIN ASSUMPTION-68)=1, (THE PIN ASSUMPTION-71)=2,
;|        CELL-75=3 is no good.
;|Retracting the premise <CELL-73 (PIN of ASSUMPTION-71): 2>.
;|Removing 2 from (THE PIN ASSUMPTION-71).
;|Removing 2 from (THE C U) because (THE B U)==(THE PIN ASSUMPTION-71).
```

```
;|<U:MAXER-61> computed 3 for its part B from pins A, C.
DONE
```

This of course caused a contradiction between the default value 3 and the computed value 2. Because the latter was computed from assumptions, one of the assumptions was arbitrarily chosen to be the culprit and retracted.[4]

```
(what (the c u))
;The value 3 in CELL-67 was computed in this way:
;   (THE C U) ← 3
OKAY?
```

Indeed the value 2 has disappeared, and been replaced by the specified `default` value.

```
(what (the a u))
;The value 1 in CELL-63 was computed in this way:
;   (THE A U) ← (ASSUMPTION 1)
OKAY?
```

The assumption for `a` is still in force.

```
(what (the b u))
;The value 3 in CELL-65 was computed in this way:
;   (THE B U) ← (ARCMAX 3 (ASSUMPTION 1))
OKAY?
```

On the other hand, the assumption for `b` has been retracted, and `b` was computed from the default value 3 and the assumption 1.

Now, to make things more complicated, let us insist that `a` be 5.

```
(== (the a u) (default 5))
;|Contradiction when merging <CELL-63 (A of U): 1> and <CELL-77 (DEFAULT): 5>.
;|Deeming 1 in CELL-70 (computed by rule ASSUMPTION-RULE) to be the culprit.
;|The set (THE PIN ASSUMPTION-68)=1, CELL-77=5 is no good.
;|Retracting the premise <CELL-70 (PIN of ASSUMPTION-68): 1>.
;|Removing 1 from (THE PIN ASSUMPTION-68).
```

The default value 5 conflicted with the assumed value 1, and the latter was therefore retracted. A nogood set was formed in the process.

```
;|Removing 3 from (THE B U) because (THE A U)==(THE PIN ASSUMPTION-68).
;|<ASSUMPTION-71:ASSUMPTION-71> computed 2 for its part PIN.
;|<U:MAXER-61> computed 3 for its part A from pins B, C.
```

4. Note that if several assumptions are involved, the system currently chooses one arbitrarily. It might be useful to have a "hook" to allow a user function to discriminate among assumptions.

The value 3 for b had been computed from the assumption 1, and so must be retracted also. Once this is done, the old assumption for b is free to re-assert the value 2. From this assumption and the value 3 on c, the value $3 = \text{arcmax}_3 2$ can be computed for a. This of course contradicts the default value 5 just placed there.

```
;|Contradiction when merging <CELL-63 (A of U): 3> and <CELL-77 (DEFAULT): 5>.
;|Deeming 2 in CELL-73 (computed by rule ASSUMPTION-RULE) to be the culprit.
;|The set (THE PIN ASSUMPTION-71)=2, CELL-75=3, CELL-77=5 is no good.
;|Retracting the premise <CELL-73 (PIN of ASSUMPTION-71): 2>.
;|Removing 2 from (THE PIN ASSUMPTION-71).
;|Removing 3 from (THE A U) because (THE B U)==(THE PIN ASSUMPTION-71).
```

The contradiction rested on the assumption of 2 for b, and so it was deemed the culprit and retracted again, along with its consequences.

There remains a more fundamental contradiction, however: the default value 5 for a is incompatible with the default value 3 for c.

```
;|Contradiction in <U:MAXER-61> among these parts: A=5, C=3.
;;; These are the premises that seem to be at fault:
;        <CELL-77 (DEFAULT): 5>,
;        <CELL-75 (DEFAULT): 3>.
;;; Choose one of these to retract and RETURN it.
```

We choose to retract the value 3 from c.

```
(return cell-75)
;|Retracting the premise <CELL-75 (DEFAULT): 3>.
;|Removing 3 from CELL-75.
;|<ASSUMPTION-71:ASSUMPTION-71> computed 2 for its part PIN.
;|<U:MAXER-61> computed 5 for its part C from pins A, B.
DONE
```

Once the value 3 has been retracted, the assumption for b is free to re-assert the value 2. This occurs because when the default value 3 is forgotten for c, the nogood set $\{(b, 2), (c, 3), (a, 5)\}$ is examined in the function `forget` and all relevant owners awakened. From this assumed value 2 and the default value 5, the value 5 is computed for c.

```
(what (the a u))
;The value 5 in CELL-63 was computed in this way:
;   (THE A U) ← 5
OKAY?
(what (the b u))
;The value 2 in CELL-65 was computed in this way:
;   (THE B U) ← (ASSUMPTION 2)
OKAY?
(what (the c u))
```

```
;The value 5 in CELL-67 was computed in this way:
;   (THE C U) ← (MAX 5 (ASSUMPTION 2))
OKAY?
```

Now $a = 5, b = 2$, and $c = 5$.

Suppose now that we assert the default value 0 for c. This is similar to the situation earlier where 3 was asserted for c, with one difference: then, the assumed values $a = 1$ and $b = 2$ were individually compatible with $c = 3$, and only in combination contradictory; here, however, the assumptions are individually incompatible with $c = 0$, and so we expect *both* assumptions to be suppressed.

```
(== (the c u) (default 0))
;|Contradiction when merging <CELL-67 (C of U): 5> and <CELL-79 (DEFAULT): 0>.
;|Deeming 2 in CELL-73 (computed by rule ASSUMPTION-RULE) to be the culprit.
;|The set (THE PIN ASSUMPTION-71)=2, CELL-77=5, CELL-79=0 is no good.
;|Retracting the premise <CELL-73 (PIN of ASSUMPTION-71): 2>.
;|Removing 2 from (THE PIN ASSUMPTION-71).
;|Removing 5 from (THE C U) because (THE B U)==(THE PIN ASSUMPTION-71).
```

The computed value 5 in c conflicted with the new value 0, and was withdrawn because it depended on an assumption.

```
;|Contradiction in <U:MAXER-61> among these parts: A=5, C=0.
;;; These are the premises that seem to be at fault:
;       <CELL-77 (DEFAULT): 5>,
;       <CELL-79 (DEFAULT): 0>.
;;; Choose one of these to retract and RETURN it.
```

Moreover, the value 5 in for a conflicts with the value 0 for c. We will retract the value 5 for a.

```
(return cell-79)
;|Retracting the premise <CELL-77 (DEFAULT): 5>.
;|Removing 5 from CELL-77.
```

At this (highly volatile!) point, the only value extant is 0 for c. However, the assumptions are about to be awakened.

```
;|<ASSUMPTION-71:ASSUMPTION-71> computed 2 for its part PIN.
;|Contradiction in <U:MAXER-61> among these parts: B=2, C=0.
;|Deeming 2 in CELL-73 (computed by rule ASSUMPTION-RULE) to be the culprit.
;|The set (THE PIN ASSUMPTION-71)=2, CELL-79=0 is no good.
;|Retracting the premise <CELL-73 (PIN of ASSUMPTION-71): 2>.
;|Removing 2 from (THE PIN ASSUMPTION-71).
```

The assumption for b tries out the value 2 and is rebuffed. A nogood set is formed, and the assumption retracted.

```
;|<ASSUMPTION-68:ASSUMPTION-68> computed 1 for its part PIN.
;|Contradiction in <U:MAXER-61> among these parts: A=1, C=0.
;|Deeming 1 in CELL-70 (computed by rule ASSUMPTION-RULE) to be the culprit.
;|The set (THE PIN ASSUMPTION-68)=1, CELL-79=0 is no good.
;|Retracting the premise <CELL-70 (PIN of ASSUMPTION-68): 1>.
;|Removing 1 from (THE PIN ASSUMPTION-68).
DONE
```

Precisely the same fate befalls the other assumption of 1 for a. It is still the case that the only extant value is 0 for c. However, it is now known *why* the assumptions cannot hold, and this information has been recorded in nogood sets.

```
(what (the a u))
;CELL-63 has no value.  I can express it in this way:
;   (THE A U) = (ASSUMPTION 1)
OKAY?
(what (the b u))
;CELL-65 has no value.  I can express it in this way:
;   (THE B U) = (ASSUMPTION 2)
OKAY?
```

These explanations are a little strange. Probably `what` should be augmented to use nogood information to explain the absence of a value, but this thought is not pursued here.

Let us finally disconnect c from the other cells of its node (and in particular the `default` cell supplying the value 0).

```
(disconnect (the c u))
;|Disconnecting (THE C U) from CELL-75, CELL-79.
;|Removing 0 from (THE C U).
;|<ASSUMPTION-71:ASSUMPTION-71> computed 2 for its part PIN.
;|<ASSUMPTION-68:ASSUMPTION-68> computed 1 for its part PIN.
;|<U:MAXER-61> computed 2 for its part C from pins A, B.
DONE
```

Disconnecting c from the source of the value 0 causes all the old nogood information to be expunged. This awakens the assumptions, which find no nogood sets to suppress their values. From the assumptions a new value is computed for c.

```
(what (the c u))
;The value 2 in CELL-67 was computed in this way:
;   (THE C U) ← (MAX (ASSUMPTION 1) (ASSUMPTION 2))
OKAY?
```

This brings us full circle, to the beginning of the example.

```
(declare (special *contradictions* *backtracks*))

(defun queens (n)
  (setq *contradictions* 0)
  (setq *backtracks* 0)
  (queensearch '() n 0)
  (format t "~%Total of ~D contradictions and ~D backtracks."
          *contradictions* *backtracks*)
  'done)

(defun queensearch (previous n k)
  (cond ((= k n)
         (format
             t
             "~%Solution: (~{~D~^,~}) after ~D contradictions and ~D backtracks."
             (reverse previous) *contradictions* *backtracks*))
        (t (dotimes (i n)
             (do ((x previous (cdr x))
                  (j 1 (+ j 1)))
                 ((null x)
                  (queensearch (cons i previous) n (+ k 1)))
               (cond ((or (= i (car x))                ;column test
                          (= (- i (car x)) j)          ;diagonal test
                          (= (- (car x) i) j))         ;other diagonal test
                      (ctrace "Contradiction: (~{~D~^,~}) kills ~D."
                              (reverse previous) i)
                      (increment *contradictions*)
                      (return)))))
           (increment *backtracks*))))
```

TABLE 5-18. A LISP Solution to the N Queens Problem.

5.4.2. Oneof Assumptions Can Express and Solve the Four Queens Problem

The generalized N queens problem is that of placing N chess queens on an N by N chessboard so that no two queens attack each other; that is to say, no two queens are on the same row, column, or diagonal. The usual approach notes that every row must have exactly one queen on it, and then tries to place one queen on each row. Using this idea the problem may be formulated as: for $0 \leq i < N$ find $0 \leq q_i < N$ such that

$$\forall i \, \forall j \;\; ((0 \leq i < N \wedge 0 \leq j < N \wedge i \neq j) \Rightarrow \\ (q_j \neq q_i \wedge q_j - q_i \neq j - i \wedge q_j - q_i \neq i - j))$$

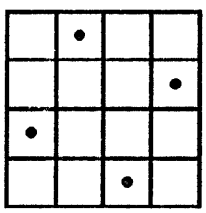
This is a standard problem used to illustrate backtracking control structures, because a solution to the problem can easily be expressed as a non-deterministic program: for each row, non-deterministically choose a column in that row; then check to see whether there is a conflict on any column or diagonal. In a sequential simulation of a non-deterministic program, a conflict causes a failure back to the most recent choice point.

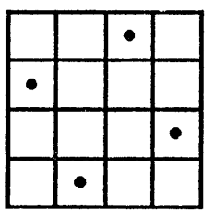
A LISP program for the usual solution to the N queens problem is shown in Table 5-18. Rather than using explicit backtracking and failure mechanisms, it merely takes advantage of the observation in [Sussman 1972] that chronological backtracking mechanisms are equivalent to a series of nested `do` loops. Each recursive call to `queensearch` tries to choose a column for one row (row k; rows and columns are numbered starting with 0). It loops over all choices from 0 to $N - 1$ using `dotimes`, and for each choice checks for a conflict with all previous choices (which are in the list `previous`). If a conflict is found, a contradiction is noted via the trace mechanism and a counter `*contradictions*` incremented (for statistical, not algorithmic, purposes). If no conflict exists, the choice is added to the `previous` list and a recursive call made to choose for the next row. If all choices fail, either immediately or because a recursive call returned, then `queensearch` returns (after incrementing another counter, `*backtracks*`) so that the previous row may try a new choice. (The program as it stands will find *all* solutions, not just one. To find just one, a non-local exit could be made after printing a solution.)

As an example of running the `queens` program, here is the output (with tracing turned off) for the cases $N = 4$, $N = 6$, and (in part) $N = 8$:

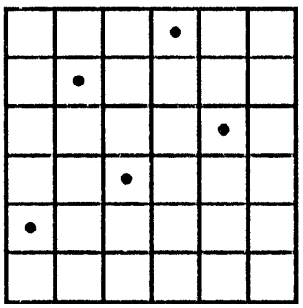
```
NIL
(queens 4)
Solution: (1,3,0,2) after 18 contradictions and 4 backtracks.
Solution: (2,0,3,1) after 26 contradictions and 7 backtracks.
Total of 44 contradictions and 15 backtracks.
DONE
(queens 6)
Solution: (1,3,5,0,2,4) after 140 contradictions and 25 backtracks.
Solution: (2,5,1,4,0,3) after 334 contradictions and 64 backtracks.
Solution: (3,0,4,1,5,2) after 408 contradictions and 79 backtracks.
Solution: (4,2,0,5,3,1) after 602 contradictions and 118 backtracks.
Total of 742 contradictions and 149 backtracks.
DONE
(queens 8)
Solution: (0,4,7,5,2,6,1,3) after 763 contradictions and 105 backtracks.
        [Ninety solutions omitted.]
Solution: (7,3,0,2,5,1,6,4) after 12901 contradictions and 1852 backtracks.
Total of 13664 contradictions and 1965 backtracks.
DONE
```

The two solutions for $N = 4$ are:





The trouble with chronological backtracking is that often choices are undone because of failures that did not (necessarily) stem from those choices. Suppose, for example, that for the 6 queens problem queens have been successfully placed in the first four rows, and a choice is to be made for the last row:



None of the squares of the last row is a valid place for a queen. Under a chronological backtracking regime, this failure will first cause a new choice for the queen in the second-to-last row. This is somewhat paradoxical, as the configuration for the first four rows collectively kills all squares of the last row, and so cannot appear in any valid solution, while the queen in the penultimate row can appear in that column in a valid solution!

Another problem with chronological backtracking is that when a failure occurs all information as to why that failure occurred is thrown away. [Sussman 1972] In the context of the N queens problem, it can well occur that a large series of configurations for the last several rows is tried and discarded, then failure causes one queen in an early row to be nudged over, and then many of the same configurations of the last several rows must be investigated once again—even if their failure had not depended on the queen that got nudged!

The LISP program of Table 5-18 examines eighteen invalid board positions before finding a solution. These are shown in Figure 5-8. The small dark circles indicate queens. A line drawn between two queens indicates a conflict on a column or diagonal. A light circle around a queen indicates the culprit—the one which will be changed as a result of the contradiction (under chronological backtracking, the culprit is always the last queen placed). A bold circle around a queen indicates an indirect culprit—a queen that must be moved because all the choices for the previous culprit had been exhausted.

The contradiction in situation (g) is the same as that in situation (c). This contradiction had to be rediscovered when the queen in the second row was moved, even though that queen had had nothing to do with the contradiction. There are no other examples of this in the $N = 4$ case, but for large N it happens quite frequently.

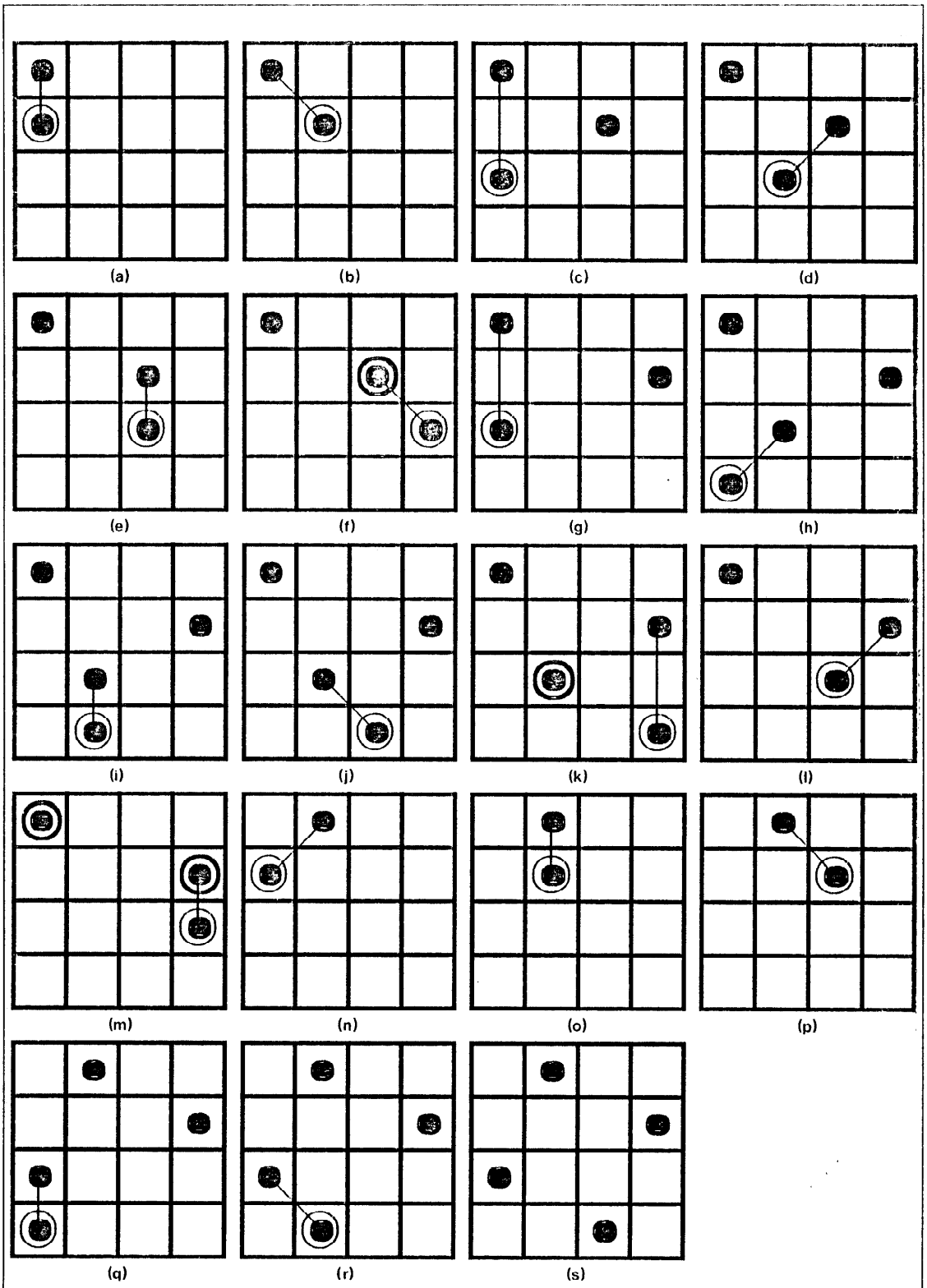


FIGURE 5-8. Situations Examined for Four Queens Using Chronological Backtracking.

```
(variable q0) (variable q1) (variable q2) (variable q3) ;variables
(create e01 equality)                                    ;column constraints
(create e02 equality)
(create e03 equality)
(create e12 equality)
(create e13 equality)
(create e23 equality)
(== (the a e01) q0) (== (the b e01) q1) (== (the p e01) (constant 0))
(== (the a e02) q0) (== (the b e02) q2) (== (the p e02) (constant 0))
(== (the a e03) q0) (== (the b e03) q3) (== (the p e03) (constant 0))
(== (the a e12) q1) (== (the b e12) q2) (== (the p e12) (constant 0))
(== (the a e13) q1) (== (the b e13) q3) (== (the p e13) (constant 0))
(== (the a e23) q2) (== (the b e23) q3) (== (the p e23) (constant 0))
```

TABLE 5-19. Constraints for the Four Queens Problem (i).

```
(create xe01 equality) (create xa01 adder)              ;northwest-to-southeast
(create xe02 equality) (create xa02 adder)              ; diagonal constraints
(create xe03 equality) (create xa03 adder)
(create xe12 equality) (create xa12 adder)
(create xe13 equality) (create xa13 adder)
(create xe23 equality) (create xa23 adder)
(== (the a xa01) q0) (== (the c xa01) q1) (== (the b xa01) (the a xe01))
(== (the a xa02) q0) (== (the c xa02) q2) (== (the b xa02) (the a xe02))
(== (the a xa03) q0) (== (the c xa03) q3) (== (the b xa03) (the a xe03))
(== (the a xa12) q1) (== (the c xa12) q2) (== (the b xa12) (the a xe12))
(== (the a xa13) q1) (== (the c xa13) q3) (== (the b xa13) (the a xe13))
(== (the a xa23) q2) (== (the c xa23) q3) (== (the b xa23) (the a xe23))
(== (the b xe01) (constant 1)) (== (the p xe01) (constant 0))
(== (the b xe02) (constant 2)) (== (the p xe02) (constant 0))
(== (the b xe03) (constant 3)) (== (the p xe03) (constant 0))
(== (the b xe12) (constant 1)) (== (the p xe12) (constant 0))
(== (the b xe13) (constant 2)) (== (the p xe13) (constant 0))
(== (the b xe23) (constant 1)) (== (the p xe23) (constant 0))
```

TABLE 5-20. Constraints for the Four Queens Problem (ii).

```
(create ye01 equality) (create ya01 adder)              ;southwest-to-northeast
(create ye02 equality) (create ya02 adder)              ; diagonal constraints
(create ye03 equality) (create ya03 adder)
(create ye12 equality) (create ya12 adder)
(create ye13 equality) (create ya13 adder)
(create ye23 equality) (create ya23 adder)
(== (the a ya01) q0) (== (the c ya01) q1) (== (the b ya01) (the a ye01))
(== (the a ya02) q0) (== (the c ya02) q2) (== (the b ya02) (the a ye02))
(== (the a ya03) q0) (== (the c ya03) q3) (== (the b ya03) (the a ye03))
(== (the a ya12) q1) (== (the c ya12) q2) (== (the b ya12) (the a ye12))
(== (the a ya13) q1) (== (the c ya13) q3) (== (the b ya13) (the a ye13))
(== (the a ya23) q2) (== (the c ya23) q3) (== (the b ya23) (the a ye23))
(== (the b ye01) (constant -1)) (== (the p ye01) (constant 0))
(== (the b ye02) (constant -2)) (== (the p ye02) (constant 0))
(== (the b ye03) (constant -3)) (== (the p ye03) (constant 0))
(== (the b ye12) (constant -1)) (== (the p ye12) (constant 0))
(== (the b ye13) (constant -2)) (== (the p ye13) (constant 0))
(== (the b ye23) (constant -1)) (== (the p ye23) (constant 0))
```

TABLE 5-21. Constraints for the Four Queens Problem (iii).

```
(== q0 (oneof '(0 1 2 3)))                              ;assumptions
(== q1 (oneof '(0 1 2 3)))
(== q2 (oneof '(0 1 2 3)))
(== q3 (oneof '(0 1 2 3)))
```

TABLE 5-22. Constraints for the Four Queens Problem (iv).

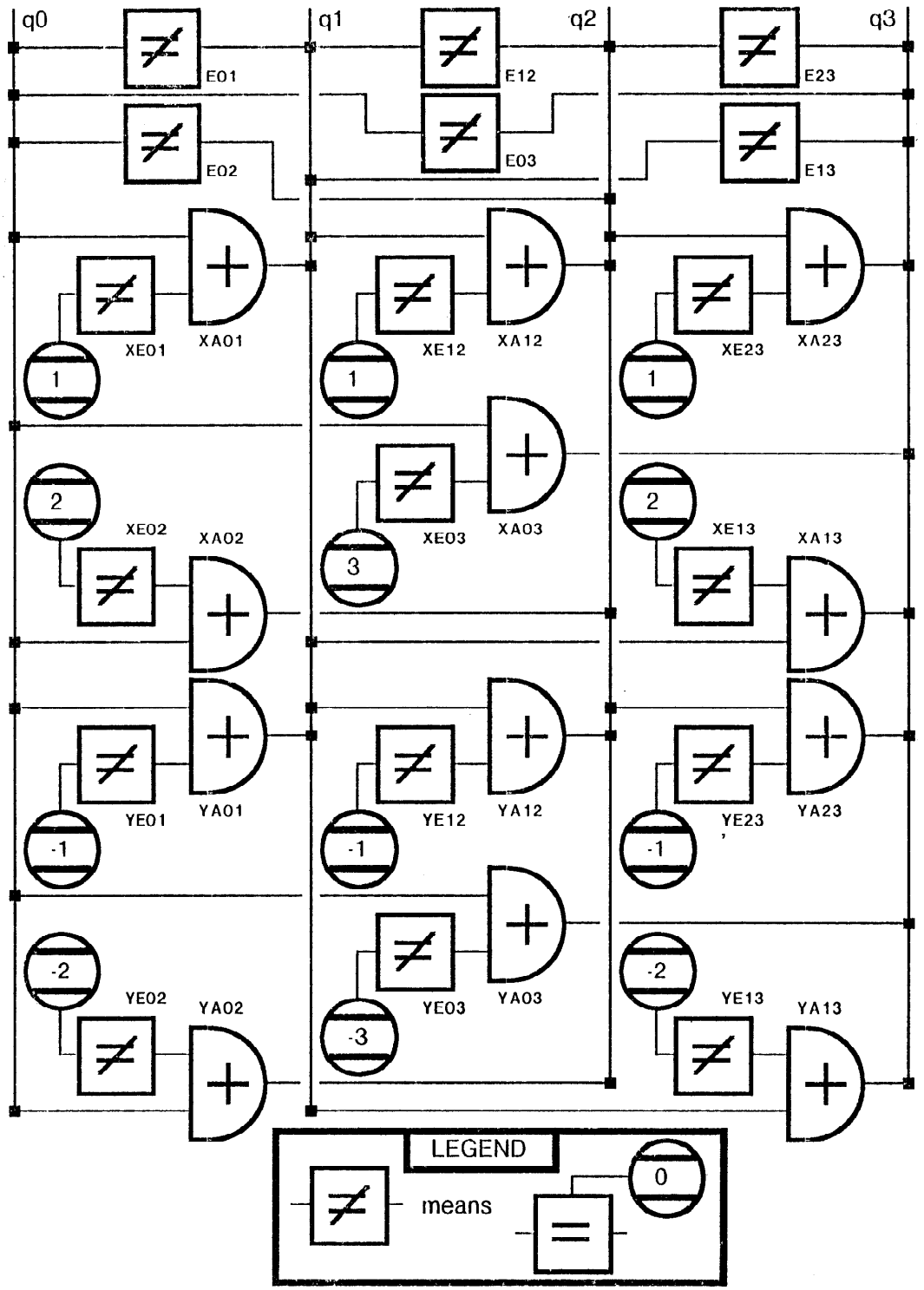


FIGURE 5-9. Constraint Network for the Four Queens Problem.

Table 5-19, Table 5-20, Table 5-21, and Table 5-22 show the constraints for the case of four queens. The variables q0, q1, q2, and q3 represent the column numbers of the queens in rows 0, 1, 2, and 3, respectively. The equalities e *mn* have their p pins equated to zero, and so require that q*m* and q*n* be different, for each pair *m*, *n*. The equalities xe*mn* and the adders xa*mn* enforce the relationships q*n* — q*n* ≠ *n* — *m*; similarly, the equalities ye*mn* and the adders ya*mn* enforce the relationships q*n* — q*n* ≠ *m* — *n*. The constraints are diagrammed in Figure 5-9.

Running this constraint network causes twelve contradictions to occur before a valid situation is achieved (valid situations of course constitute solutions to the problem). The sequence of situations considered is shown in Figure 5-10. It initially follows the same sequence of situations as in Figure 5-8, except that situation (g) is skipped over (because that contradiction had been explored already, and the record shows that it is independent of q1). Note, however, that unlike the LISP program of Table 5-18, the constraint system does not guarantee to check the constraints in any particular order. The LISP program always finds a contradiction with the most recent already placed queen that conflicts, because it searches the rows in that order. The constraint language does not specify any temporal ordering, and the system is free to check the constraints in any order (or even in parallel). Thus, for example, in situation (i) the system happened to record a conflict between q3 and q1 rather than between q3 and q2. Either conflict is an equally good reason for rejecting the situation. Similarly, in situation (k) the system noted a contradiction between q3 and q0 where the LISP program had seen a conflict between q3 and q1. Moreover, in situation (k) the LISP program chose q2 as the indirect culprit, because it must always retract the most recent choice, whether relevant or not; but the constraint system was free to choose *any* previous *relevant* choice as the culprit, and in fact it serendipitously chose q0, producing situation (x). From there it was only two more steps to a solution. (Note that a situation (z) with q3 = 1 was skipped over because of the contradiction previously recorded for situation (i).)

The trace output from the run is given here in condensed form without commentary. Trace messages concerning awakening of devices and removing of values have been eliminated, as have messages saying that devices computed values for their parts, except that those concerning the oneof cells have been retained.

```
;|<ONEOF-889:ONEOF-889> computed 0 for its part PIN.
;|<ONEOF-892:ONEOF-892> computed 0 for its part PIN.
;|Contradiction in <E01:EQUALITY-619> among these parts: P=0, A=0, B=0;
;|  it calculated 1 for P from the others by rule EQUALITY-RULE-16.
;|Deeming 0 in CELL-894 (computed by rule ONEOF-RULE) to be the culprit.
;|The set CELL-662=0, (THE PIN ONEOF-889)=0, (THE PIN ONEOF-892)=0 is no good.
;|Retracting the premise <CELL-894 (PIN of ONEOF-892): 0>.
;|<ONEOF-892:ONEOF-892> computed 1 for its part PIN.
;|Contradiction in <XE01:EQUALITY-673> among these parts: P=0, A=1, B=1;
```

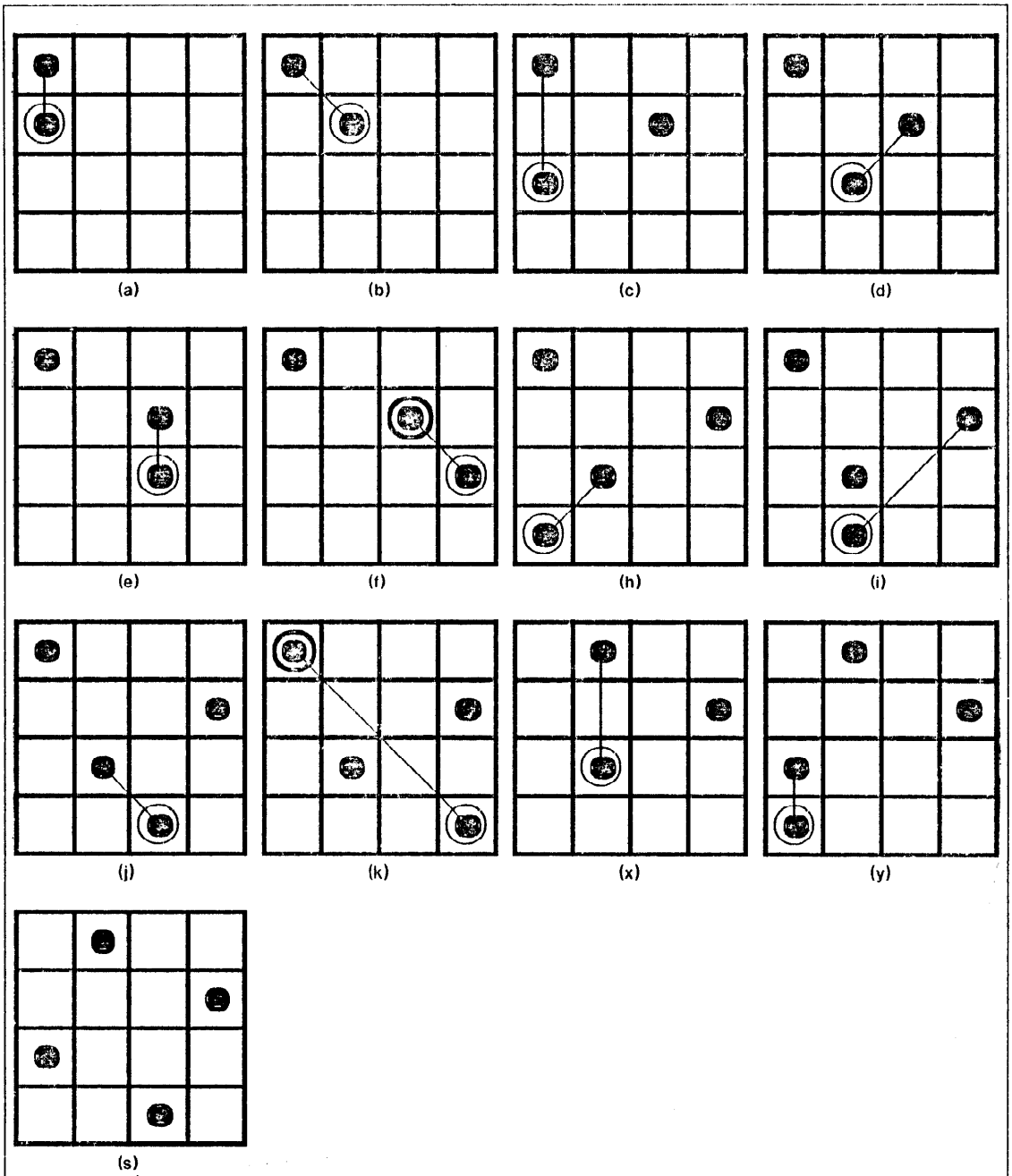


FIGURE 5-10. Situations Examined for Four Queens Using Non-chronological Backtracking.

```
;|  it calculated 1 for P from the others by rule EQUALITY-RULE-16.
;|Deeming 1 in CELL-894 (computed by rule ONEOF-RULE) to be the culprit.
;|The set CELL-758=1, CELL-760=0, (THE PIN ONEOF-889)=0,
;|        (THE PIN ONEOF-892)=1 is no good.
```

```
;|Retracting the premise <CELL-894 (PIN of ONEOF-892): 1>.
;|<ONEOF-892:ONEOF-892> computed 2 for its part PIN.
;|<ONEOF-895:ONEOF-895> computed 0 for its part PIN.
;|Contradiction in <E02:EQUALITY-626> among these parts: P=0, A=0, B=0;
;|  it calculated 1 for P from the others by rule EQUALITY-RULE-16.
;|Deeming 0 in CELL-897 (computed by rule ONEOF-RULE) to be the culprit.
;|The set CELL-664=0, (THE PIN ONEOF-889)=0, (THE PIN ONEOF-895)=0 is no good.
;|Retracting the premise <CELL-897 (PIN of ONEOF-895): 0>.
;|<ONEOF-895:ONEOF-895> computed 1 for its part PIN.
;|Contradiction in <YE12:EQUALITY-823> among these parts: P=0, A=-1, B=-1;
;|  it calculated 1 for P from the others by rule EQUALITY-RULE-16.
;|Deeming 1 in CELL-897 (computed by rule ONEOF-RULE) to be the culprit.
;|The set CELL-878=-1, CELL-880=0, (THE PIN ONEOF-892)=2,
;|        (THE PIN ONEOF-895)=1 is no good.
;|Retracting the premise <CELL-897 (PIN of ONEOF-895): 1>.
;|<ONEOF-895:ONEOF-895> computed 2 for its part PIN.
;|Contradiction in <XE02:EQUALITY-687> among these parts: P=0, A=2, B=2;
;|  it calculated 1 for P from the others by rule EQUALITY-RULE-16.
;|Deeming 2 in CELL-897 (computed by rule ONEOF-RULE) to be the culprit.
;|The set CELL-762=2, CELL-764=0, (THE PIN ONEOF-889)=0,
;|        (THE PIN ONEOF-895)=2 is no good.
;|Retracting the premise <CELL-897 (PIN of ONEOF-895): 2>.
;|<ONEOF-895:ONEOF-895> computed 3 for its part PIN.
;|Contradiction in <XE12:EQUALITY-715> among these parts: P=0, A=1, B=1;
;|  it calculated 1 for P from the others by rule EQUALITY-RULE-16.
;|Deeming 3 in CELL-897 (computed by rule ONEOF-RULE) to be the culprit.
;|The set CELL-770=1, CELL-772=0, (THE PIN ONEOF-892)=2,
;|        (THE PIN ONEOF-895)=3 is no good.
;|Retracting the premise <CELL-897 (PIN of ONEOF-895): 3>.
;|All of the values (0 1 2 3) for (THE PIN ONEOF-895) are no good.
;|Deeming 2 in CELL-894 (computed by rule ONEOF-RULE) to be the culprit.
;|The set CELL-664=0, CELL-762=2, CELL-764=0, CELL-770=1, CELL-772=0,
;|        CELL-878=-1, CELL-880=0, (THE PIN ONEOF-889)=0,
;|        (THE PIN ONEOF-892)=2 is no good.
;|Retracting the premise <CELL-894 (PIN of ONEOF-892): 2>.
;|<ONEOF-892:ONEOF-892> computed 3 for its part PIN.
;|<ONEOF-895:ONEOF-895> computed 1 for its part PIN.
;|<ONEOF-898:ONEOF-898> computed 0 for its part PIN.
;|Contradiction in <YE23:EQUALITY-851> among these parts: P=0, A=-1, B=-1;
;|  it calculated 1 for P from the others by rule EQUALITY-RULE-16.
;|Deeming 0 in CELL-900 (computed by rule ONEOF-RULE) to be the culprit.
;|The set CELL-886=-1, CELL-888=0, (THE PIN ONEOF-895)=1,
;|        (THE PIN ONEOF-898)=0 is no good.
;|Retracting the premise <CELL-900 (PIN of ONEOF-898): 0>.
;|<ONEOF-898:ONEOF-898> computed 1 for its part PIN.
;|Contradiction in <YE13:EQUALITY-837> among these parts: P=0, A=-2, B=-2;
;|  it calculated 1 for P from the others by rule EQUALITY-RULE-16.
;|Deeming 1 in CELL-900 (computed by rule ONEOF-RULE) to be the culprit.
```

```
;|The set CELL-882=-2, CELL-884=0, (THE PIN ONEOF-892)=3,
;|        (THE PIN ONEOF-898)=1 is no good.
;|Retracting the premise <CELL-900 (PIN of ONEOF-898): 1>.
;|<ONEOF-898:ONEOF-898> computed 2 for its part PIN.
;|Contradiction in <XE23:EQUALITY-743> among these parts: P=0, A=1, B=1;
;|  it calculated 1 for P from the others by rule EQUALITY-RULE-16.
;|Deeming 2 in CELL-900 (computed by rule ONEOF-RULE) to be the culprit.
;|The set CELL-778=1, CELL-780=0, (THE PIN ONEOF-895)=1,
;|        (THE PIN ONEOF-898)=2 is no good.
;|Retracting the premise <CELL-900 (PIN of ONEOF-898): 2>.
;|<ONEOF-898:ONEOF-898> computed 3 for its part PIN.
;|Contradiction in <XE03:EQUALITY-701> among these parts: P=0, A=3, B=3;
;|  it calculated 1 for P from the others by rule EQUALITY-RULE-16.
;|Deeming 3 in CELL-900 (computed by rule ONEOF-RULE) to be the culprit.
;|The set CELL-766=3, CELL-768=0, (THE PIN ONEOF-889)=0,
;|        (THE PIN ONEOF-898)=3 is no good.
;|Retracting the premise <CELL-900 (PIN of ONEOF-898): 3>.
;|All of the values (0 1 2 3) for (THE PIN ONEOF-898) are no good.
;|Deeming 0 in CELL-891 (computed by rule ONEOF-RULE) to be the culprit.
;|The set CELL-766=3, CELL-768=0, CELL-778=1, CELL-780=0, CELL-882=-2,
;|        CELL-884=0, CELL-886=-1, CELL-888=0, (THE PIN ONEOF-889)=0,
;|        (THE PIN ONEOF-892)=3, (THE PIN ONEOF-895)=1 is no good.
;|Retracting the premise <CELL-891 (PIN of ONEOF-889): 0>.
;|<ONEOF-889:ONEOF-889> computed 1 for its part PIN.
;|Contradiction in <E02:EQUALITY-626> among these parts: P=0, A=1, B=1;
;|  it calculated 1 for P from the others by rule EQUALITY-RULE-16.
;|Deeming 1 in CELL-897 (computed by rule ONEOF-RULE) to be the culprit.
;|The set CELL-664=0, (THE PIN ONEOF-889)=1, (THE PIN ONEOF-895)=1 is no good.
;|Retracting the premise <CELL-897 (PIN of ONEOF-895): 1>.
;|<ONEOF-895:ONEOF-895> computed 0 for its part PIN.
;|<ONEOF-898:ONEOF-898> computed 0 for its part PIN.
;|Contradiction in <E23:EQUALITY-654> among these parts: P=0, A=0, B=0;
;|  it calculated 1 for P from the others by rule EQUALITY-RULE-16.
;|Deeming 0 in CELL-900 (computed by rule ONEOF-RULE) to be the culprit.
;|The set CELL-672=0, (THE PIN ONEOF-895)=0, (THE PIN ONEOF-898)=0 is no good.
;|Retracting the premise <CELL-900 (PIN of ONEOF-898): 0>.
;|<ONEOF-898:ONEOF-898> computed 2 for its part PIN.
DONE
```

This example shows that a dependency-directed backtracking system is at least potentially much more efficient than a chronological backtracking system. Of course, this run was a little lucky; it could just as easily have followed the same path as the LISP program, skipping only situation (g). If, however, there were a higher-level decision function controlling which constraints to try first, then the system might always perform much better. (Thus we are lead to the idea of meta-constraints, for controlling the operations of the constraint interpreter.) Suppose, for example, that a dependency-directed backtracking system for the N queens problem were always to obey these

additional efficiency heuristics:

- The first k queens must be validly placed before trying to place queen $k + 1$. (The constraint system happened to behave in this manner for the previous example, but the constraint language does not guarantee to try the assumptions in a nested-loop order. The system is in principle free to try assumptions in any order—but this fact is being suspended as a heuristic here.)
- When checking a placement for a queen and it conflicts with more than one previously placed queen, the *least* recently placed conflicting queen should be held responsible for the conflict. (This is equivalent to checking previous queens in the reverse of the order used by the LISP program.)
- When a culprit must be chosen, always choose the most recently placed queen of those responsible for the contradiction (according to the records).

If these ordering heuristics are followed, then sixteen invalid positions are tried before a solution is found.

To point up once more the need for explanation mechanisms to exploit the nogood sets, here are given the explanations for the values of `q0`, `q1`, `q2`, and `q3` at the end of the above run.

```
(what q0)
;The value 1 in CELL-612 was computed in this way:
;   Q0 ← (ONEOF (0 1 2 3))
OKAY?
(what q1)
;The value 3 in CELL-614 was computed in this way:
;   Q1 ← (ONEOF (0 1 2 3))
OKAY?
(what q2)
;The value 0 in CELL-616 was computed in this way:
;   Q2 ← (ONEOF (0 1 2 3))
OKAY?
(what q3)
;The value 2 in CELL-618 was computed in this way:
;   Q3 ← (ONEOF (0 1 2 3))
OKAY?
```

These explanations are singularly unsatisfying: they imply "I just guessed them." This is partly true, but fails to take into account the additional constraints imposed and the tremendous computational effort invested in satisfying them.

This entire example has assumed that the cost of avoiding examining a position by using nogood sets is less than the cost of just generating the position and checking it. This may not be the case for this example with this implementation of the constraint system. However, nogood sets can save a great deal when the cost of generating and checking a position is large. They can also save a

great deal when not all the choices are directly connected to each other. In the N queens problem, every choice interacts with every other choice. If each choice were to interact with only some other choices, then nogood sets can eliminate many more cases.

5.5. Discussion of the Assumption and Nogood Set Mechanisms

As of the end of Chapter Four, before the assumption mechanisms were introduced, the constraint system strove to compute the largest possible set of values that could be both consistently and determinately asserted. Consistency means that no constraints are violated; determinacy means that no arbitrary choices on the part of the system are involved—a computed value for a node must be the case, and no other value will do for that node. Any value that is forced is asserted, and only those that are forced. Thus the system would conservatively compute a *minimal* maximal set of values; let us call this the set of *required* values.

Assumption mechanisms allow a constraint network to compute a larger set of values. One could imagine a constraint system that would *automatically* make assumptions about the values of nodes when no forced values can be computed for them. Such a system would endeavor to find consistent values for the greatest possible number of nodes, in some sense. Such a set of values would perforce contain the set of required values as a subset. Thus we can say that an assumption mechanism tries to find consistent *extensions* of the set of required values.

One difficulty with a general, domain-independent automatic assumption mechanism is that it may well thrash, perhaps even trying to solve the unsolvable. It is all too easy to set up Diophantine equations whose solutions involve extremely large integers that would be infeasible to guess.

The mechanism we have exhibited here is a compromise between a fully automatic assumption mechanism and none at all. The assumption constructs added to the language permit the user to explicitly advise the system on which extensions to pursue and what values to try. The `assume` construct in effect says, "The extension for which this node has value n may be interesting, if it is consistent." By connecting several `assume` cells together, a number of alternative extensions involving the same node can be suggested, and the system can choose among them. In this way which nodes to consider for extension are explicitly indicated, and the search space for each node delineated. The `oneof` construct adds a little more power by providing a total predicate for the search possibilities. This gives one the leverage to perform exhaustive case analysis and perform resolution on nogood sets.

For some purposes it might be useful to separate two properties of the `oneof` construct: the limiting of the value space to a definite finite set, and the advice to try an extension by assuming one. If there were a construct `valuespace` for the former property alone, then the effect of

```
(== x (oneof '(a b c ...)))
```

```
(defprim firstoneof (pin))

(progn 'compile
       (push 'firstoneof-rule (ctype-rules firstoneof))
       (defprop firstoneof-rule () trigger-names)
       (defprop firstoneof-rule (pin) output-names)
       (defprop firstoneof-rule firstoneof tentative)
       '(firstoneof rule))

(defun firstoneof (valuelist)
  (let ((a (gen-constraint firstoneof ())))
    (setf (con-name a) (con-id a))
    (setf (con-info a) valuelist)
    (awaken a)
    (the pin a)))

(defprop firstoneof ((pin (firstoneof !))) treeforms)
```

Compare this with Table 5-3.

TABLE 5-23. Implementation of the `firstoneof` Construct.

could be achieved by

```
(== x (valuespace '(a b c ...)))
(== x (assume a))
(== x (assume b))
(== x (assume c))
   ⋮
```

The latter states that it is an error for `x` to take on a value not among *a*, *b*, *c*, . . ., and separately that each of these value may be considered for constructing extensions. There might be uses for wanting to advise the system that only some of the values are useful to try for extensions.

The notion of `valuespace` itself can be divided into two parts. One part is triggered when a new value is computed, and raises a contradiction if the value is not in the set. The other part is triggered when a value is forgotten, and examines nogood sets to see whether resolution can be performed. An instance of the first part is built into the `gate` and `equality` primitives of Table 2-7 (page 53) and Table 3-5 (page 79): each has a rule which signals a contradiction if the `p` pin has a value other than 0 or 1. However, the rule has no provision for making a deduction by resolution if both 0 and 1 are tried and fail. There seems to be no gain in having one part without the other.

The assumption mechanisms given here provide no means for ordering values to be tried, either locally (at a single node) or globally (among several nodes). The `oneof` construct, as implemented here, happens to try the values in the order stated. However, the definition of the construct at the user language level does not guarantee this; the system is free to try values in any order. A slightly different distinction is that the `oneof` construct does not guarantee always to assert the

earliest consistent value in the list. When it is asked to assume a new value, it happens (in this implementation) to scan the list in order, looking for possibilities. However, if the third value in the list is consistently asserted, and then a nogood set for some reason forbids the first value in the list to be consistently asserted, `oneof` will not notice this. The possibly useful construct `firstoneof` would notice this, and strive always to assert earlier values in the list if possible. That is, it would undertake always to construct an extension using the earliest possible value in its list. As an example, consider two parallel examples using `oneof` and `firstoneof`. In each case an adder is created, the `a` and `b` pins equated to 1, the `c` pin equated to an assumption, and then the `b` pin disconnected from its constant.

```
(create foo adder)
<FOO:ADDER-385>
(== (the a foo) (constant 1))
DONE
(== (the b foo) (constant 1))
DONE
(== (the c foo) (oneof '(0 1 2 3)))
DONE
(what (the c foo))
;The value 2 in CELL-391 was computed in this way:
;   (THE C FOO) ← (ONEOF (0 1 2 3))
OKAY?
(disconnect (the b foo))
DONE
(what (the c foo))
;The value 2 in CELL-391 was computed in this way:
;   (THE C FOO) ← (ONEOF (0 1 2 3))
OKAY?
(what (the b foo))
;The value 1 in CELL-389 was computed in this way:
;   (THE B FOO) ← (- (ONEOF (0 1 2 3)) 1)
OKAY?
```

The value 2 remains on the `c` pin--the `oneof` cell is happy with any of its four values. On the other hand:

```
(create bar adder)
<BAR:ADDER-400>
(== (the a bar) (constant 1))
DONE
(== (the b bar) (constant 1))
DONE
(== (the c foo) (firstoneof '(0 1 2 3)))
DONE
(what (the c bar))
;The value 2 in CELL-406 was computed in this way:
```

```
(defun firstoneof-rule (*me*)
  (let ((*rule* 'firstoneof-rule)
        (pin-cell (the pin *me*)))
    (let ((values (con-info *me*)))
      (do-named loop-over-possibilities
                ((v values (cdr v))
                 (killers '()))
                ((null v)
                 (ctrace "All of the values ~S for ~S are no good."
                         values
                         (cell-goodname pin-cell))
                 (let ((losers '()))
                   (dolist (killer killers)
                     (dolist (x (cdr killer))
                       (or (eq (car x) (cell-repository pin-cell))
                           (let ((cell (if (rep-boundp (car x))
                                           (rep-supplier (car x))
                                           (car (rep-cells (car x))))))        ;???
                             (or (memq cell losers)
                                 (push cell losers))))))
                   (process-contradiction losers))
                 (firstoneof-rule *me*))
        (do-named outer-loop
                  ((x (cdr (assoc (car v) (node-nogoods pin-cell)))
                      (cdr x)))
                  ((null x)
                   (cond ((node-boundp pin-cell)
                          (and (not (equal (node-contents pin-cell) (car v)))
                               (cond ((eq (node-supplier pin-cell) pin-cell)
                                      (retract pin-cell)
                                      (setc pin (car v)))
                                     (t (contradiction pin)))))
                         (t (setc pin (car v))))
                   (return-from loop-over-possibilities))
          (do-named inner-loop
                    ((c (cdar x) (cdr c)))
                    ((null c)
                     (push (car x) killers)
                     (return-from outer-loop))
            (and (not (eq (caar c) (cell-repository pin-cell)))
                 (or (not (rep-boundp (caar c)))
                     (not (equal (rep-contents (caar c)) (cdar c))))
                 (return-from inner-loop)))))))))
```

Compare this with Table 5-4.

TABLE 5-24. The Rule for `firstoneof`.

```
;   (THE C BAR) ← (FIRSTONEOF (0 1 2 3))
OKAY?
(disconnect (the b bar))
DONE
(what (the c bar))
;The value 0 in CELL-406 was computed in this way:
;   (THE C BAR) ← (FIRSTONEOF (0 1 2 3))
```

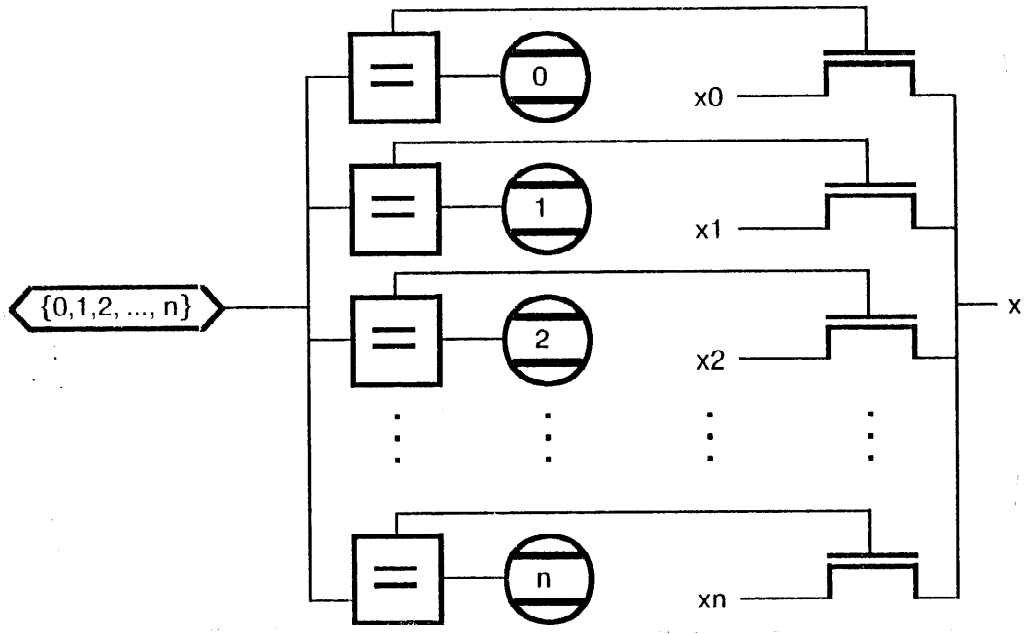


FIGURE 5-11. Constraint Network for Making a General Choice.

```
OKAY?
(what (the b bar))
;The value -1 in CELL-404 was computed in this way:
;   (THE B BAR) ← (- (FIRSTONEOF (0 1 2 3)) 1)
OKAY?
```

When the constant 1 was removed from `(the b bar)`, the `firstoneof` cell noted that the value 0 was no longer forbidden, and retracted the value 2 to try the value 0 again, which in fact worked. Thus `firstoneof` always tries to use the first consistent value in its list. The code for `firstoneof` (a trivial modification to that for `oneof`) appears in Table 5-23 and Table 5-24.

One might wish to have a more general `oneof` mechanism then selection from a set of constants. It would be useful to have a kind of device called, say, `choice`, with pins named `x`, `a0`, `a1`, ..., `an` (for $(n + 1)$-way multiplexing); the intent is that `x` would be connected to exactly one of the other pins. Observe, however, that this is a special case of an even more general and useful device, which we might call a `multiplexor` (by analogy with hardware) or a `case` (by analogy with software) device, with pins named `x`, `a0`, `a1`, ..., `an`; `s` must be an integer from 0 to n, and `x` is connected to the `s`'th `a` pin. Then by connecting a cell `(oneof '(0 1 2 ... n))` to the `s` pin, we get a choice box. A multiplexor is then easily constructed from `equality` and `gate` devices, and so the general `choice` construction can be simulated as in Figure 5-11. If all the `gate` devices are changed to `equality` devices, then the individual choices are additionally constrained to be distinct.

Part Two

Engineering

In aequora elucet sol
Effulgens plurimum:
Quam maxime is tentat ut
Sit mare placidum—
Absurdum quidem, quod hoc fit
Ad noctis medium.
—Lewis Carroll
Aliciae per Speculum Transitus
Translation by Clive Harcourt Carruthers (1966)

Hora coctava per protiniam teremeles
Limagiles teretant et quoque gyrirotant.
Sunt tenuiscopi macrilli: saepeque virci
Edomipali etiam vocibus eruditant.
—Lewis Carroll
"Taetriferocias"
Aliciae per Speculum Transitus
Translation by Clive Harcourt Carruthers (1966)

HOWLAND OWL: We've got to use the old *savvy*, the *know-how*, the *moxie*, the *mother-wit*, *ars celare artem*!

CHURCHY LA FEMME: You *said* it!

HOWLAND OWL: Thank you.

CHURCHY LA FEMME: [*Aside*] I'm behind him as least *one hunderd poor cent.*

SEMINOLE SAM: I'm behind him about seven miles . . . *What'd he say?*

CHURCHY LA FEMME: Who knows . . . ? It was in *Latin* an' *that* is recommendation enough for *me.*

SEMINOLE SAM: Wonder what language the *Romans* used for the old *14 karat bamboozle?*

—Walt Kelly
The Pogo Party

Sic omelet magnolia in tobasco bunion.
—Rube Goldberg

With his filibeg fair filligreed
With finest filiform.
He fleetly footed froo an' fro
The figwort in the storm.
A flaught of borealis and a
Firkin fine of fat
Was fimbricated on the fringe
Of Frelinghuysen's hat.
—Walt Kelly (1952)
I Go Pogo

Chapter Six

Efficiency

IN PREVIOUS CHAPTERS we have concentrated on developing, in as simple a manner as possible, the fundamental concepts of a constraint language and means of implementing them. The development has been linear; at each step we made incremental improvements, building on previous work. In this chapter we will undertake a complete revision of the implementation. A few new "user features" will be added, but the primary emphasis will be on implementation techniques designed to enhance the efficiency of the system.

A summary of the changes and improvements made in the new version to be presented in this chapter:

- Where possible, arrays will be used internally rather than lists. (The assumption is that an array element can be accessed in constant time by indexing, while accessing a list element takes time linear in the position of the element in the list.)
- The pins of a constraint will be considered to constitute a "frame" or "binding contour" (these words are meant only to invoke certain mental associations). The names of pins in rules will be "compiled out" and replaced by numeric indices into the frame. This allows a rule to access a pin in constant time rather than by using the `lookup` operation on entry to the rule.
- When there is a reason to awaken a device, in previous versions all the rules of that device would be awakened. Here the rules will be categorized according to pin number, and the awakening circumstances categorized as "added value", "forgotten value", or "nogood set changed". This will provide a two-dimensional access to a pre-computed bucket of applicable

rules to be awakened. Such a bucket will be potentially much smaller than the total set of rules associated with a device.

- The control structure of previous systems was based on explicit LISP function calls, with the result that the order in which things were done depended on the order in which procedures of the implementation invoked others. To reduce this explicit dependence, a task queue control structure will be used. Each task can perform some work, and in the process enqueue more tasks. A strong invariant to be enforced is that when a task completes *any* queued task may correctly be performed next. Other strong invariants can be established about the state of the constraint network data structures as of the time of choosing the next task to perform.
- Not only rules but also contradictions are treated as tasks to be queued. This allows us to make some strong claims about the state of the system when it is contradictory. In particular, we will be able to show that functions such as `disconnect` and `what` will produce meaningful results when applied while the network is in a contradictory state. This is easier to show because the relevant state is made explicit as LISP data structures, rather than having part of it implicit in the internal LISP program state.
- The introduction of the task queue discipline allows certain efficiency heuristics to be introduced by having multiple queues with various priorities. Certain kinds of rules can be given high or low priority, for example.
- In the previous versions, a rule could often be run many times because it was awakened for several different reasons. However, once it has been decided to run a rule for whatever reason, the computation performed by the rule does not depend (directly) on that reason. In this implementation, a bit will be set when a rule is enqueued for a device, and reset when the rule is run; a rule is not enqueued if its bit is already set. In this way between the time a rule is queued and the time it is run, it will not be enqueued redundantly.
- In this implementation multiple sources of support for the value of a node will be recorded. Also, the history of equatings will be recorded, so that explanations can say which equatings were involved in a computation. (This could lead eventually to automatic retraction of equatings as well as of default values, but that will not be done here.) This is all accomplished by retaining the cell/repository node structure that has been used so far, but moving some of the repository fields (contents, boundp, rule) into the cells, so that each cell can record its own value.
- At first there were `constant` cells, and then both `constant` and `default` cells. Here we will introduce a three-level hierarchy of valued cells: `constant`, `default`, and `parameter`. This will allow certain heuristics to speed up processing of nogood sets.
- Rather than having a variety of idiosyncratic notations for the various algebraic expressions for a device, a general notation will be introduced for writing any arbitrarily rooted sub-tree of a constraint network.
- Some new user facilities for manipulating the network will be introduced, such a `detach` and

`disequate`, to be explained below.

- The situation where the user re-uses a variable name for some other purpose (for example, saying `(create foo adder)` some time after saying `(variable foo)`) will be dealt with explicitly and handled cleanly.

6.1. The New Improved Language

The user interacts with the system by typing a sequence of statements. Each statement can be one of these:

- `(create` *constraint-name constraint-type*`)` creates a constraint instance. Thereafter the global name *constraint-name* represents that instance. This also implicitly brings into existence a collection of variables (pins of the constraint) named by using the `the` construct (described below).
- `(variable` *variable-name*`)` declares a global variable.
- `(destroy` *global-name*`)` causes the name *global-name* no longer to represent anything. If the name had most recently been a global variable, then that variable no longer exists, and it is as if all equatings of that variable had never been made. If the name had most recently represented a constraint instance, then it is as if that instance had never existed, and as if any equatings of its pins had never been made.
- `(==` *thing-1 thing-2*`)` equates two quantities.
- `(disequate` *thing-1 thing-2*`)` makes it as if any equating of *thing-1* to *thing-2* had never taken place. It doesn't matter whether there actually had previously been such an equating.
- `(detach` *thing*`)` makes it as if any equatings of *thing* to anything else had never been made. However, *thing* itself still exists, so this is not the same as destroying it. Also, a pin of a constraint can be detached but not destroyed.
- `(disconnect` *thing*`)` makes it as if any equatings of *thing* had never been made, but also as if the things that *thing* had been equated to had all been equated to each other. Thus if `a` had previously been equated to `b`, `c`, and `d`, and `d` had been equated to `e` and `f`, then after `(disconnect a)` the variable `a` still exists, not equated to anything; and `b`, `c`, and `d` are all mutually equated; and `d` is still equated to `e` and `f`, but `e` and `f` are not (directly) equated to `b` and `c`.
- `(dissolve` *thing*`)` makes it as if every thing equated to `thing`, directly or indirectly, had not been equated to anything. It is like detaching everything equated (directly or indirectly) to *thing*.
- `(retract` *thing*`)` causes the source of the value in *thing* to be forgotten. This is useful only if this source is a `default` or `parameter` (see below); it makes it as if the `default` or

`parameter` had been disconnected (more or less).

- (`change` *thing integer*) causes the source of the value in *thing* to be changed to *integer*. This only works if the source is a `default` or `parameter` (see below); it makes it as if the `default` or `parameter` had originally had *integer* as its value.
- (`disallow` *thing-1 thing-2 ... thing-n*) indicates that the combination of premise values on which the specified things are based is arbitrarily disallowed.[1]
- *thing* makes inquiry as to the value of the thing. The precise nature of the output is not specified here.
- (`why` *thing*) gives local information as to why *thing* does or does not have a value. The precise nature of the output is not specified here.
- (`why-ultimately` *thing*) gives global information as to why *thing* does or does not have a value. The precise nature of the output is not specified here.
- (`what` *thing*) prints part of the network as an algebraic expression in order to explain why *thing* does or does not have a value. The precise nature of the output is not specified here.

The following constructs may be used to represent a *thing*:

- *variable-name*, the name of a declared global variable. The name of a constraint instance may *not* be used—it is meaningless for this purpose.[2] A variable may not be referred to until it has been declared by a `variable` statement.
- (`constant` *integer*), which effectively means an anonymous variable with *integer* as its associated value. It is not permitted to `retract` a `constant` variable.
- (`default` *integer*), which effectively means an anonymous variable with *integer* as its associated value. A `default` variable is assumed not to change value very often (see the `change` statement above), but this affects only efficiency heuristics, not the computational behavior of the system. The value may be retracted from a `default` variable.
- (`parameter` *integer*), which effectively means an anonymous variable with *integer* as its associated value. A `parameter` variable is assumed to be likely to change its value frequently (see the `change` statement above), but this affects only efficiency heuristics, not the computational behavior of the system. The value may be retracted from a `default` variable.
- (`assume` *integer*), which is like a `parameter` variable plus an implicit constraint that causes the variable to have the value *integer* whenever that is consistently possible. If the value is retracted, this constraint may cause it to re-appear. If there are competing assumptions (for

1. This is useful for finding more than one solution for a network; after one is found, it is disallowed to force the next to be found. This strategy is reminiscent of how multiple solutions are generated in PROLOG.

2. Suppose that it were meaningful? Pursuing this thought leads to the possibility of a meta-circular constraint language, one in which constraints themselves are objects of the language which can be constrained. This is not considered in this dissertation, but the possibility is discussed in the Conclusions.

example, either of two assumptions may be asserted but not both at once), the choice of which to assert is entirely up to the system.

- `(oneof` *integer-list*), which is like a `parameter` variable plus an implicit constraint that causes the variable to have as its value one of the integers in *integer-list* (a contradiction occurs if this is not possible). If one of the integers in the list is retracted, the implicit constraint requires another to appear in its place.
- `(firstoneof` *integer-list*), which is like a `oneof` variable except that the earliest value in the list that can *consistently* be the value of the variable must actually *be* the value of the variable. Even if the variable consistently has a value, it may be retracted by the implicit constraint and a new value (occurring earlier in the list) substituted.
- `(the` *pin-name constraint-name*), which means the pin *pin-name* of the constraint *constraint-name*. The pin-names which may be used by a constraint are determined by the type of the constraint used in the `create` statement which created the constraint. A pin may not be referred to until its constraint has been declared in a `create` statement.

The constraint-types provided by the language, with their associated pin names and a short description of their purpose, are:

`adder {a, b, c}`	$c = a + b$.
`multiplier {a, b, c}`	$c = a \times b$.
`maxer {a, b, c}`	$c = \max(a, b)$.
`minner {a, b, c}`	$c = \min(a, b)$.
`equality {p, a, b}`	$p \Leftrightarrow (a = b)$, where the truth value for p is represented by 0 for *false* or 1 for *true*.
`gate {p, a, b}`	$p \Rightarrow (a = b)$.
`lesser {a, b}`	Contradiction unless $a < b$.
`lesser! {p, a, b}`	$p \Leftrightarrow \text{lesser}(a, b)$.
`lesser? {p, a, b}`	$p \Rightarrow \text{lesser}(a, b)$.
`?lesser {a, b}`	Contradiction unless $a < b$. Also, if a is known, then $a + 1$ is considered a good guess for b; and if b is known, then $b - 1$ is considered worth trying for a.
`?lesser! {p, a, b}`	$p \Leftrightarrow ?\text{lesser}(a, b)$.
`?lesser? {p, a, b}`	$p \Rightarrow ?\text{lesser}(a, b)$.
`?maxer {a, b, c}`	$c = \max(a, b)$. Also, if a is known and b is not, then a is considered a good guess for c, and similarly if b is known and not a.
`?minner {a, b, c}`	$c = \min(a, b)$. Also, if a is known and b is not, then a is considered a good guess for c, and similarly if b is known and not a.
`signum {s, a}`	$s = \text{signum}(a)$; that is, s is -1, 0, or 1 according to whether a is

negative, zero, or positive.

When a contradiction occurs, then the user may be asked to specify which of several `default` or `parameter` values to retract. The nature of this interaction is not specified here.

This description is not complete, of course, and it speaks of "values" and "contradictions" without explaining them. It is not a complete description of the language, but only a syntactic summary with some brief indications of semantics.

We would like the following property to be true of the constraint system, however: except for the ordering considerations explicitly expressed above, the order in which statements are input to the system makes no difference in the structure of the network constructed, and makes no difference in the computational results except for the cases where the system is explicitly given free choice among alternatives (as with competing `assume` constructs). A slightly different property is that from any input sequence of statements one can derive, using purely syntactic techniques, a new sequence made up only of `variable`, `create`, and `==` statements which produces a network of the same structure and containing the same computational values (again excepting free choices of the system). Moreover, all the `variable` statements can precede all the `create` statements, which in turn can precede all the `==` statements. Indeed, within each group the statements can be arbitrarily re-ordered, for example lexicographically.

6.2. The New Improved Techniques

In this section we examine the techniques and data structures to be used in the implementation. Actually, not all of them are completely new, but are derived from those used in previous versions.

6.2.1. Cells Explicitly Record Multiple Support and Equatings

The exact equating specified by the user are to be recorded in a recoverable form. To do this we record each equating explicitly (as discussed in §2.2.1 and shown in Figure 2-4 (page 42)a. Moreover, in order to be able to assign reponsibility for propagation to specified equatings without creating circular explanations, a propagation path among equated cells is maintained. However, for speed, this path is computed once when the cells are equated, but then not actually used for propagating. (It's not clear that this really buys any speed in a sequential implementation such as this, but in a parallel implementation this technique allows for fast broadcasting of a value newly arrived at a node, without sacrificing the dependency structure of the equatings.) The assumption behind this technique is that values change more frequently than equatings are done and undone.

In order that multiple support for values can be recorded, every cell can have its own value.

Cells which are pins of constraints can always record a value provided by a rule of the constraint, for example. To this end the contents, boundp, and rule components are removed from the repository structure and added to each cell.

A repository is thus now a data structure with five components:

(a) *id*, a unique symbol used mostly for sorting the nodes and for debugging purposes. The LISP value of the *id* is the repository.

(b) *cells*, a list of cells. A repository plus its associated cells make up a node.

(c) *supplier*, one of the cells. There is always a *supplier* cell, whether or not any cell of the node has a value. (This is for purposes similar to the use of an artificial supplier in the `tree-form-trace` function of Table 3-16 (page 98).)

(d) *nogoods*, a table of buckets of nogood sets, as described in §5.3.

(e) *contra*, the number of cells in the node which have their own values which do not agree with the value claimed by the *supplier*. Hence if this is non-zero the node is in a contradictory state.

A cell has ten components:

(a) *id*, a unique symbol used mostly for debugging purposes. The LISP value of the id is the cell.

(b) repository, the repository of the cell. The cell must be on the repository's cells list.

(c) *mark*, used for graph-marking algorithms.

(d) *owner*, which is either a constraint (in which case the cell is a pin of the constraint), or `()` , in which case this may be a global, `constant`, `default`, or `parameter` cell.

(e) name. If the *owner* is a constraint, then this is an integer (not a symbol as before), an index into the table of pins for the constraint's type. For a global cell, this is the LISP symbol which is the name of the cell; the value of the LISP symbol is the cell. For `constant` cells this is the symbol `constant`, and for `default` and `parameter` cells this is a generated name used for debugging.

(f) *state*, which describes whether and how the cell has a value (see below).

(g) *contents*, usually the value (if any) of the cell, but see below.

(h) *rule*, the rule by which the value was computed if the cell indeed has a value, or `()` if the cell has no value. There are three artificial rules called `*constant-rule*`, `*default-rule*`, and `*parameter-rule*`, used to distinguish `constant`, `default`, and `parameters` cells respectively.

(i) *equivs*, a list of other cells of the node (members of the *cells* list of this cell's *repository*) to which this cell has been explicitly equated by a `==` statement. Exception: if one says `(== x x)` (which is perfectly legal and meaningful, if of doubtful utility), then `x` will *not* appear on its own *equivs* list.

(j) *link*, one of the cells in the *equivs* list, or `()` . The *link* components of the cells of a node describe a valid propagation pattern within the node along explicit `==` connections.

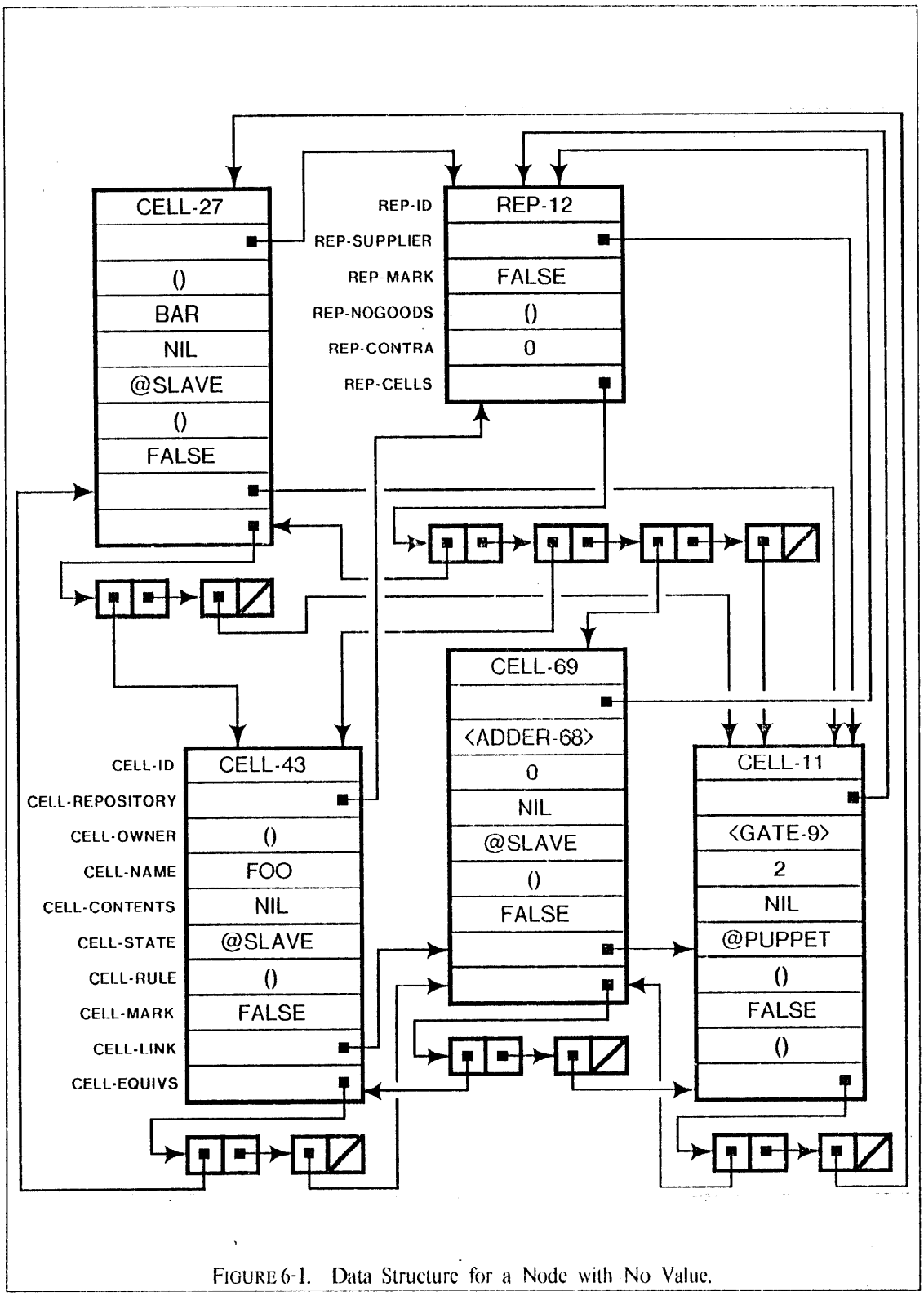


FIGURE 6-1. Data Structure for a Node with No Value.

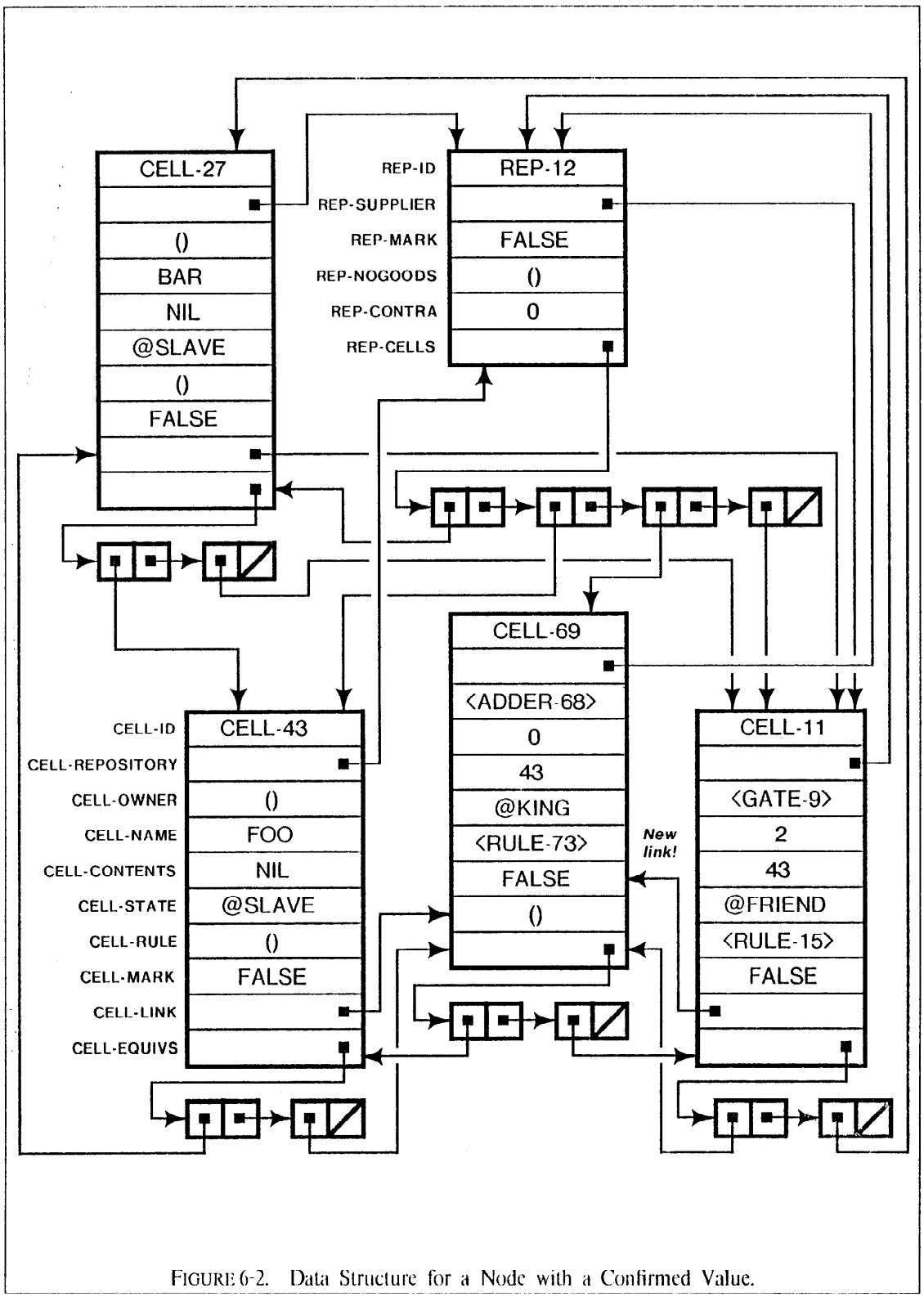


FIGURE 6-2. Data Structure for a Node with a Confirmed Value.

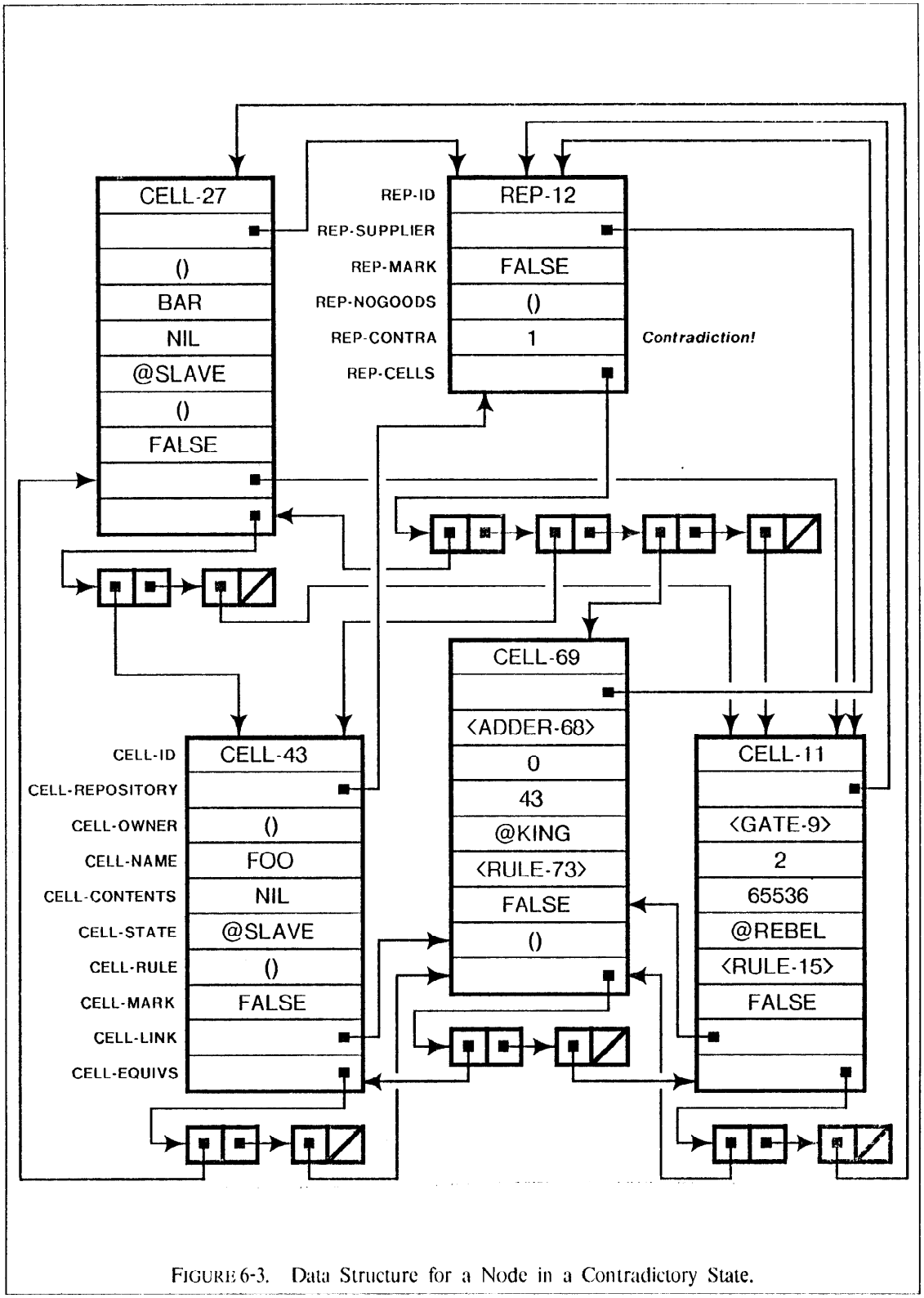


FIGURE 6-3. Data Structure for a Node in a Contradictory State.

The old boundp component of repositories in the old scheme of things has been replaced by a state component in cells. The boundp component had two states: "value" or "no value". The state component has six states, and encodes whether a cell has a value and also documents to some extent the cell's relationship to other cells. The information in the state component is partly redundant, as it encodes information obtainable from other cell components or components of other cells in the node. This redundancy sometimes enhances speed, and sometimes just permits some error checks. In any case, I believe that recording the six states explicitly aids visualization of what's going on. The six states have symbolic names; as a point of convention symbolic names here will begin with "`@`".

(1) `@king`. This cell has a value (in its contents component), and is the supplier for the node. The rule component of the cell indicates how the value was derived.

(2) `@puppet`. This cell has no value, but was arbitrarily chosen as the supplier for the node because no other cell of the node has a value either. (Every node has a supplier, and the supplier must be either a king or a puppet.) The contents and rule components are `()`.

(3) `@slave`. This cell has no value of its own. It takes on the value (if any) of the node's supplier; thus if the node's supplier is a king, then all slave cells inherit values. (If the node's supplier is a puppet, then all the other cells must be slaves, and have no values, inherited or otherwise.) The contents and rule components are `()`.

(4) `@friend`. This cell is not the node's supplier, but it has its own value, and it is the same value as that of the supplier. The contents and rule component are as for a king.

(5) `@rebel`. This cell is not the node's supplier, but it has its own value, and it is *not* the same value as that of the supplier (hence the node is in a contradictory state). The contents and rule component are as for a king. (The contra component of a repository is simply the number of rebel cells in the node. There is no special way to indicate that two rebels have the same value.)

(6) `@dupe`. This cell is in effect a slave to a rebel. It has no value of its own, but agrees with a rebel rather than with the supplier. (This situation arises only as a result of applying `==` to two nodes with differing values: one node is chosen arbitrarily to have its king and friends changed to rebels, and its slaves to dupes of the former king. The node can then be queued for contradiction processing later. Dupes tend to disappear over time.) The rule component is `()`, but the contents component is the rebel cell of which this cell is the dupe.

Thus kings, friends, and rebels have their own values; puppets, slaves, and dupes do not. Kings and puppets are suppliers; friends and slaves agree with suppliers; rebels and dupes oppose suppliers (necessarily kings). The cases of being a slave to a king and a slave to a puppet are purposely *not* distinguished in the state. This means that on encountering a slave one must check the supplier; but then again when a puppet becomes a king it is not necessary to change the states of all cells in the node. This speeds up the "good" cases of propagation without contradiction.

The link components of a node form a spanning tree for the node. The supplier of the node

must have `()` for its link, and all others must point to some other cell of the node. Consider the graph of explicit `==` connections. Then the links form a subgraph which is a strict tree. Moreover, the links indicate a direction for the edges (if cell `x`'s link is cell `y`, then the edge between `x` and `y` is indicated and directed it from `x` *to* `y`). Considering these directions, then the tree is rooted at the supplier and from any leaf following edges in the indicated direction will lead to the supplier. This property is useful for determining which equatings were responsible for a cell's getting a value. (However, the value for a dupe or slave can be found quickly just by looking at the contents of the supplier of the repository of the cell, rather than having to follow an indefinite number of link edges.)

As an example, Figure 6-1 shows a node of four cells, of which two are global variables and two are pins. None have values, so one of the pins has been arbitrarily chosen to be the puppet. Four equatings were done; the last (between the two variables) being redundant. Note that the link components converge on the puppet, which has a null link. (The figure does not show the pointers between the structures and the LISP symbols whose values are the structures; instead, the name of the LISP symbol is written.)

Figure 6-2 shows the same node after the adder has computed the value 43 for its pin, and then later the gate also computes the value 43. Because the adder happened to compute its value first, it was made king. This entailed rearranging the links (actually only one) to converge on the king; this is noted in the figure. When the gate then computed a value, its pin became a friend of the king. Note that the rule components of the pins now contain rules.

In Figure 6-3 the gate has retracted the value 43 for its pin (presumably because some premise was changed) and instead asserted the value **65536**. This makes the gate's pin a rebel. The links need not change, but the state of the gate's pin changes to `@rebel`. Note that one slave's link actually points to the rebel; this does not make it a dupe, however—it still inherits the king's value. The point is to do the minimum work necessary; there might actually be no equatings allowing a path from slave to king without going through a rebel, and yet when a rebel appears we would not want such slaves to become dupes because that would entail retracting the old value from the new dupes and their consequences and re-propagating the rebel's value—and we should be reluctant to do that because, after all, it is not clear whether the rebel's value is "correct": it may well soon disappear.

6.2.2. Constraints Use Arrays Indexed by Pin Number

The previous implementations of constraint-types, constraints, and rules used lists of things that took time to access. In particular, when a rule was invoked it was necessary to use `lookup` to find the pin-cells; a rule which used all the cells would take time quadratic in the number of pins to do this. Of course, this wasn't so bad since the particular constraint-types provided had only a few

pins, but we would like not to preclude the implementation of constraint-types with many pins.

Here we will use records (defined types) and arrays for collections of things which must be random-accessed, and lists for things that must be traversed linearly anyway. We will make `rule` be a new data type for representing rules—property lists are nice for fast prototyping, but not necessarily for fast execution.

In the new implementation a constraint-type will be a data structure with six components:

(a) *name*, a LISP symbol whioch is the name of the constraint-type. The LISP value of the symbol is the constraint-type structure.

(b) *vars*, an array of distinct symbols. These are the names of the pins. The array is zero-origin; the indices for an array of length n are the integers from 0 to n -- 1, inclusive. The position in this array of a pin-name is the number for that pin-name; thus we may speak of pin-numbers.

(c) *symbol*, a LISP symbol used to represent constraints of this type in algebraic expressions. Sometimes, but not always, this is the same as the *name*.

(d) *added-rules*, an array indexed by pin-number. Element j is a bucket (a list) of rules having pin j as a trigger. Thus, when pin j's node receives a new value, exactly these rules should be awakened.

(e) *forget-rules*, an array indexed by pin-number. Element j is a bucket of rules having pin j as an output pin. When a value is forgotten for pin j, exactly these rules should be awakened.

(f) *nogood-rules*, an array indexed by pin-number. Element j is a bucket of rules having pin j as an output pin and which might formerly have been prevented from asserting a value for the pin because of a nogood set. When the status of a nogood set involving the node containing pin j, exactly these rules should be awakened.

A constraint has five components:

(a) *name*, the global name of the constraint. The LISP value of the name is the constraint. (The *id* component has been eliminated, as steps are taken in this implementation to ensure that the global name uniquely identifies the constraint.)

(b) *ctype*, the constraint-type of which this constraint is an instance.

(c) *values*, an array indexed by pin-number. This array is of the same length as the *vars* array of the *ctype*. Element j is a cell, pin j of this instance of the constraint-type. The *owner* of pin j of a constraint is the constraint, and the *name* of pin j is the integer j.

(d) *info*, a slot used by certain constraint types to associate instance-specific information with each instance.

(e) *queued-rules*, an integer, initially 0. This is used in conjunction with the *id-bit* component of rules (see below).

A rule has six components:

(a) *code*, a LISP symbol which serves as both the name of the rule structure itself (for debugging

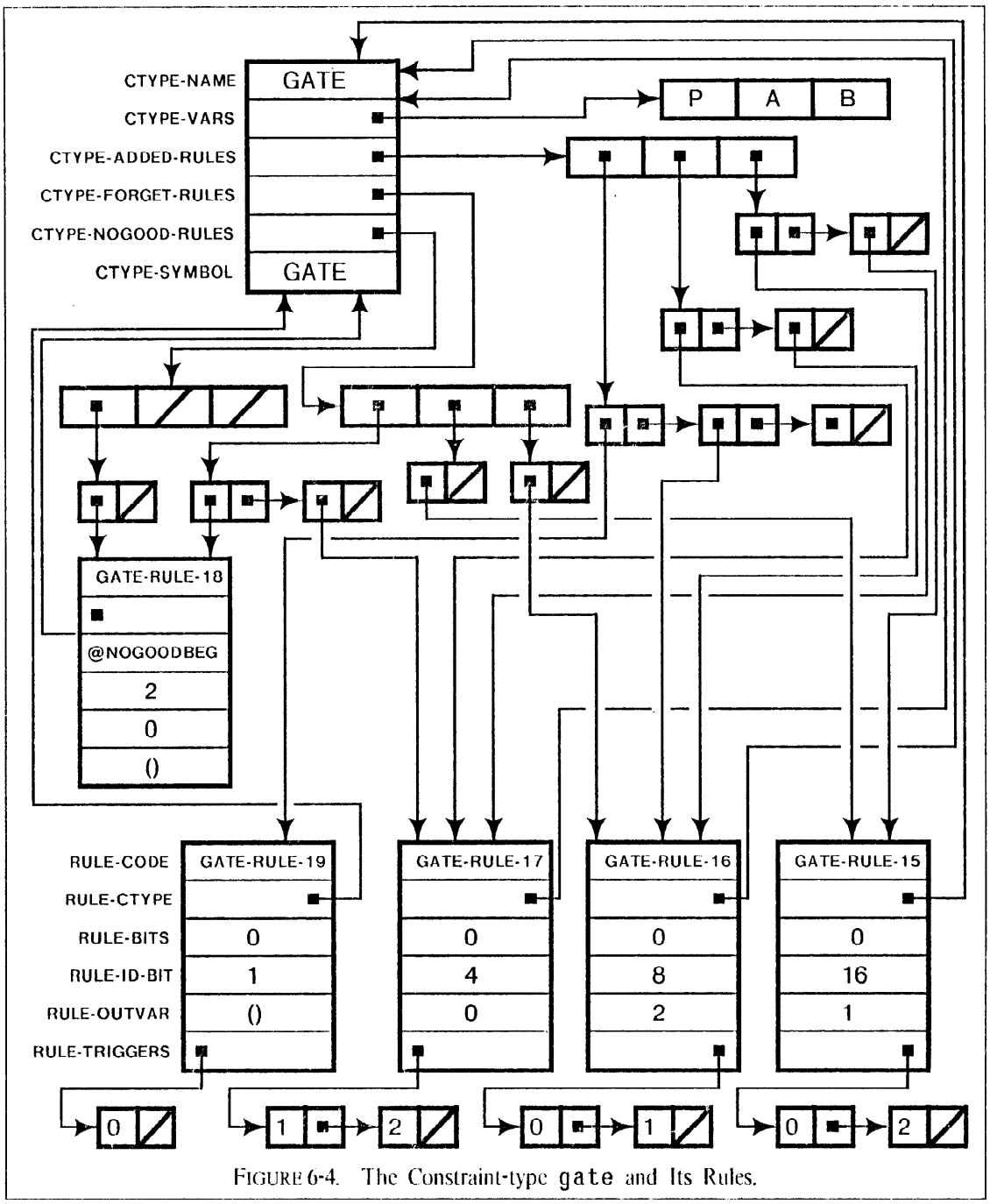


FIGURE 6-4. The Constraint-type `gate` and Its Rules.

purposes) and also the name of the LISP function which implements the rule (taking advantage of the fact that most LISP systems, including Lisp Machine LISP, allow a name to be used simultaneously as a variable name and a function name without conflict).

(b) *ctype*, the constraint-type with which the rule is associated.

(c) *triggers*, a list of pin-numbers of the pins which are the triggers for this rule.

(d) *outvar*, the pin-number of the single pin for which this rule computes a value; or `()`, indicating that the rule never computes a value for a pin.

(e) *bits*, an integer used to encode some flag bits. These flags are integers which are powers of two (set) or zero (clear), and *bits* is the sum of the flag values. (Of course, the name is suggestive of the fact that a two's-complement representation of the integers is being taken advantage of.) The two flags encoded here are called `@nogood` and `@nogoodbeg`. If either is set, then it is possible for a nogood set to prevent the rule from asserting a value; such rules should appear in buckets of the *nogood-rules* array of the *ctype*. The `@nogoodbeg` bit differs from the `@nogood` bit in that it is very meek, and will not produce a value for a node if the node has a conflicting value; a simple `@nogood` rule is willing to assert its value boldly and cause a contradiction. Therefore `@nogoodbeg` rules are not invoked unless the output pin has no value (and so must beg for a value).

(f) *id-bit*, an integer which is a power of two. All the rules associated with a given constraint-type have distinct *id-bit* components. This bit is used to identify whether a rule has been queued for processing but not yet processed. When a rule is about to be awakened on a constraint, the *queued-rules* component of the constraint is checked; if the rule's id-bit is set in the *queued-rules*, then the rule need not be queued now because that would be redundant. Otherwise, the rule/constraint pair is queued and bit set in the *queued-rules* component of the constraint. When a rule/constraint pair is dequeued, the bit is reset in the *queued-rules* component. This technique keeps the queues from being bloated and the system from wastefully running the same rule many times.

The reader may have noted that previously rules could in principle `setc` more than one pin, but here the definition of the *outvar* component implies that a rule may set at most one pin. This will make it easier to move some of the rule machinery out of the individual rules into a common processing routine.

Figure 6-4 shows the data structures for the constraint-type `gate`, whose new definition is:

```
(defprim gate (p a b)
  ((p) (if (or (= p 0) (= p 1)) @dismiss @lose))
  ((p &nogoodbeg) () (resolve-among '(0 1)))
  (p (a b) (if (= a b) @dismiss 0))
  (b (p a) (if (= p 1) a @dismiss))
  (a (p b) (if (= p 1) b @dismiss)))
```

The first rule has no output pin; the second has output pin `p`, no input pins, and should have the `@nogoodbeg` bit set. This definition format will be discussed more thoroughly below.

6.2.3. Constants Are Considered an Immutable Part of the Wiring

In this implementation it is not permitted to retract a constant. In Chapter Four it was mentioned that the ability to retract constants is easy to provide, and so one might as well. Here there are two counterarguments, one theoretical and one pragmatic—take your pick! (1) One ought to have a way of wiring essential constants into the network and have them considered part of the network structure, on a par with `==` connections, rather than alterable parameters. (2) If constants are immutable, they can be shared. We will use a hash table to record generated constant cells so that if "`(constant 43)` is typed many times only one `constant` cell is generated.

It is still useful to have two kinds of retractable valued cells, and so we call these types `default` and `parameter` cells. In Chapter Four, the distinction was drawn to guide a heuristic about which cells should be preferred for retraction. Here, the distinction will instead guide a heuristic about the formation of nogood sets.

6.2.4. A Queue-Based Control Structure Aids Efficiency Heuristics

In this implementation there are seven queues, a rather arbitrary number, to be sure. They are arranged in a simple priority order as an efficiency heuristic; but again, I emphasize that a fundamental principle of the system is that when the time comes to dequeue a task, *any* task from *any* queue may be validly chosen and executed; the ordering of the queues and the ordering within a queue affects only speed and choices explicitly reserved to the whim of the system by the language. The queues, in order from highest to lowest priority, are:

(1) `*contra-queue*`: Outstanding contradictions to be processed. Contradiction entries are of three kinds. A `@node` contradiction indicates that a node is in a contradictory state (has at least one rebel). A `@constraint` contradiction indicates that a rule explicitly signalled a contradiction. A `@resolution` contradiction indicates that a new nogood set was derived by resolving old nogood sets. (Note that these situations obtained at the time the entry was queued. By the time the entry is dequeued for processing the situation may have already been solved. This is legitimate and must be accounted for. If the network contains a contradiction, then there must be an entry for it on the queue; but not vice versa.)

Contradictions are given highest priority because there is (probably) no point in computing new values from inconsistent information. However, see the descriptions of `*defer-queue*` and `*punt-queue*` below.

(2) `*detector-queue*`: Rules which have no output pin. Such rules are called "detectors" because all they can do is detect contradictions; they compute no values. Each queue item is a pair of a rule and a constraint to apply the rule to.

Detector rules are given higher priority than ordinary rules because if on the one hand there is no contradiction, the rule might as well be run now rather than later; while if on the other hand

there is a contradiction, we would like it to be detected as quickly as possible. (This idea, and some other ideas about the ordering of the queues, is taken from [Stallman 1977].)

(3) `*vanilla-queue*`: Ordinary (plain vanilla-flavor) rules.

(4) `*nogood-queue*`: Rules with the `@nogood` or `@nogoodbeg` bit set. Such rules are likely to make assumptions, and so are accorded lower priority than vanilla rules, which are likely to be certain of their calculations.

(5) `*defer-queue*`: Contradictions which have been deferred until rules have been processed. When a contradiction occurs, the user has the option of choosing a premise to retract, *or* requesting that the contradiction processing be "deferred" (postponed until rule computations have been done) or "punted" (postponed indefinitely).

The reason for the deferral mechanism is that sometimes it is necessary to make two or more changes to the system at once—for example, to alter several parameters. The alterations together make the network consistent, but if done sequentially leave the network temporarily inconsistent. It may be desirable to postpone contradictions until all the changes have been made, whereupon many will be discovered to be "false alarms".

(6) `*rebel-queue*`: Rules some of whose triggers were rebels or dupes at the time of queuing. Entries are triples of rule, constraint, and the reason for awakening (needed for re-queuing the rule into one of the higher-priority rule queues).

The rationale here is that there is no point in computing values from contradictory information. When all outstanding contradictions have been processed (from either `*contra-queue*` or `*defer-queue*`), then there can be no more rebels, and rules are moved from `*rebel-queue*` to the other rule queues.

(7) `*punt-queue*`: Contradictions which have been postponed indefinitely. The user may be asked occasionally whether to process these, but they are not processed without explicit approval. (Of course, if they are not processed then the network may remain in a contradictory state.)

6.2.5. Generalized Algebraic Notation Can Express Any Network

The use of special notations for the different "points of view" of a constraint are eliminated. Instead of having "+" to represent addition and "—" to express its inverse; instead of using "log" and the radical sign to express inverses of exponentiation; instead of having to invent silly names like "arcmax", we will use a single symbol for each constraint type, and represent inverses by a special device.

A constraint can be notated by writing down the name of its type and what its pins are connected to. Rather than writing these pieces of information separately, in algebraic notations we use a positional convention, which is that the pins are ordered and assigned to positions spatially

relative to the position of the name of the constraint type. In FORTRAN, for example, the position to the left and right of a "+" are assigned to two pins; writing a variable in such a position means that the variables are connected to the pins: `x+y` means that `x` is connected to the first pin of the instance of +, and `y` to the second pin. (Now in FORTRAN the + device can only compute in one direction—it is not a constraint—but the use of the word "pins" is meant to be suggestive.) In LISP the name is preceded by "(" and successive positions to the right are connected to successive pins, with the last followed by a ")". In both languages another positional convention holds that the expression itself represents one pin: wherever that expression is written, the extra pin is connected to the pin for that location. In FORTRAN one writes `a*b+c`; parsing rules specify that this is equivalent to `(a*b)+c`. Now the two input pins of * are connected to two variables `a` and `b`, and the result pin to the first input of +, because the * expression was written in the position for the first input pin of +. (All of these observations appear in [Steele 1979].)

Here we will allow a constraint to be notated in a manner similar to a LISP expression. However, we will not distinguish one pin as the "output" pin, the one to be connected to the position where the expression is written. Instead, we have a device to specify which pin serves that purpose. We notate a constraint as the name of the constraint-type followed by expressions to connect to *all* the pins. However, one expression can be replaced by the symbol %, which indicates that that pin is the one that goes "out the top" to connect to the containing expression's operator.

For example, if the pins of the + constraint are called `c`, `a`, and `b` in that order, then we can write $y - x$ as `(+ y % x)` or as `(+ y x %)`. In the first case, `y` is connected to the `c` pin, `x` to the `b` pin, and the `a` pin represents the result. The expression $a - \text{arcmax}_b(c + d/e)$ can be written as `(+ a % (max b % (+ % c (* d % e))))`.

Note that this notational convention makes the order of the pin-names in a `defprim` declaration important.

As a convenient convention, if the *first* pin in an expression is to be a %, then the % may be elided. Therefore the expression `(+ % x y)` may be written as simply `(+ x y)`. This allows non-inverse forms of familar algebraic operators to be written in their usual LISP form.[3]

6.2.6. The Size of Nogood Sets Can be Heuristically Reduced

When a contradiction is discovered, either because some rule of a constraint detected it or because two distinct values collided at a node, then contradiction processing locates the premises

3. Other authors have sometimes done something similar to this. For example, it allows a functional notation for relations in predicate logic, and one for example defines $R(S[a], T[b, c])$ to be an abbreviation for $\exists x\ \exists y\ (S(a, x) \wedge T(b, c, y) \wedge R(x, y))$, where R, S, and Y are relations. I have purposely allowed elision of the *first* argument rather than the last. Suppose the relation subset(a, b) means that a is a subset of b. If the last argument is elided, then subset(a) means "that x of which a is a subset", rather than "a subset of a", which is the meaning if the first argument is elided. Similarly, one would like lessthan(x) to mean "something less than x", not "something which x is less than".

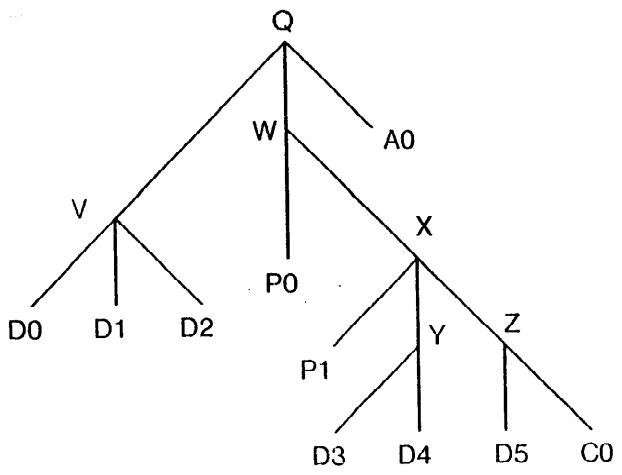


FIGURE 6-5. Summarizing Default Cells in the Network.

of the conflict. These premises collectively form a nogood set, as discussed in §5.2.1. Nogood sets can be very large and require a great deal of time to search when checking assumptions. Here three techniques are outlined for reducing the size of nogood sets, all based on distinctions among valued cells.

The first technique is simply to exclude `constant` cells from nogood sets. They are now to be regarded as a fixed part of the network structure, as permanent as `==` connections, and never to be automatically retracted (indeed, the only way to "retract" a constant now is to disconnect it). In the example of the four queens problem in 5.4.2, nogood sets were often formed containing one or two assumptions and half a dozen constants that were regarded as fixed. Every time a nogood set was checked to see whether it excluded a value, each constant of the set would be checked to ensure that it was still constant! Excluding constants from the nogood sets eliminates this overhead.

The second technique is to order the elements of the nogood set according to the expected frequency of change. Parameter cells by definition are assumed to be more variable than default cells, so if parameter cells are put at the front of nogood sets where they will be checked first, we might expect to be able to terminate check loops more quickly.

The third technique involves using the network to summarize a collection of default cells (which, again, are assumed not to change frequently). Consider the schematic diagram in Figure 6-5. The three values `V`, `W`, and `P2` caused a contradiction at `Q`. The value `V` was computed from `D0`, `D1`, and `D2`; `W` from `P0` and `Y`; and so on. The nodes `Pj` and `Dj` represent parameters and defaults, respectively; `C0` is a constant, and `A0` is an assumption. The leaves of the tree are the premises of the contradiction at `Q` .

Now in the previous versions of the constraint system, the leaves would all go into the nogood set. By the first technique listed above, we now omit the constant `C0` from the nogood set. Suppose, however, that we were to put nodes other than premises into nogood sets? This is perfectly meaningful. For the situation of Figure 6-5, we might make up a nogood set containing `V`, `W`, and `A0`. This would record the fact that the values at those three nodes are contradictory. However, this is not very useful—the constraint combining them at `Q` determined that in one step anyway, and the purpose of nogood sets is to summarize global, not local, information. Another possible nogood set would be `V`, `P0`, `P1`, `X`, `Z`, and `A0`.

The question is, which nogood sets are useful? In this implementation, nogood sets are checked only by rules which produce assumptions (i.e., `&nogood` and `&nogoodbeg` rules), *and* by the `change` statement.[4] Therefore, nogood sets must contain assumptions or cells likely to be changed (which by definition are parameters but not defaults) to be useful (this is why in previous versions `process-contradiction` did not record nogood sets unless an assumption was involved), and then for speed ought to contain as few other cells as possible.

There are many ways a nogood set (or many nogood sets) could be chosen for a contradiction. The heuristic method used here is that if any cell in the dependency tree is supported only by default cells (and constant cells, but they don't count anyway), then that cell may be used in the nogood set in lieu of the default cells, provided there are more than one (that is, the summarization is used only if it strictly decreases the size of the nogood set). Assumptions and parameter cells must be explicitly included in nogood sets. Thus for Figure 6-5 the nogood set actually chosen would be `V` (summarizing `D0`, `D1`, and `D2`), `P0`, `P1`, `Y` (summarizing `D3`and `D4`), and `D5` (which could be summarized by `Z` but is not because it wouldn't decrease the size of the nogood set).

6.2.7. Statistics Counters Measure Performance

This version of the constraint system is instrumented with a system of statistics counters (a generally useful package of LISP functions in its own right). This will allow us to count such events as number of rules queued, number of cells generated, and so on.

4. In principle, *any* computed result could be checked against existing nogood sets in `process-setc`, and perhaps this would be a good thing; but I deemed this an unnecessary complication with unclear benefits.

```
;;; Values of the STATE component of a CELL.
(defconst @king (list '@king))                ;has value, is supplier
(defconst @puppet (list '@puppet))          ;no value, is supplier
(defconst @friend (list '@friend))          ;has value, believes supplier
(defconst @slave (list '@slave))          ;no value, believes supplier
(defconst @rebel (list '@rebel))          ;has value, opposes supplier
(defconst @dupe (list '@dupe))                ;no value, believes a rebel

;;; Bits which can be set in the RULE-BITS component of a RULE.
(defconst @rule-nogood 1)
(defconst @rule-nogoodbeg 2)

;;; Values returned by the CODE function of a RULE.
(defconst @lose (list '@lose))                ;contradiction detected
(defconst @dismiss (list '@dismiss))        ;no value computed

;;; Special flags returned by CHOOSE-CULPRIT.
(defconst @defer (list '@defer))          ;contradiction should be deferred
(defconst @punt (list '@punt))                ;contradiction should be punted

;;; Reasons for enqueuing a RULE.
(defconst @added (list '@added))          ;a trigger received a new value
(defconst @forget (list '@forget))          ;the outvar is begging for a value
(defconst @nogood (list '@nogood))          ;a nogood for the outvar was flushed

;;; Reasons for contradictions in entries of *CONTRA-QUEUE* and friends.
(defconst @node (list '@node))                          ;the node contains a rebel
(defconst @constraint (list '@constraint))        ;a constraint rule detected it
(defconst @resolution (list '@resolution))        ;resolution on nogood sets
```

TABLE 6-1. Definitions of Symbolic Constants.

6.3. The New Improved Implementation

In this section the entire source code for the new system is presented, complete in itself.

6.3.1. Symbolic Constants Provide Names for Internal Marker Values

Symbols declared by the `defconst` construct are symbolic constants. They are implemented as global LISP variables, but their values are supposed not to change. As a matter of programming convention, symbolic constants of the constraint system have names beginning with "@". Some of the constants are special numeric values, but most are just LISP objects whose nature doesn't particularly matter as long as they are recognizably distinct from all other objects. For this purpose a freshly consed list of the name of the constant is used. This makes the constant recognizable when it is printed; the fresh consing ensures uniqueness. Definitions of symbolic constants used

in the constraint system appear in Table 6-1. The names `@king`, `@puppet`, `@friend`, `@slave`, `@rebel`, and `@dupe` are used to mark the state of a cell, for instance. These constants are thus effectively the elements of PASCAL-style enumerated data types.

```
(defvar *all-statistics-counters* '())

(defmacro statistics-counter (name description)
  (let ((varname (symbolconc "*" name "-STATISTICS-COUNTER*")))
    `(progn 'compile
            (or (assq ',varname *all-statistics-counters*)
                (push '(,varname ,description) *all-statistics-counters*))
            (defvar ,varname 0))))

(defmacro statistic (name)
  `(increment ,(symbolconc "*" name "-STATISTICS-COUNTER*")))

(defun stats ()
  (dolist (x (reverse *all-statistics-counters*))
    (format t "~%;~7D = ~A" (symeval (car x)) (cadr x))))

(defun reset-stats ()
  (dolist (x *all-statistics-counters*) (set (car x) 0)))
```

TABLE 6-2. Statistics Counter Mechanism.

6.3.2. Statistics Counters Make It Easy to Instrument Code

Table 6-2 defines a simple statistics-gathering mechanism. The declaration

```
(statistics-counter foo "Globbitzes frobbotzed while nurbling the scrol")
```

creates a statistics counter named `foo`. The string is a description of the meaning of the counter. This has the effect of defining a global variable named `*foo-statistics-counter*`, initializing it to zero, and adding it to a list of all declared statistics counters. One can insert into a piece of code the statement

```
(statistic foo)
```

which will cause the count in counter `foo` to be incremented. The function `stats` will print all the statistics, one per line, in the format:

```
;   2960 = Globbitzes frobbotzed while nurbling the scrol
;  45613 = Queued rules with no output pin
    ⋮
```

The function `reset-stats` resets all the counters to zero.

In the code to follow there will be many call on the `statistic` macro. Like calls to `ctrace` and `require-`*type*, they can be ignored for purposes of understanding the computation.

```
(deftype constraint-type
  (ctype-name ctype-vars ctype-added-rules ctype-forget-rules
   ctype-nogood-rules ctype-symbol)
  (format stream "<Constraint-type ~S>" (ctype-name constraint-type)))

(deftype constraint (con-name con-ctype con-values con-info (con-queued-rules 0))
  (format stream "<~S:~S>"
          (con-name constraint) (ctype-name (con-ctype constraint))))

(deftype rule ((rule-triggers '()) (rule-outvar ()) rule-code
               (rule-bits 0) (rule-ctype ()) rule-id-bit)
  (format stream
          "<~:[~4*~;~:[(~S~@[~* &NOGOOD~]~@[~* &NOGOODBEG~])←~;~S←~2*~]~]~
            ~S(~{~S~^,~})>"
          (rule-outvar rule)
          (zerop (rule-bits rule))
          (and (rule-ctype rule)
               (rule-outvar rule)
               (aref (ctype-vars (rule-ctype rule)) (rule-outvar rule)))
          (bit-test @rule-nogood (rule-bits rule))
          (bit-test @rule-nogoodbeg (rule-bits rule))
          (rule-code rule)
          (and (rule-ctype rule)
               (forlist (tr (rule-triggers rule))
                 (aref (ctype-vars (rule-ctype rule)) tr)))))
```

TABLE 6-3. Data Structures for Constraint-types, Constraints, and Rules.

6.3.3. Rules Are Data Structures and Catalogued in Arrays

Table 6-3 shows the definitions of the data structures described in §6.2.2. An additional feature of interest is the new printing formats for constraints and rules. A constraint is now uniquely named by a global variables, and so its id is not printed. Instead it just prints as the name and its type:

```
(create foo adder)
<FOO:ADDER>
```

The format for printing rules is intended to show the functional dependence of the rule by using the outvar and trigger information. The five rules for `gate` which were defined in §6.2.2 print in this way:

```
<A←GATE-RULE-15(P,B)>
<B←GATE-RULE-16(P,A)>
<P←GATE-RULE-17(A,B)>
<(P &NOGOODBEG)←GATE-RULE-18()>
<GATE-RULE-19(P)>
```

Rules 15, 16, and 17 are vanilla-flavor rules; rule 18 has the `@nogoodbeg` bit set and has no triggers; and rule 19 has no output pin, and so is a detector rule.

```
(deftype repository ((rep-cells ()) (rep-supplier ()) rep-id
                     (rep-nogoods '()) (rep-contra 0))
  (format stream "<Repository~@[ for ~{~S~^,~}~]>"
          (cell-ids repository)))

(defmacro node-cells (cell) `(rep-cells (cell-repository ,cell)))
(defmacro node-supplier (cell) `(rep-supplier (cell-repository ,cell)))
(defmacro node-mark (cell) `(cell-mark (node-supplier ,cell)))
(defmacro node-nogoods (cell) `(rep-nogoods (cell-repository ,cell)))
(defmacro node-contra (cell) `(rep-contra (cell-repository ,cell)))

(deftype cell (cell-id cell-repository cell-owner cell-name
               (cell-contents ()) (cell-state @lose) (cell-rule ())
               (cell-equivs '()) (cell-link ()) (cell-mark ()))
  (progn (format stream "<~S (~S~@[ of ~S~])"
                 (cell-id cell)
                 (if (cell-owner cell)
                     (aref (ctype-vars (con-ctype (cell-owner cell)))
                           (cell-name cell))
                     (cell-name cell))
                 (and (cell-owner cell) (con-name (cell-owner cell))))
         (select (cell-state cell)
           ((@puppet) (format stream " PUPPET>"))
           ((@slave) (format stream " SLAVE~@[ ~S~]>"
                             (select (cell-state (node-supplier cell))
                               ((@king) (node-value cell))
                               ((@puppet) ())
                               (otherwise
                                   (list 'bad-supplier
                                         (cell-state (node-supplier cell)))))))
           ((@king) (format stream "~@[~* [OPPOSED]~] KING ~S>"
                            (plusp (node-contra cell))
                            (cell-value cell)))
           ((@friend) (format stream "~@[~* [OPPOSED]~] FRIEND ~S>"
                              (plusp (node-contra cell))
                              (cell-value cell)))
           ((@rebel) (format stream " REBEL ~S AGAINST ~S>"
                             (cell-value cell)
                             (if (eq (cell-state (node-supplier cell)) @king)
                                 (node-value cell)
                                 (list 'bad-supplier
                                       (cell-state (node-supplier cell))))))
           ((@dupe) (format stream " DUPE ~S AGAINST ~S>"
                            (cell-value cell)
                            (if (eq (cell-state (node-supplier cell)) @king)
                                (node-value cell)
                                (list 'bad-supplier
                                      (cell-state (node-supplier cell))))))
           (otherwise (format stream " BAD STATE ~S>" (cell-state cell))))))

(defun cell-ids (rep)
  (require-repository rep)
  (forlist (x (rep-cells rep)) (cell-id x)))
```

TABLE 6-4. Data Structures for Repositories and Cells.

6.3.4. Cells Have Fields That Were Formerly in Repositories

The definitions for the new repository and cell data structures described in §6.2.1 are shown in Table 6-4. As before, macros named `node-cells`, `node-supplier`, `node-nogoods`, and `node-contra` are provided for accessing fields of a repository given a cell. Sometimes a graph-marking algorithm wants to mark a node and not just an individual cell, but we have moved the mark component from repositories to cells; the solution is to define `node-mark` to access the mark component of the node's supplier. (This is only one reason why a node always has a supplier.)

The printing format for cells has been updated to be more informative. Consider for example this interaction. After these statements:

```
(create foo gate)
<FOO:GATE>
(progn (variable x)
       (== x (the p foo))
       (== (the a foo) (parameter 5))
       (== (the b foo) (parameter 5))
       (variable y)
       (== y (default 6))
       (== y (the b foo)))
```

(yes, `progn` is not part of the language, but I cheated for conciseness), `foo` is a gate; its `p` is connected to `x`; its `a` has the parametric value 5; and its `b` has the parametric value 5 and is also connected to `y` which had the default value 6. This causes a contradiction of course, within which we can examine the cells:

```
;;; These are the premises that seem to be at fault:
;       <CELL-78 (DEFAULT-76) REBEL 6 AGAINST 5> == Y,
;       <CELL-72 (PARAMETER-70) [OPPOSED] KING 5> == Y.
;;; Choose one of these to retract and RETURN it.
x
<CELL-61 (X) SLAVE>
(the p foo)
<CELL-55 (P of FOO) PUPPET>
(the a foo)
<CELL-57 (A of FOO) SLAVE 5>
(the b foo)
<CELL-59 (B of FOO) SLAVE 5>
y
<CELL-75 (Y) DUPE 6 AGAINST 5>
(node-supplier (the b foo))
<CELL-72 (PARAMETER-70) [OPPOSED] KING 5>
(cell-contents y)
```

```
<CELL-78 (DEFAULT-76) REBEL 6 AGAINST 5>
```

(Note the use in the LISP code of the special form `select`, which is like the `case` statement of algebraic languages. Note too that an `otherwise` clause has been provided, to print cells which somehow have a bad state component. Similarly, the code is tolerant of a supplier other than a king or puppet (this can occur when printing a trace message). The printer is a debugging tool, and debugging tools must be fairly robust, tolerating errors in the data structures!)

```
(defun node-boundp (cell)
  (require-cell cell)
  (select (cell-state (node-supplier cell))
    ((@king) t)
    ((@puppet) ())
    (otherwise (lose "The supplier ~S has a bad state." (node-supplier cell)))))

(defun node-value (cell)
  (require-cell cell)
  (let ((s (node-supplier cell)))
    (or (eq (cell-state s) @king)
        (lose "Supplier ~S for cell ~S isn't a @KING." s cell))
    (cell-contents s)))

(defun cell-value (cell)
  (require-cell cell)
  (select! (cell-state cell)
    ((@slave) (node-value cell))
    ((@king @friend @rebel) (cell-contents cell))
    ((@puppet) (lose "Can't take value of the @PUPPET ~S." cell))
    ((@dupe)
     (let ((c (cell-contents cell)))
       (require-cell c)
       (or (and (eq (cell-repository cell) (cell-repository c))
                (eq (cell-state c) @rebel))
           (lose "Bad @DUPE indirection from ~S to ~S." cell c))
       (cell-contents c)))))

(defun node-rule (cell)
  (require-cell cell)
  (let ((s (node-supplier cell)))
    (or (eq (cell-state s) @king)
        (lose "Supplier ~S for cell ~S isn't a @KING." s cell))
    (cell-rule s)))

(defun cell-true-supplier (cell)
  (require-cell cell)
  (select! (cell-state cell)
    ((@king @rebel @friend @puppet) cell)
    ((@slave) (node-supplier cell))
    ((@dupe) (cell-contents cell))))
```

TABLE 6-5. Functions for Accessing Values of Cells and Nodes.

6.3.5. The Value of a Cell May Differ from the Value of Its Node

Because the node data structure can now tolerate contradictions in the form of rebel cells, the value of a cell is not necessarily that of the node's supplier. Table 6-5 provides some functions which are useful for manipulating values of cells and nodes. Those whose names begin with `node-` deal with the supplier of the given cell's node; those whose names begin with `cell-` deal with the given cell itself.

The function `node-boundp` is a predicate true iff the node has a value; it checks the supplier (and in the process ensures that it is a king or puppet). (It is not necessary to have a separate function `cell-boundp`. If a cell is a king or puppet, then it is the supplier, and so is bound iff the node is. If it is a friend, rebel, or dupe, then it is bound; but then there must be a king, and again is the cell bound iff the node is. Finally, a slave by definition has a value iff the node (the supplier) does.)

If the node has a value, then `node-value` will fetch the value (the contents component of the suplier cell, which must be a king). Similarly, `cell-value` will get the value of the given cell, which is the cell's own value if it is a king, friend, or rebel; the supplier's value, for a slave; or the believed-in rebel's value, for a dupe. (A puppet can never have a value.) The function `node-rule` gets the rule used to compute the node's value. (The function (actually macro) `cell-rule` obviously accesses the cell's rule component, as specified in the definition of the data type `cell`. The concept of fetching the rule that computed the cell's value is reasonable, and also ought to be called `cell-rule` by these conventions, but it turned out not to be needed, and so the naming difficulty was avoided. One can instead take the `cell-rule` of the `cell-true-supplier` of the cell.)

(The LISP special form `select!` is similar to `select` but automatically supplies an `otherwise` clause which signals a correctable LISP error if no other clause is selected. In contrast, `select` merely returns `()` if no clause is selected, by analogy with `cond`. Using `select!` makes debugging easier without having to provide explicit error checking whenever a selection statement is written.)

The function `cell-true-supplier` returns the supplier which the given cell "believes". Kings, rebels, friends, and puppets believe in themselves (puppets have no values, but they are still suppliers); slaves believe in the node's supplier; and dupes believe in some rebel they point to.

```
(statistics-counter gen-repository "Repositories generated")


(defun gen-repository ()
  (statistic gen-repository)
  (let ((r (make-repository))
        (n (gen-name 'rep)))
    (setf (rep-id r) n)
    (set n r)
    r))

(defun node-lessp (x y)
  (require-cell x)
  (require-cell y)
  (alphalessp (rep-id (cell-repository x)) (rep-id (cell-repository y))))

(statistics-counter gen-cell "Cells generated")

(defun gen-cell (name &optional (owner () ownerp))
  (and ownerp (require-constraint owner))
  (if ownerp (require-integer name) (require-symbol name))
  (statistic gen-cell)
  (let ((c (make-cell))
        (r (gen-repository))
        (n (gen-name 'cell)))
    (setf (cell-id c) n)
    (set n c)
    (setf (cell-owner c) owner)
    (setf (cell-name c) name)
    (setf (cell-repository c) r)
    (push c (rep-cells r))
    (setf (cell-state c) @puppet)
    (setf (rep-supplier r) c)
    c))
```

TABLE 6-6. Generation of Repositories and Cells.

6.3.6. A Newly Generated Cell is Its Own Puppet

The code for generating new repositories and cells in in Table 6-6. Note the two statistics counters for counting the number of repositories and cells generated.

If a new cell has no owner, then its name must be a symbol; but if it has an owner, its name must be an integer (a pin number). The initial state of any new cell is `@puppet`, and it becomes the supplier of its one-cell node.

```
(deftype hashtable ((hashtable-population 0) hashtable-array
                    (hashtable-probes 0) (hashtable-lookups 0)
                    hashtable-key-extractor hashtable-load-factor-limit)
  (format stream "<Hashtable ~S size=~D population=~D load factor=~S avg probes=~S>"
          (hashtable-key-extractor hashtable)
          (array-length (hashtable-array hashtable))
          (hashtable-population hashtable)
          (// (float (hashtable-population hashtable))
              (array-length (hashtable-array hashtable)))
          (if (zerop (hashtable-lookups hashtable)) '?
              (+ (// (float (hashtable-probes hashtable))
                     (hashtable-lookups hashtable))
                 1))))

(defun gen-hashtable (kex &optional (size 63.) (load-factor-limit 0.75))
  (let ((h (make-hashtable)))
    (setf (hashtable-array h) (array-n (- (↑ 2 (haulong size)) 1)))
    (setf (hashtable-key-extractor h) kex)
    (setf (hashtable-load-factor-limit h) load-factor-limit)
    h))
```

TABLE 6-7. Hash Table Definition and Generation.

6.3.7. Hash Tables Store and Retrieve Objects Indexed by Given Keys

We will have an application for hash tables in a moment, and so we pause here to define an implementation. This section is not a general treatise on hashing, and the code presented here is not even a particularly good (or particularly bad) hashing technique. For a more general treatment, see [Knuth 1973]. Table 6-7 shows the definition for the data type `hashtable`, which has these components:

(a) *array*, an array used to store hashed records. A record may be any object other than `()`, which is used to indicate an unused array position.

(b) *key-extractor*, which is a function which when given a record will extract the record's key, which must be an integer. (While keys must be integers, they may be *very large* integers, and so simply using the key itself as the array index is not a practical technique.)

(c) *population*, the number of occupied positions in the array. This is present purely for speed; it could be computed by scanning the array.

(d) load-factor-limit. The *load factor* of the hash array is the population divided by the total array size, i.e., the percentage of used positions. The *load-factor-limit* is a limit on this percentage; when the population becomes too large, then the array must be expanded. (Analyses such as those in [Knuth 1973] indicate that the expected time to access a hash array is roughly a function of the load factor. Hence keeping the load factor low will improve the access time.)

```
(defun hash-lookup (n hashtable)
  (prog ret ()
        (increment (hashtable-lookups hashtable))
        (let ((a (hashtable-array hashtable))
              (kex (hashtable-key-extractor hashtable)))
          (let ((s (array-length a)))
            (do ((probe (mod n s) (mod (+ probe 1) s)))
                ((null (aref a probe))
                 (return-from ret () probe))
              (increment (hashtable-probes hashtable))
              (and (equal (funcall kex (aref a probe)) n)
                   (return-from ret (aref a probe) probe)))))))

(defun hash-install (k obj hashtable)
  (let ((a (hashtable-array hashtable))
        (kex (hashtable-key-extractor hashtable)))
    (or (null (aref a k)) (lose "~D slot already filled in ~S." k hashtable))
    (aset obj a k)
    (increment (hashtable-population hashtable))
    (let ((s (array-length a)))
      (and (> (hashtable-population hashtable)
              (* s (hashtable-load-factor-limit hashtable)))
           (let ((newarray (array-n (+ (* s 2) 1))))
             (setf (hashtable-array hashtable) newarray)
             (dotimes (j s)
               (or (null (aref a j))
                   (multiple-value-bind (item slot)
                       (hash-lookup (funcall kex (aref a j)) hashtable)
                     (and item (lose "Weird hashtable bug: item ~S in ~S."
                                     item hashtable))
                     (aset (aref a j) newarray slot)))))))
    obj))
```

TABLE 6-8. Hash Table Lookup and Install Operations.

(e) *lookups*, the number of times the hashtable has been accessed. This is a statistics counter, but is not done via the standard statistics counter mechanism so that it will be per-hashtable.

(f) *probes*, another statistics counter, measuring the number of unsuccessful accesses to the *array*. This plus *lookups*, all divided by *lookups*, is the average number of accesses per lookup (a quantity one seeks to minimize in the interests of speed).

The function `gen-hashtable` takes a key-extractor function and creates a hashtable around it. The initial size defaults to 63, and the load factor to 0.75. The size is constrained to be one less than a power of two. (The LISP function `haulong`, applied to an integer x, computes $\lceil \log_2(|x| + 1) \rceil$ (it is the "length of x in bits"); thus

$$2^{\text{haulong}(x)} - 1$$

is some n which is one less than a power of two and not less than x. This is a not unreasonable length for a hashtable, using *key* (mod n) as the hashing function. The function `array-n` takes an integer and constructs a zero-origin array of that length.)

The function `hash-lookup` (Table 6-8) takes a key and a hashtable and tries to find a record with that key in the table. It returns two values. The first is the record if one was found, or `()` if none was found. The second is the index into the hash array where the record was found or the search terminated. (This returning of two values is done via the Lisp Machine LISP multiple-value mechanism. If several arguments are given to the `return` function or one of its variants, then all the arguments are collectively returned from the enclosing `prog`. If the `prog`'s function was invoked via a normal function call, then the first value returned is the functional value, and the rest are discarded. However, special forms such as `multiple-value-bind` can be used to get the other values. This technique avoids consing up and picking apart a list of results; internally all the returned values are passed "on the stack".)

The function `hash-install` takes an index supplied by `hash-lookup`, a record (which should have as key that key used to obtain the index), and a hashtable. It installs the record in the table, and if the load factor has exceeded the limit it creates a new hash array, installs it in the hastable data structure, and copies the contents of the old array into the new one by re-hashing them.

```
(progn 'compile
       (defglobal *constant-rule* (make-rule))
       (setf (rule-code *constant-rule*) 'constant-code)
       (defun constant-code (*me*) (lose "Constant rule invoked on ~S." *me*)))

(progn 'compile
       (defglobal *default-rule* (make-rule))
       (setf (rule-code *default-rule*) 'default-code)
       (defun default-code (*me*) (lose "Default rule invoked on ~S." *me*)))

(progn 'compile
       (defglobal *parameter-rule* (make-rule))
       (setf (rule-code *parameter-rule*) 'parameter-code)
       (defun parameter-code (*me*) (lose "Parameter rule invoked on ~S." *me*)))

(defun globalp (cell)
  (require-cell cell)
  (and (null (cell-owner cell)) (null (cell-rule cell))))
```

TABLE 6-9. Dummy Rules for Constant, Default, and Parameter Cells.

6.3.8. Constant, Default, and Parameter Cells Have Dummy Rules

Valued cells (those created by the `constant`, `default`, and `parameter` constructs) are distinguished by the presence of distinguished dummy rules, which are the values of the variables `*constant-rule*`, `*default-rule*`, and `*parameter-rule*`, defined in Table 6-9. For uniformity, every cell which has its own value (whether a valued cell or a pin) must have a rule. However, it is an error ever to invoke the rule of a valued cell. To guard against this possibility (as a matter of defensive programming), these dummy rules are provided wioth code components that will signal a meaningful error.

Cells for global variables, on the other hand, never have values of their own; they can be distinguished by this fact. Hence the predicate `globalp`, true iff its argument (a cell) is a global cell, merely checks that the cell has no owner and no rule.

The function `initialized-cell` creates a valued cell of specified type (here specified by which dummy rule is provided). A valued cell is initially its own king. Default and parameter cells are generated in similar ways; a name is generated, an initialized cell of that name generated with the appropriate dummy rule, the name given the cell as its value, and the cell returned.

For constants, however, a hashtable is used. The global LISP variable `*constants*` is a hashtable used for hashing all constant cells. The function `constant-value` serves as the key-extractor. To generate a constant of given value, the value is used as the lookup key for the hash-table (note the use of `multiple-value-bind` to get both the cell, if any, and the hash index). If a cell with that value is already in the table, it is returned; thus constants are "shared" among

```
(statistics-counter init-cell "Initialized cells")

(defun initialized-cell (value name rule)
  (require-integer value)
  (let ((cell (gen-cell name)))
    (setf (cell-contents cell) value)
    (setf (cell-rule cell) rule)
    (setf (cell-state cell) @king)
    cell))

(defun default (value)
  (let ((name (gen-name 'default)))
    (let ((cell (initialized-cell value name *default-rule*)))
      (set name cell)
      cell)))

(defun parameter (value)
  (let ((name (gen-name 'parameter)))
    (let ((cell (initialized-cell value name *parameter-rule*)))
      (set name cell)
      cell)))

(defun constant-value (cell) (cell-contents cell))
(defglobal *constants* (gen-hashtable 'constant-value))

(defun constant (value)
  (require-integer value)
  (multiple-value-bind (item slot) (hash-lookup value *constants*)
    (or item (hash-install slot
                           (initialized-cell value 'constant *constant-rule*)
                           *constants*))))
```

TABLE 6-10. Generation of Constant, Default, and Parameter Cells.

requests. Otherwise a new constant cell is created and installed in the hashtable. (The definition of hashtables in §6.3.7 was a little long, but see now how concisely one can be used! This is the mark of a useful data abstraction.)

The sharing of constant cells is not without peril. We must ensure that a constant cell, once created, is immutable, and particular can never be retracted. Later we will see code that checks for this explicitly.

```
(defmacro variable (name) `(progn (*destroy ',name) (setq ,name (gen-cell ',name))))

(defmacro create (name type) `(*create ',name ,type))

(defun *create (name type)
  (prog2 (*destroy name)
         (gen-constraint type name)
         (run?)))

(statistics-counter gen-constraint "Constraints generated")

(defun gen-constraint (ctype name)
  (require-constraint-type ctype)
  (statistic gen-constraint)
  (require-symbol name)
  (let ((c (make-constraint)))
    (set name c)
    (setf (con-name c) name)
    (setf (con-ctype c) ctype)
    (setf (con-values c)
          (array-of (fortimes (j (array-length (ctype-vars ctype)))
                      (gen-cell j c))))
    (doarray (bucket (ctype-forget-rules ctype))
      (dolist (rule bucket)
        (and (null (rule-triggers rule))
             (enqueue-rule rule c @forget))))
    c))
```

TABLE 6-11. Declaration of Variables and Constraints.

6.3.9. Declaration of Variables and Constraints May Require Housekeeping

The `variable` and `create` constructs are implemented as LISP macros in Table 6-11. A feature common to both is that before proceeding the the definition the function `*destroy` is called. We will see the definition of this much later; suffice it for now to note that it implements the `destroy` operation, causing any old value of the variable to be explicitly garbage-collected. The reason for this care is that the versions of the constraint system in the previous chapter were subject to a subtle (but fortunately seldom encountered) difficulty: if one were to declare a variable or constraint, then re-declare it, the old and new declarations might co-exist in a single network, causing some confusion. For example, the sequence of statements

```
(variable x)
(create foo adder)
(== x (the a foo))
(create foo adder)
(== x (the a foo))
```

would cause the variable `x` to be connected to the `a` pins of *two* adders, the old `foo` and the new `foo`! Explicit destruction of the old value avoids this. Destroying a variable or constraint first disconnects it from everything else.

When a constraint is generated, the initialization is a little more complicated than before. An array of pin cells must be created for the values component. (The LISP function `array-of` takes a list and creates a zero-origin array with the same length as the list, and initializes the array elements from the list in order.)

When the constraint has been generated, there is one final task. There may be rules of the constraint-type which have output pins and no triggers (probably, but not necessarily, they are `@nogood` or `@nogoodbeg` rules which produce a value speculatively). These rules must be awakened immediately, to beg for a value, because all their triggers are satisfied! This is done by the doubly nested loop at the end of `gen-constraint`. The awakening is done by enqueuing the relevant rules.

An important principle is that queued rules must be given a chance to run. Therefore, whenever there is a possibility that a rule (or, for that matter, a contradiction or any other task (though there are no other kinds in this implementation, I strive for generality!)) has been queued, the function `run?` must be called. This function has the responsibility for starting up the task scheduler if appropriate. Nearly all the functions which implement user statements of the constraint language end by calling the `run?` function; `*create` is one example.

```
(deftype queue (queue-name (queue-entries '()) (queue-count 0))
  (format stream "<~S ~D entr~:@P ~D enqueuing~:P>"
          (queue-name queue) (length (queue-entries queue)) (queue-count queue)))

(defglobal *all-queues* '())

(defun queue-stats ()
  (dolist (q (reverse *all-queues*)) (print q)))

(defun reset-queues ()
  (dolist (q *all-queues*)
    (setf (queue-entries q) ())
    (setf (queue-count q) 0)))

(defmacro defqueue (name)
  `(progn 'compile
          (declare (special ,name))
          (setq ,name (make-queue))
          (setf (queue-name ,name) ',name)
          (push ,name *all-queues*)
          ',name))
```

TABLE 6-12. Queue Data Structure and Definition.

6.3.10. A Queue Is Yet Another Abstract Data Structure

A queue is implemented as a data structure with a name, a list of entries, and a statistics counter. (As with hashtables, queue statistics are maintained on a per-queue basis. The counter counts the number of entries ever enqueued on the queue. This minus the length of the list of entries yields the number of entries ever dequeued.) The global LISP variable `*all-queues*` accumulates a list of all queues ever defined, and the function `queue-stats` prints the queue statistics (simply by printing the queues, inasmuch as the print function for queues prints the relevant statistics anyway).

The function `reset-queues` causes all queues to be reset; their entries are forcibly removed, and their counters reset to zero. This is a useful debugging tool when a computation blows up in the middle.

The macro `defqueue` defines a queue of a given name. It make a queue data structure, associates the name with it, and adds the queue to the list of all queues.

The operations on queues are defined in Table 6-13. The predicate `queuep` is true iff the argument queue has any entries; it is not legal to dequeue an entry unless this predicate is true. (This is not to be confused with `queue-p`, defined by the `deftype` declaration of the queue data type, which is a predicate true iff its argument is a queue!)

```
(defmacro queuep (queue) `(not (null (queue-entries ,queue))))

(defglobal *queue-trace* t)

(defun enqueue (item queue)
  (require-queue queue)
  (and *queue-trace*
       (ctrace "Enqueuing ~S onto ~S." item (queue-name queue)))
  (increment (queue-count queue))
  (push item (queue-entries queue)))

(defun dequeue (queue)
  (require-queue queue)
  (and *queue-trace*
       (ctrace "Dequeuing ~S from ~S."
               (car (queue-entries queue))
               (queue-name queue)))
  (pop (queue-entries queue)))

(defun movequeue (to from)
  (require-queue to)
  (require-queue from)
  (and *queue-trace*
       (ctrace "Moving ~S to ~S." from to))
  (setf (queue-count to) (+ (queue-count to) (length (queue-entries from))))
  (setf (queue-entries to) (append (queue-entries from) (queue-entries to)))
  (setf (queue-entries from) '()))

(defmacro fromqueue ((var queue) . body)
  `(let ((,var ()))
     (unwind-protect (prog2 (setq ,var (dequeue ,queue))
                            (progn ,@body)
                            (setq ,var ()))
                     (and ,var (enqueue ,var ,queue)))))
```

TABLE 6-13. Queue Operations.

The enqueuing and dequeuing operations provide `ctrace` output. However, because queue operations are so numerous, the trace output from queuing operations can swamp all other trace output. Therefore, a special switch `*queue-trace*` is provided to suppress tracing of queue operations while permitting other trace output.

The function `enqueue` adds an entry to a queue (incrementing the statistics counter for the queue); the function `dequeue` removes an entry and returns it. This particular implementation happens to provide LIFO (last-in, first-out) queues. This was done for no particular reason other than that it was easy. A useful experiment would be to compare different queuing methods for efficiency in running constraint systems.

The operation `movequeue` moves all the entries from one queue to another in one fell swoop; it is more efficient than separately dequeuing and enqueuing each entry. The statistics counter of the to-queue is incremented by the number of entries moved.

The macro `fromqueue` is provided for dequeuing and processing queue entries in a protected manner. The form

```
(fromqueue ( var queue) ... body ... )
```

expands into

```
(let (( var ()))
     (unwind-protect (prog2 (setq var (dequeue queue))
                            (progn ... body ... )
                            (setq var ()))
                     (and var (enqueue var queue))))
```

Now the LISP special form `unwind-protect` guarantees to execute all argument forms but the first when returning from evaluation of the first. Even if there is some kind of error, or `throw` operation, the extra argument forms are evaluated as the stack is unwound (hence the name) past that point. The `fromqueue` macro binds the specified variable, then dequeues a queue entry from the queue and assigns it to *var*. Things are a trifle unsafe during the instant between the `dequeue` and the `setq`—an asynchronous interrupt at that point could mess things up—but all is well once thf irst `setq` has taken place. If for any reason an error occurs during processing of the *body*, then the entry will be re-queued. (An important case of this is that when processing a contradiction control may be given to the user to choose a culprit. He might well just quit to the LISP top level—in which case the contradiction queue entry must not be lost.) Only if processing is successfully completed and *var* set to `()` is the re-enqueuing avoided. (The safety factor is the entire reason for the `fromqueue` macro. That is why the simpler expansion

```
(let (( var (dequeue queue)))
     (unwind-protect (prog1 (progn ... body ... )
                            (setq var ()))
                     (and var (enqueue var queue))))
```

is not used. The period of unsafety from asynchronous interrupts would extend over the setting up of the `unwind-protect` mechanism.)

```
;;; Definitions of queues, in priority order
(defqueue *contra-queue*)        ;contradictions to be processed
(defqueue *detector-queue*)       ;rules with no outvars
(defqueue *vanilla-queue*)        ;plain rules
(defqueue *nogood-queue*)         ;&NOGOOD and &NOGOODBEG rules
(defqueue *defer-queue*)         ;contradictions deferred until rules processed
(defqueue *rebel-queue*)         ;rules which depended on contradictory values
(defqueue *punt-queue*)                ;contradictions deferred indefinitely

(defglobal *run-flag* t)
(defglobal *rebel-flag* ())

(defun run? () (and *run-flag* (run!)))

(statistics-counter run "Iterations of top-level-loop queue scan")

(defun run! ()
  (do () (())                                          ;forever
    (statistic run)
    (cond ((queuep *contra-queue*)
           (fromqueue (item *contra-queue*) (run-contra item)))
          ((queuep *detector-queue*)
           (fromqueue (item *detector-queue*) (run-rule item)))
          ((queuep *vanilla-queue*)
           (fromqueue (item *vanilla-queue*) (run-rule item)))
          ((queuep *nogood-queue*)
           (fromqueue (item *nogood-queue*) (run-rule item)))
          ((queuep *defer-queue*)
           (movequeue *contra-queue* *defer-queue*))
          ((and (null *rebel-flag*) (queuep *rebel-queue*))
           (setq *rebel-flag* t)
           (do ()
               ((not (queuep *rebel-queue*)))
             (let ((item (dequeue *rebel-queue*)))
               (enqueue-rule (car item) (cadr item) (caddr item)))))
          ((and (queuep *punt-queue*) (y-or-n-p "Process punted contradictions?"))
           (movequeue *contra-queue* *punt-queue*))
          (t (return 'done)))))
```

TABLE 6-14. Constraint System Queue Definitions and Task Scheduler.

6.3.11. The Task Scheduler Simply Scans the Queues in Order

Table 6-14 declares the queues used in this implementation of the constraint system (which were described in §6.2.4). The priority order of the queues has nothing to do with the order of declaration (though they are in fact declared in order for readability); the priority order is determined by the task scheduler, the function `run!`. To provide a handle on the scheduler for debugging purposes (for example, to examine the state of the queues after entries have been queued and before they are processed), there is a switch `*run-flag*`. The function `run?` calls `run!` only if `*run-flag*` is set (which it normally is).

The task scheduler checks the first four queues in order, and whichever is first discovered to have an entry, one entry is carefully dequeued and processed. Otherwise, if `*defer-queue*` has an entry, all of the entries are moved to `*contra-queue*`for processing. Otherwise, if `*rebel-queue*` has an entry, all entries are separately dequeued and distributed to the other rule queues. (The `movequeue` function could have been used here at the cost have having three separate queues `*detector-rebel-queue*`, `*vanilla-rebel-queue*`, and `*nogood-rebel-queue*`.) Failing that, then if `*punt-queue*` has any entries, the user is asked whether they should be moved to `*contra-queue*` for processing. (The LISP function `y-or-n-p` queries the user at the terminal by printing the string, then reading a character and returning true if `y`, `Y`, `t`, `T`, space, etc. is typed, or false if `n`, `N`, rubout, etc. is typed.) If there is nothing at all to do, `run!` returns `done`.

```
(statistics-counter enqueue-rule "Rules enqueued")
(statistics-counter enqueue-added-rule "Added rules enqueued")
(statistics-counter enqueue-forget-rule "Forget rules enqueued")
(statistics-counter enqueue-nogood-rule "Nogood rules enqueued")

(defun enqueue-rule (rule con reason)
  (require-rule rule)
  (require-constraint con)
  (statistic enqueue-rule)
  (or (eq (rule-ctype rule) (con-ctype con))
      (lose "The CTYPE of ~S doesn't match that of ~S." rule con))
  (let ((queue-item (cons rule con)))
    (select! reason
      ((@added)
       (statistic enqueue-added-rule)
       (cond ((null (rule-outvar rule))
              (enqueue queue-item *detector-queue*))
             ((bit-test @rule-nogood (rule-bits rule))
              (enqueue queue-item *nogood-queue*))
             ((and (bit-test @rule-nogoodbeg (rule-bits rule))
                   (not (node-boundp (aref (con-values con) (rule-outvar rule)))))
              (enqueue queue-item *nogood-queue*))
             (t (enqueue queue-item *vanilla-queue*))))
      ((@forget)
       (statistic enqueue-forget-rule)
       (cond ((or (bit-test @rule-nogood (rule-bits rule))
                  (bit-test @rule-nogoodbeg (rule-bits rule)))
              (enqueue queue-item *nogood-queue*))
             (t (enqueue queue-item *vanilla-queue*))))
      ((@nogood)
       (statistic enqueue-nogood-rule)
       (and (not (and (bit-test @rule-nogoodbeg (rule-bits rule))
                      (node-boundp (aref (con-values con) (rule-outvar rule)))))
            (enqueue queue-item *nogood-queue*))))))
```

TABLE 6-15. Deciding in Which Queue to Enqueue a Rule.

6.3.12. The Priority of a Rule Depends on Its Properties

The function `enqueue-rule` of Table 6-15 is used to make an entry on the standard rule queues (those other than `*rebel-queue`). The function takes a rule, the constraint to apply it to, and the reason for the enqueuing. (The reason is not actually used much here, except to eliminate some case-checking. In an early version of this implementation, there was a more complicated priority structure that depended on the reason for queuing as well as the characteristics of the rule. This complex structure was simplified for the purposes of the current presentation.) The reason must be one of the symbolic constants `@added`, `@forget`, or `@nogood`.

If the rule has no output pin (this can occur only if the reason is `@added`), then it is a detector rule and is enqueued on `*detector-queue*`. If the rule has the `@nogood` or `@nogoodbeg`

bit set, then it should be enqueued on `*nogood-queue*`. A `@nogoodbeg` rule, however, should not be enqueued if its output pin has a value already (but this cannot occur if the reason is `@forget`). In all other cases the rule is enqueued on `*vanilla-queue*`.

6.3.13. Rule Definitions Explicitly Specify Output Pins

Before examining the details of how rules are run, it is appropriate to review the new format for rule definitions alluded to in §6.2.2 and the conventions for computing values. Recall as an example the definition given before for `gate`:

```
(defprim gate (p a b)
  ((p) (if (or (= p 0) (= p 1)) @dismiss @lose)).
  ((p &nogoodbeg) () (resolve-among '(0 1)))
  (p (a b) (if (= a b) @dismiss 0))
  (b (p a) (if (= p 1) a @dismiss))
  (a (p b) (if (= p 1) b @dismiss)))
```

A rule definition may have two or three elements. The last is the body, a single LISP form which computes the value. The penultimate form is a list of names of trigger pins. The first of three, if present, may be either the name of an output pin, or a list containing the name of an output pin and/or keywords. (Following the Lisp Machine LISP convention, keywords begin with "&".) Currently the only keywords are `&nogood` and `&nogoodbeg` (but the syntax allows adding new keywords later), which may not be used together and may only be used when an output pin is specified.

The rule body is executed only if all the trigger pins have values. In addition, a `&nogoodbeg` rule need not be run if its output pin already has a value. Within the rule body the names of trigger pins may be used as LISP variables to refer to the values of the pins (as before). The body should return either an integer or one of the values `@lose` or `@dismiss`, meaning "contradiction" and "no value", respectively. A detector rule is not permitted to return an integer.

```
(statistics-counter run-rule-try "Attempts to run a rule")
(statistics-counter run-rule-win "Successfully run rules")
(statistics-counter run-rule-dismiss "Rule runs which dismissed")

(defun run-rule (queue-item)
  (let ((rule (car queue-item))
        (con (cdr queue-item)))
    (require-rule rule)
    (require-constraint con)
    (or (eq (rule-ctype rule) (con-ctype con))
        (lose "The CTYPE of ~S doesn't match that of ~S." rule con))
    (statistic run-rule-try)
    (setf (con-queued-rules con) (logclr (rule-id-bit rule) (con-queued-rules con)))
    (do-named check-loop
              ((tr (rule-triggers rule) (cdr tr)))
              ((null tr)
               (ctrace "Running rule ~S on ~S." rule con)
               (statistic run-rule-win)
               (let ((result (funcall (rule-code rule) con)))
                 (select result
                   ((@lose)
                    (signal-contradiction (forlist (tr (rule-triggers rule))
                                            (aref (con-values con) (car tr)))
                                          con))
                   ((@dismiss) (statistic run-rule-dismiss))
                   (otherwise
                       (require-integer result)
                       (or (rule-outvar rule)
                           (lose "Rule ~S has no output pin but returned ~S."
                                 rule result))
                       (process-setc con rule result)))))
      (let ((trigger (aref (con-values con) (car tr))))
        (select! (cell-state trigger)
          ((@slave) (or (node-boundp trigger) (return-from check-loop)))
          ((@dupe))
          ((@king @friend @rebel)
           (or (bit-test @rule-nogood (rule-bits (cell-rule trigger)))
               (bit-test @rule-nogoodbeg (rule-bits (cell-rule trigger)))
               (return-from check-loop)))
          ((@puppet) (return-from check-loop)))))))
```

TABLE 6-16. Applying a Rule to a Constraint.

6.3.14. The Triggers of a Rule Must Have Values When It Is Run

The function `run-rule` (Table 6-16) takes a queue entry (containing a rule and a constraint) from a rule queue and runs the rule on the constraint if appropriate. First the bit in the constraints queued-rules component corresponding to the rule's id-bit is reset, to indicate that the rule is no longer on the queue for that constraint. (The LISP function `(logclr x y)` performs `(logand (lognot x) y)`—it clears bits of `y` where `x` has one-bits.) Next all the triggers of the rule are checked. If a trigger is a slave, then its supplier must have a value. If the trigger is a dupe, it necessarily has a value. On the other hand, a puppet never has a value.

The test in the other three cases may seem a trifle strange: the trigger passes the test only if the rule that supplied the value is a `&nogood` or `&nogoodbeg` rule. The key is to realize that if the trigger is a king, friend, or rebel, then the value must have been computed by the constraint we are considering applying a rule for. Now, there is a convention in this constraint system that no ordinary rule ever awakens another rule for the same constraint; the same effect can be achieved by letting the second rule's triggers include those of the first rule and duplicating the first rule's computation. Such duplication is seldom necessary in practice, and the effort saved by not awakening rules is considerable.

If the triggers pass the test, then the rule code (a LISP function) is applied to the constraint. If the result is `@lose`, a contradiction is signalled via the function `signal-contradiction`. If it is `@dismiss`, then a statistic is tallied and nothing else occurs. Otherwise, the result must be an integer to be installed as the output pin's value (there must be an output pin) via the function `process-setc`.

```
(defun process-setc (con rule value)
  (require-constraint con)
  (require-rule rule)
  (require-integer value)
  (let ((cell (aref (con-values con) (rule-outvar rule)))
        (sources (forlist (tr (rule-triggers rule))
                   (aref (ctype-vars (con-ctype con)) tr))))
    (ctrace "~S computed ~S for its pin ~S~
                        ~:[~2*~; from pin~P ~{~S~^, ~}~]."
            con
            value
            (aref (ctype-vars (con-ctype con)) (cell-name cell))
            sources
            (length sources)
            sources)
    (process-setc-work con rule value cell)))

(statistics-counter process-setc-override "Rules which overrode other rules")
(statistics-counter process-setc-supersede "Rules which superseded other rules")
```

TABLE 6-17. Installing a Computed Value in a Pin (i).

6.3.15. Installing a Value in a Pin Changes the Pin's Cell-state

The function `process-setc` (Table 6-17) does some error-checking, prints a trace message, and then hands off the real work to `process-setc-work`.

The function `process-setc-work` (Table 6-18) does the error checks all over again (for robustness, never trust *any* code on another page). (Note: when `con` is `()` then the given cell is (rather, *was*) a `default` or `parameter` cell, and may not be a king, friend, rebel, or dupe. This is used by the `change` function for altering defaults and parameters.) There are then several cases, depending on the current cell-state of the pin.

If the output pin is a king, friend, or rebel, then some other rule of the same constraint has already computed a value for the pin. If the new value is not the same as the current value, then it is a hard error (rules of the same constraint shouldn't conflict), *unless* the rule which computed the old value was a `&nogood` or `&nogoodbeg` rule, in which case the new value may *override* the old one: the old one is forcibly forgotten, and then the processing restarted (by simply calling `process-setc-work` tail-recursively). If the new value is the same as the old value, then nothing need be done, but as a peculiar heuristic the new rule *supersedes* the old one as the justification if the new rule's triggers are a subset of the old one's triggers—this makes the value less likely to be forgotten if an unnecessary trigger is forgotten. However, it is *not* correct to do this merely because the size of the new rule's trigger set is smaller than that of the old rule: that might introduce circular dependency structures. The new trigger set must be a subset of the old.

```
(defun process-setc-work (con rule value cell)
  (and con (require-constraint con))
  (require-rule rule)
  (require-integer value)
  (require-cell cell)
  (select! (cell-state cell)
    ((@king @friend @rebel)
     (or con (lose "No constraint in PROCESS-SETC-WORK?"))
     (cond ((not (equal (cell-contents cell) value))
            (cond ((bit-test @rule-nogoodbeg (rule-bits (cell-rule cell)))
                   (ctrace "Rule ~S overrides value ~S of rule ~S with ~S."
                           rule (cell-contents cell) (cell-rule cell) value)
                   (statistic process-setc-override)
                   (forget cell)
                   (process-setc-work con rule value cell))
                  (t (lose "Rules ~S and ~S of ~S disagreed on value for pin ~S ~
                            (respective values were ~S and ~S)."
                           (cell-rule cell)
                           rule
                           con
                           (aref (ctype-vars (con-ctype con)) (cell-name cell))
                           (cell-contents cell)
                           value))))
           ((contains (rule-triggers (cell-rule cell)) (rule-triggers rule))
            (statistic process-setc-supersede)
            (setf (cell-rule cell) rule))))                ;bogus heuristic
    ((@puppet)
     (setf (cell-contents cell) value)
     (setf (cell-rule cell) rule)
     (setf (cell-state cell) @king)
     (awaken-all (node-cells cell) @added cell))
    ((@slave @dupe)
     (setf (cell-contents cell) value)
     (setf (cell-rule cell) rule)
     (cond ((node-boundp cell)
            (cond ((equal value (node-value cell))
                   (setf (cell-state cell) @friend))
                  (t (setf (cell-state cell) @rebel)
                     (increment (node-contra cell))
                     (note-rule-contradiction con rule cell))))
           ((eq (cell-state cell) @dupe)
            (lose "~S was a @DUPE in a valueless node." cell))
           (t (usurper cell)
              (setf (cell-state cell) @king)
              (awaken-all (node-cells cell) @added cell))))))
```

TABLE 6-18. Installing a Computed Value in a Pin (ii).

If the output pin is a puppet, then it can simply be made king, the value installed, and all the cells of the node awakened (except the output pin itself—this suppression is accomplished by the third argument to `awaken-all`).

If the output pin is a slave or a dupe, then if the node's supplier has a value (it must if the pin is a dupe!), the output pin becomes a friend or a rebel depending on whether or not the new value

agrees with the king's. If it becomes a rebel, the contra count of the node is incremented, and the contradiction created is noted via `note-rule-contradiction`.

If the output pin is a slave and the node is supplied by a puppet, then the output pin usurps the puppet's throne, makes itself king, and awakens all the cells excepting itself. (Usurpation causes the specified cell to become the supplier of the node.)

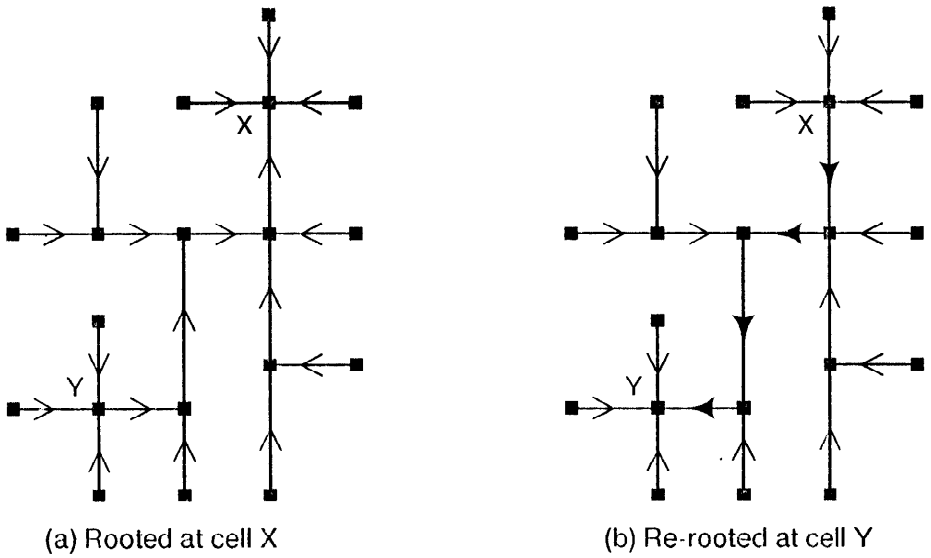


FIGURE 6-6. Usurping a Supplier.

```
(statistics-counter usurper "Usurpations")

(defun usurper (cell)
  (require-cell cell)
  (statistic usurper)
  (let ((s (node-supplier cell)))
    (point-links-toward cell)
    (let ((sc (cell-state cell))
          (sx (cell-state s)))
      (setf (cell-state s) sc)
      (setf (cell-state cell) sx))))

(defun point-links-toward (cell)
  (require-cell cell)
  (do ((x cell (prog1 (cell-link x) (setf (cell-link x) y)))
       (y () x))
      ((eq x (node-supplier cell))
       (setf (cell-link x) y)
       (setf (node-supplier cell) cell))))
```

TABLE 6-19. Usurping the Throne of the Supplier of a Node.

6.3.16. Usurping a Supplier Simply Reverses Links from Usurper to Supplier

The operation of usurpation takes a cell and causes that cell to be the supplier of its node. The primary task here is rearrangement of the link components so that all link paths lead to the new supplier. This is easy. Consider the path along link edges between the proposed usurper and the current supplier. If those edges are simply reversed in direction, then the desired result is produced. An example of this appears in Figure 6-6, wheih depicts the link edges connecting the cells of a node. In Figure 6-6a the cell `X` is the supplier. In Figure 6-6b the cell `Y` has usurped `X`, and the links along the path from `Y` to `X` have been reversed (indicated by heavy arrowheads).

The function `point-links-toward` (Table 6-19) accomplishes this task. (In structure `point-links-toward` is similar to the LISP function `nreverse`, which destructively reverses a list.) At each step `x` is a cell along the path from usurper to supplier, and `y` trails one step behind; on each iteration, and at the end, one link edge is reversed. The function `usurper` calls `point-links-toward` and also then exchanges the cell-states of the usurper and old supplier, as a convenience (often this does the right thing, as when a slave usurps a puppet).

```
(defun signal-contradiction (cells con)
  (require-constraint con)
  (ctrace "Contradiction in ~S~@[ among these pins: ~:{~S=~S~:^, ~}~]."
          con
          (forlist (cell cells)
            (require-cell cell)
            (list (aref (ctype-vars (con-ctype con)) (cell-name cell))
                  (cell-value cell))))
  (enqueue (list* @constraint
                  con
                  (forlist (cell cells)
                    (require-cell cell)
                    (cons cell (cell-value cell))))
           *contra-queue*))

(defun note-rule-contradiction (con rule cell)
  (and con (require-constraint con))
  (require-rule rule)
  (require-cell cell)
  (or (and (eq (cell-state cell) @rebel)
           (eq (cell-state (node-supplier cell)) @king))
      (lose "~S doesn't conflict with ~S after all!" cell (node-supplier cell)))
  (and con
       (let ((triggers (forlist (tr (rule-triggers rule))
                                (aref (con-values con) tr))))
         (ctrace "Contradiction in ~S~@[ among these pins: ~:{~S=~S~:^, ~}~];~
             ~%;/|   it calculated ~S for ~S from the others by rule ~S."
                 con
                 (cons (list (aref (ctype-vars (con-ctype con)) (cell-name cell))
                             (node-value cell))
                       (forlist (c triggers)
                         (require-cell c)
                         (list (aref (ctype-vars (con-ctype con)) (cell-name c))
                               (cell-value c))))
                 (cell-contents cell)
                 (aref (ctype-vars (con-ctype con)) (cell-name cell))
                 rule)))
  (enqueue (list @node cell (node-supplier cell))
           *contra-queue*))

(defun disallow (&rest cells)
  (dolist (c cells) (require-cell c))
  (let ((prems (fast-premises* cells)))
    (enqueue (cons @resolution (forlist (p prems) (cons p (cell-value p))))
             *contra-queue*)
    (run?)))
```

TABLE 6-20. Signalling Contradictions.

6.3.17. Signalling a Contradiction Merely Queues a Contradiction Task

The functions `signal-contradiction` (used by the function `run-rule` in Table 6-16 (page 226)) and `note-rule-contradiction` (used by the function `process-setc-work` in Table 6-18 (page 229)) each signal a contradiction by enqueuing a task to process it later. Apparently the only difference between them is the error checks they perform and the trace output emitted; however, they enqueue slightly different kinds of tasks. The function `signal-contradiction` (Table 6-20) is called when some rule returned `@lose` to indicate that a contradiction was detected without returning a value. In this case the contradiction is blamed on the constraint, and a `@constraint` contradiction task is enqueued. The queue entry contains the constraint which detected the contradiction, and an association list of pins with values, indicating the trigger values that caused the contradiction. This information must be saved because by the time the contradiction task is dequeued for processing the pins may have different values, but the contradiction is based on those particular values. If the pins no longer have those values, then the contradiction described by the queue entry is no longer in effect.

On the other hand, `note-rule-contradiction` enqueues a `@node` contradiction task. The queue entry contains two cells of the same node which are in conflict, one being the supplier (at the time the task is enqueued) and the other a rebel. If when the task is processed the cells no longer conflict, then the contradiction is no longer in effect.

The user function `disallow` exemplifies the third kind of contradiction task, of type `@resolution`. The queue entry contains an association list of cells and values as for a `@constraint` task, but mentions no constraint. The cells have no local association, but have been determined from global considerations to be contradictory when they have those values. Usually such a collection of cells is obtained by resolution of nogood sets, but here `disallow` allows the user to specify an arbitrary contradictory set of cells. The collective premises of the cells supplied by the user are tracked down and declared contradictory. As with all user interface functions which enqueue tasks, `disallow` finishes by calling `run?` to enable task scheduling if appropriate.

```
(statistics-counter run-contra "Contradictions dequeued for processing")
(statistics-counter run-contra-node "@NODE contradictions dequeued for processing")
(statistics-counter run-contra-constraint
                    "@CONSTRAINT contradictions dequeued for processing")
(statistics-counter run-contra-resolution
                    "@RESOLUTION contradictions dequeued for processing")

(defun run-contra (queue-item)
  (statistic run-contra)
  (setq *rebel-flag* ())
  (select! (car queue-item)
    ((@node)
     (statistic run-contra-node)
     (let ((c1 (cadr queue-item))
           (c2 (caddr queue-item)))
       (require-cell c1)
       (require-cell c2)
       (or (null (cdddr queue-item))
           (lose "Bad @NODE contradiction queue item ~S." queue-item))
       (or (not (eq (cell-repository c1) (cell-repository c1)))
           (not (eq (cell-true-supplier c1) c1))
           (not (eq (cell-true-supplier c2) c2))
           (and (node-boundp c1) (equal (cell-contents c1) (cell-contents c2)))
           (process-contradiction queue-item (cdr queue-item)))))
    ((@constraint)
     (statistic run-contra-constraint)
     (let ((con (cadr queue-item))
           (alist (cddr queue-item)))
       (require-constraint con)
       (do ((a alist (cdr a)))
           ((null a)
            (process-contradiction queue-item (forlist (a alist) (car a)) con))
         (let ((cell (caar a))
               (val (cdar a)))
           (require-cell cell)
           (require-integer val)
           (or (and (node-boundp cell) (equal (cell-contents cell) val))
               (return))))))
    ((@resolution)
     (statistic run-contra-resolution)
     (let ((alist (cdr queue-item)))
       (do ((a alist (cdr a)))
           ((null a) (process-contradiction queue-item (forlist (a alist) (car a))))
         (let ((cell (caar a))
               (val (cdar a)))
           (require-cell cell)
           (require-integer val)
           (cond ((or (not (node-boundp cell))
                      (not (equal (cell-contents cell) val)))
                  (install-nogood-set
                      (forlist (a alist) (cons (cell-repository (car a)) (cdr a))))
                  (return)))))))))
```

TABLE 6-21. Running a Contradiction Task.

6.3.18. Contradictions Must Still Hold at the Time of Processing

The purpose of the function `run-contra` (Table 6-21) is to verify that a dequeued contradiction task still describes a contradiction. If the contradiction is still in the network, it is handed off to `process-contradiction`; if not, then the task is dismissed and forgotten.

There are three kinds of contradiction (`@node`, `@constraint`, and `@resolution`), and so three cases in the code for `run-contra`.

For a `@node` task, the queue entry contains exactly two cells, which at the time the task was queued were cells of the same node asserting different values. The contradiction no longer holds if they no longer share a repository (and so are no longer of the same node—they may have been disconnected!); if either is not its own true supplier (if one is now a slave or dupe, then another contradiction will have been enqueued involving the new king or rebel, so this one need not be processed); if either has no value; or if their values agree.

For a `@constraint` task, the queue item has a constraint and an association list pairing pins of the constraint with values that triggered a contradiction. The contradiction still holds only if all the cells still have values matching the paired values.

For a `@resolution` task, the queue item has just an association list pairing cells with values that triggered a contradiction. The contradiction still holds only if all the cells still have values matching the paired values. If the contradiction does not now hold, however, it is nevertheless important that a nogood set be installed.

```
(defmacro mark-cell (cell val) `(setf (cell-mark ,cell) ,val))
(defmacro unmark-cell (cell) `(setf (cell-mark ,cell) ()))
(defmacro cell-markp (cell) `(cell-mark ,cell))

(declare (special *defaults* *parameters* *nogoods* *default-trees* *links*))

(defun fast-premises (cell)
  (require-cell cell)
  (prog ((*defaults* '())
         (*parameters* '())
         (*nogoods* '())
         (*default-trees* '())
         (*links* '()))
        (let ((flag (fast-premises-mark cell)))
          (select flag
            ((@lose @dismiss))
            (otherwise (push (if (null (cdr flag)) (car flag) cell)
                             *default-trees*))))
        (fast-premises-unmark cell)
        (return (append *defaults* *parameters* *nogoods*)
                *defaults* *parameters* *nogoods* *default-trees* *links*)))

(defun fast-premises* (cells)
  (prog ((*defaults* '())
         (*parameters* '())
         (*nogoods* '())
         (*default-trees* '())
         (*links* '()))
        (let ((flag (fast-premises-mark* cells)))
          (select flag
            ((@lose @dismiss))
            (otherwise (setq *default-trees*
                             (if (< (length flag) (length cells)) flag cells)))))
        (fast-premises-unmark* cells)
        (return (append *defaults* *parameters* *nogoods*)
                *defaults* *parameters* *nogoods* *default-trees* *links*)))
```

TABLE 6-22. Fast Computation of Premises and Related Quantities.

6.3.19. Computation of Premises Also Determines Summarizations of Defaults

Before we consider the processing of contradictions, it is appropriate to discuss the tracing of premises and the determination of an appropriate summarization (as described in §6.2.6). The function `fast-premises` in Table 6-22 computes not only the list of premises, but also separate lists of parameter, default, and "nogood" (assumption) premises, a list of summarizations of the defaults (called the "default-trees" because each summarization is the root of a tree whose leaves are defaults), and the set of links between cells traversed at each node (this is the set of equatings along which the computation traveled). These quantities are accumulated in the LISP special variables bound to empty lists in the `prog`. The list of premises is simply the concatenation of the lists of defaults, parameters, and assumptions. After `fast-premises-mark` and `fast-premises-unmark` are called, *all six lists* are returned as values, using the Lisp Machine LISP multiple-value convention. If `fast-premises` is called as a simple LISP function, the list of premises is the result (as before), and the other five lists are discarded. All the lists can be obtained by using the Lisp Machine LISP `multiple-value-bind` construct.

The function `fast-premises*` performs the same operation on a list of cells.

The macros `mark-cell`, `unmark-cell`, and `cell-markp` are operations on the mark component of a cell. Note that `mark-cell` takes an extra argument which is the mark value (thus it implements not a mark *bit*, but a mark quantity.

```
(defun fast-premises-mark (cell)
  (require-cell cell)
  (and (node-boundp cell)
       (let ((s (cell-true-supplier cell)))
         (cond ((cell-markp s)
                (and (eq (cell-mark s) t) (lose "Circular dependency at ~S." cell))
                (cell-mark s))
               (t (mark-cell s t)                          ;for error checking!
                  (fast-premises-mark-links cell s)
                  (let ((result (fast-premises-mark-test s)))
                    (mark-cell s result)
                    result))))))

(defun fast-premises-mark-links (x y)
  (prog foo (links1 links2)
        (do ((c x (cell-link c)))
            ((null (cell-link c)))
          (cond ((eq c y)
                 (setq *links* (nconc links1 *links*))
                 (return-from foo))                                ;fast escape
                (t (push (cons c (cell-link c)) links1))))
        (do ((c y (cell-link c)))
            ((null (cell-link c)))
          (cond ((eq c x)
                 (setq *links* (nconc links2 *links*))
                 (return-from foo))                                ;fast escape
                (t (push (cons c (cell-link c)) links2))))
        (setq *links* (nconc links1 links2 *links*))))

(defun fast-premises-mark-test (s)
  (cond ((eq (cell-rule s) *default-rule*)
         (push s *defaults*)
         (list s))
        ((eq (cell-rule s) *parameter-rule*)
         (push s *parameters*)
         @lose)
        ((eq (cell-rule s) *constant-rule*) @dismiss)
        ((or (bit-test @rule-nogood (rule-bits (cell-rule s)))
             (bit-test @rule-nogoodbeg (rule-bits (cell-rule s))))
         (push s *nogoods*)
         @lose)
        (t (fast-premises-mark*
             (forlist (tr (rule-triggers (cell-rule s)))
               (aref (con-values (cell-owner s)) tr))))))
```

TABLE 6-23. Gathering Premise and Link Information.

The function `fast-premises-mark` (Table 6-23) is a good deal more complex than before. All the information is accumulated in the global variables bound in `fast-premises`, and the functional value of `fast-premises-mark` is used as a flag. The symbolic constants `@dismiss` and `@lose` are abusively pressed into service here. If the returned value is `@dismiss` then no default, parameter, or assumption cells were encountered in the subtree depending from the argument cell. If the returned value is `@lose` then a parameter or assumption cell was seen somewhere. Otherwise the returned value is a list of all the default cells found in the subtree.

When a cell is given to `fast-premises-mark`, its true-supplier is taken. If it is unmarked, then the mark is first set to `t`. The operation `fast-premises-mark-links` accumulates the link information between the cell and the true-supplier, and then `fast-premises-mark-test` figures out an appropriate return value. This value is then stored in the mark for the true-supplier. If this cell is ever encountered again during the tracing of premises, the contents of the mark component is returned immediately. An important point is that the value `t` can never be seen in a mark cell—that value is put in only for an error check! If it is seen, then the dependency structure must be circular (because there is a `t` in cells only along the path from the root of the tree being searched to the cell currently being considered).

The function `fast-premises-mark-links` follows the links from the cell and its true-supplier, pushing pairs of cells representing equatings onto `*links*`. Normally the supplier will be the node's supplier, and so the first `do` loop will get them all. However, if the cell is a dupe and the true-supplier a rebel, then there are three cases:

(1) Following links from the dupe leads to the rebel.

(2) Following links from the rebel leads to the dupe.

(3) Following links from either leads to the node's supplier before reaching the other.

All of these cases have to be dealt with properly.

```
(defun fast-premises-mark* (cells)
  (let ((trees '())
        (defaults ())
        (state @dismiss))
    (dolist (c cells)
      (let ((result (fast-premises-mark c)))
        (select result
          ((@lose) (setq state @lose))
          ((@dismiss))
          (otherwise
              (push c trees)
              (select state
                ((@lose))
                ((@dismiss) (setq state t) (setq defaults result))
                (otherwise (setq defaults (unionq result defaults))))))))
    (and (eq state @lose)
         (setq *default-trees*
               (nconc (if (< (length trees) (length defaults)) trees defaults)
                      *default-trees*)))
    (if (eq state t) defaults state)))

(defun fast-premises-unmark (cell)
  (require-cell cell)
  (let ((s (cell-true-supplier cell)))
    (cond ((cell-markp s)
           (unmark-cell s)
           (fast-premises-unmark*
               (forlist (tr (rule-triggers (cell-rule s)))
                 (aref (con-values (cell-owner s)) tr)))))))

(defun fast-premises-unmark* (cells)
  (dolist (cell cells) (fast-premises-unmark cell)))
```

TABLE 6-24. Tracing Premises for a List of Cells, and Unmarking.

The function `fast-premises-mark-test` handles the various cases and determines the value to be returned as described above. If the supplier is not interesting, then the triggers for the rule that computed its value are recursively traced using fast-premises-mark* (Table 6-24). This function traces each of the given cells, and combines the results. If any of them contains something other than a default cell (the value `@lose` was returned), then `@lose` must be returned from this level, and the subtrees appropriately summarized and added to `*default-trees*`. At *each* recursive level of call to `fast-premises-mark*` a heuristic summarization can be done. Note that the set-union operation `unionq` is used rather than `append` because subtress may be shared, and some of the results may have been obtained from cached lists in the cell marks. All this serves to reduce the size of nogood sets as much as possible.

The functions `fast-premises-mark` and `fast-premises-mark*`, as before, run around and reset all the cell marks.

```
(statistics-counter process-contra "Contradictions actually processed")
(statistics-counter process-contra-auto "Nogood culprits automatically chosen")

(defun process-contradiction (queue-item cells &optional (con () conp))
  (and conp (require-constraint con))
  (statistic process-contra)
  (multiple-value-bind (premises defaults params nogoods trees links)
      (fast-premises* cells)
    (cond ((not (null nogoods))
           (ctrace "Deeming ~S in ~S (computed by rule ~S) to be the culprit."
                   (cell-value (car nogoods))
                   (cell-id (car nogoods))
                   (cell-rule (car nogoods)))
           (statistic process-contra-auto)
           (form-nogood-set
               (append nogoods params trees))
           (*retract (car nogoods)))
          ((null premises) (lose "Hard-core contradiction!"))
          ((null (cdr premises))
           (and params (form-nogood-set (append params trees)))
           (*retract (car premises)))
          (t (and params (form-nogood-set (append params trees)))
             (let ((choice (choose-culprit premises)))
               (select choice
                 ((@defer) (enqueue queue-item *defer-queue*))
                 ((@punt) (enqueue queue-item *punt-queue*))
                 (otherwise (require-cell choice) (*retract choice))))))))
```

TABLE 6-25. Processing of Contradictions.

6.3.20. Contradiction Processing Traces Premises and Chooses a Culprit

Now that `fast-premises` thoughtfully divides the premises into groups and returns them, the task of `process-contradiction` (Table 6-25) is easier. It calles `fast-premises` using `multiple-value-bind` to get the six return values, and then makes some simple tests. If there are any assumptions, a nogood set is formed from the assumptions, the nogoods, and the summarizations of defaults, and then the first assumptions is arbitrarily chosen for retraction. If there are no premises at all, it is a hard contradiction. If there is one premise, it is chosen by default for retraction, and a nogood set is formed if there are any parameter cells among the premises. Otherwise, `choose-culprit` is called to select a culprit (and, as in the previous case, a nogood set is formed if any parameters are involved). If `choose-culprit` returns `@defer` or `@punt` rather than a culprit, then the contradiction is re-queued for later processing.

```
(defun form-nogood-set (cells)
  (setq cells (sort (append cells '()) #'node-lessp))
  (ctrace "The set~:{~<~%;|~8X~:15,72; ~S=~S~>~:↑,~}~<~%;|~8X~:15,72; is no good.~>"
          (forlist (c cells) (list (cell-goodname c) (cell-value c))))
  (install-nogood-set
      (forlist (c cells) (cons (cell-repository c) (cell-value c)))))

(statistics-counter nogood-set "Nogood sets installed")

(defun install-nogood-set (alist)
  (statistic nogood-set)
  (let ((nogood (cons 'nogood alist)))
    (dolist (pair alist)
      (let ((rep (car pair))
            (val (cdr pair)))
        (let ((slot (assoc val (rep-nogoods rep))))
          (cond (slot (or (member nogood (cdr slot)) (push nogood (cdr slot))))
                ((or (null (rep-nogoods rep))
                     (< val (caar (rep-nogoods rep))))
                 (push (list val nogood) (rep-nogoods rep)))
                (t (do ((ng (rep-nogoods rep) (cdr ng)))
                       ((or (null (cdr ng))
                            (< val (caar (cdr ng))))
                        (setf (cdr ng)
                              (cons (list val nogood)
                                    (cdr ng))))))))))))
```

TABLE 6-26. Formation and Installation of Nogood Sets.

```
(defun choose-culprit (losers)
  (format t "~%;;; These are the premises that seem to be at fault:~
             ~:{~%;~8X~S~@{ == ~S~}~:↑,~}."
          (forlist (p losers)
            (cons p (mapcan #'(lambda (c)
                                (and (globalp c)
                                     (eq (cell-true-supplier c) p)
                                     (list (cell-name c))))
                            (node-cells p)))))
  (format t "~%;;; Choose one of these to retract and RETURN it.")
  (let ((culprit (break "Choose Culprit")))
    (cond ((or (eq culprit @defer) (eq culprit @punt)) culprit)
          ((memq (cell-true-supplier culprit) losers)
           (cell-true-supplier culprit))
          (t (choose-culprit losers)))))
```

TABLE 6-27. Choosing a Culprit.

Nogood sets have the same structure that they did in previous versions of the system. However, `form-nogood-set` (Table 6-26) has been split into two functions, one to print a trace message and form the nogood a-list, and one (`install-nogood-set`) to do the real work. The latter function is called from within `run-contra` (Table 6-21).

The function `choose-culprit` (Table 6-27) has changed a bit, to allow the return of the flags `@defer` and `@punt` in place of a culprit. Also, a contradiction can involve several cells of the same node, and if the culprit is identified by returning a cell other than one of the premises, it isn't enough to test that it is in the same node as a premise, for that may not uniquely identify the intended culprit. Instead, if an alias is supplied then its true-supplier must be one of the premises. To aid in this discrimination, a name is printed in the message as an alias only if it is suitable for identifying a culprit.

```
(statistics-counter awaken "Awakenings")
(statistics-counter awaken-added "@ADDED awakenings")
(statistics-counter awaken-forget "@FORGET awakenings")
(statistics-counter awaken-nogood "@NOGOOD awakenings")

(defun awaken (cell reason)
  (require-cell cell)
  (statistic awaken)
  (let ((con (cell-owner cell)))
    (cond ((not (null con))
           (require-constraint con)
           (let ((rulearray (select! reason
                              ((@added)
                               (statistic awaken-added)
                               (ctype-added-rules (con-ctype con)))
                              ((@forget)
                               (statistic awaken-forget)
                               (ctype-forget-rules (con-ctype con)))
                              ((@nogood)
                               (statistic awaken-nogood)
                               (ctype-nogood-rules (con-ctype con))))))
             (dolist (rule (aref rulearray (cell-name cell)))
               (or (bit-test (rule-id-bit rule) (con-queued-rules con))
                   (do ((tr (rule-triggers rule) (cdr tr))
                        (rebelp ()
                                (or rebelp
                                    (let ((v (aref (con-values con) (car tr))))
                                      (or (eq (cell-state v) @rebel)
                                          (eq (cell-state v) @dupe))))))
                       ((null tr)
                        (cond (rebelp (enqueue (list rule con reason)
                                               *rebel-queue*))
                              (t (enqueue-rule rule con reason))))
                     (or (node-boundp (aref (con-values con) (car tr)))
                         (return)))))))))

(defun awaken-all (cells reason &optional (exception () exceptionp))
  (and exceptionp (require-cell exception))
  (dolist (cell cells)
    (require-cell cell)
    (and (not (eq cell exception))
         (awaken cell reason))))
```

TABLE 6-28. Awakening of Rules.

6.3.21. Awakening Selects Only Relevant Rules for Queuing

The rule-array structure for constraint-types is a pre-compiled catalogue indexing for each pin and each reason for awakening which rules should be run. All that `awaken` (Table 6-28) need do is check that the given cell has an owner, select the appropriate array from the constraint's type, and index into the array according to the cell's pin-number, and *voila!* all the relevant rules are in hand.

The rules could simply be enqueued on `*vanilla-queue*` and the system would work. However, an attempt is made to avoid enqueuing rules whose triggers do not all have values. (This situation might change between the time the rule is enqueued and the time it is dequeued, but if that occurs the rule will be queued anyway when the other triggers gain values.) Also, if any trigger is a rebel value then the rule is put on the low-priority `*rebel-queue*` on the intuition that one should compute values that have certain support in preference to those that do not (this can only occur anyway if contradictions have been deferred).

The function `awaken-all` awakens a list of cells for a specified reason, but will avoid awakening a particular cell if that is given as a third argument. This is generally used to awaken all the cells of a node except that which generated the value.

```
(statistics-counter forget "Values forgotten")

(defun forget (cell &optional (source () sourcep)  (via () viap))
  (require-cell cell)
  (and (eq (cell-rule cell) *constant-rule*)
       (lose "Illegal to FORGET the constant ~S." cell))
  (and sourcep (require-cell source))
  (and viap (require-cell via))
  (statistic forget)
  (ctrace "Removing ~S from ~S~:[~3*~; because ~:[of ~;~S==~]~S~]."
          (cell-contents cell)
          (cell-goodname cell)
          sourcep
          (and viap (not (eq via source)))
          (and viap (not (eq via source)) (cell-goodname via))
          (and sourcep (cell-goodname source)))
  (select! (cell-state cell)
    ((@friend)
     (setf (cell-contents cell) ())
     (setf (cell-rule cell) ())
     (setf (cell-state cell) @slave))
    ((@rebel)
     (setf (cell-state cell) @slave)
     (decrement (node-contra cell))
     (awaken cell @added)
     (let ((fcellsets '()))
       (dolist (c (rep-cells (cell-repository cell)))
         (cond ((and (eq (cell-state c) @dupe)
                     (eq (cell-contents c) cell))
                (setf (cell-state c) @slave)
                (awaken c @added)
                (push (cons c (forget-consequences c)) fcellsets))))
       (dolist (q fcellsets)
         (dolist (f (cdr q))
           (forget f cell (car q))))))
    ((@king)
     (do ((x (node-cells cell) (cdr x)))
         ((null x)
          (forget-friendless-king cell))
       (cond ((eq (cell-state (car x)) @friend)
              (usurper (car x))
              (setf (cell-contents cell) ())
              (setf (cell-rule cell) ())
              (setf (cell-state cell) @slave)
              (or (zerop (node-contra cell))
                  (dolist (c (node-cells cell))
                    (and (eq (cell-state c) @rebel)
                         (enqueue (list @node c (car x)) *contra-queue*))))
              (return)))))
    ((@slave @puppet @dupe))))
```

TABLE 6-29. Forgetting a Cell's Value and Its Consequences.

6.3.22. Forgetting a Cell's Value Lets Friends (Or Rebels) Step In

When a cell's value is forgotten, the "begging" process done in previous versions of the system need not be performed. It is not necessary to beg a rule to compute a value for its output pin, because it will do so when it is good and ready and has all its triggers; and the value, once computed, will not be lost because every cell can have a value. (The exceptions are rules with no triggers—they are invoked when the constraint is generated, or when the status of a nogood set changes if they are `&nogood` or `&nogoodbeg` rules. Also, a constraint-type's `ctype-forget-rules` array will prove very useful for explanation purposes.)

On the other hand, when a supplier cell's value is forgotten and another cell of the node has a value, the second cell may immediately step in as the new supplier for the node (this is the advantage of recording multiple support for values), and avoid further perturbations of the network. If the new value is different, however (provided by a former rebel), then rules need to be awakened on the newly added trigger value. Therefore, paradoxically, the `forget` function only performs awakenings for the reason `@added!`

```
(defun forget-consequences (cell)
  (let ((fcells '()))
    (and (cell-owner cell)
         (doarray (v (con-values (cell-owner cell)))
           (select! (cell-state v)
             ((@king @friend @rebel)
              (and (not (eq v cell))                    (member (cell-name cell)
                           (rule-triggers (cell-rule v)))
                   (push v fcells)))
             ((@slave @puppet @dupe)))))
    fcells))

(defun retract (cell)
  (*retract (cell-true-supplier cell))
  (run?))

(defun *retract (cell)
  (require-cell cell)
  (ctrace "Retracting the premise ~S." cell)
  (forget cell))

(defun change (cell value)
  (require-cell cell)
  (require-integer value)
  (let ((s (cell-true-supplier cell)))
    (let ((rule (cell-rule s)))
      (cond ((not (or (eq rule *default-rule*)
                      (eq rule *parameter-rule*)))
             (lose "Supplier of ~S is not a DEFAULT or PARAMETER." cell))
            ((or (not (*forbiddenp value s))
                 (y-or-n-p "That value is contradictory; do it anyway?"))
             (*retract s)
             (process-setc-work () rule value s)
             (run?))))))
```

TABLE 6-30. Retracting a Value, and Tracing of Consequences.

The actions taken when a cell's value is forgotten depends on the state of the cell. If it is a slave, puppet, or dupe, then nothing need be done, as it has no value. (It might seem at first that such a cell should not be forgotten in the first place. However, if `x` was computed from triggers `y` and `z`, and `y` had `z` as a trigger, and then `z` is retracted, then when `z` is forgotten both `y` and `x` must be forgotten. If `y` is forgotten, then `x` is recursively forgotten; it may then become, say, a slave. Then `x` may be forgotten again on account of `z`.)

If the cell to be forgotten is a friend, then its value quietly disappears and it becomes a slave to the king. No other cell is affected. If it is a rebel, then it and all its dupes become slaves. Their rules must be awakened for reason `@added` because they all suddenly become aware of the value of the king. (Such awakened rules are not run at once—merely queued for later processing. This is important to the integrity of the system; the forgetting process must complete before any rules are run to ensure that all computed quantities have well-founded support. (Well-foundedness is not the same

thing as consistency; it merely means that a value is correctly derived from premises. The premises need not be consistent, however, for computed values to be well-founded. Indeed, contradictions are detected by the very fact that two well-founded values conflict. Conflicting values that are not well-founded are not informative.)

If a king is to be forgotten, then a major upheaval occurs. If the king has a friend, then the friend steps into its place. This is the most desirable alternative because the primary value of the node does not change, and slaves need not be bothered. The friend usurps the throne, and if there are any rebels then contradictions tasks for the conflict between the former friend and the rebel must be enqueued.

```
(defun forget-friendless-king (cell)
  (require-cell cell)
  (dolist (nogood (cdr (assoc (cell-contents cell) (node-nogoods cell))))
    (do ((ng (cdr nogood) (cdr ng))
         (unique-loser ()))
        ((null ng)
         (and unique-loser
              (awaken-all (rep-cells unique-loser) @nogood)))
      (and (or (not (node-boundp (rep-supplier (caar ng))))
               (not (equal (node-value (rep-supplier (caar ng)))
                           (cdar ng))))
           (if unique-loser
               (return)
               (setq unique-loser (caar ng))))))
  (let ((fcellsets '()) (rebel ()))
    (dolist (c (rep-cells (cell-repository cell)))
      (select! (cell-state c)
        ((@king @dupe))
        ((@slave) (push (cons c (forget-consequences c)) fcellsets))
        ((@rebel) (setq rebel c))
        ((@friend) (lose "Already established that ~S had no friends." cell))
        ((@puppet) (lose "@KING ~S and @PUPPET ~S in same node." cell c))))
    (cond ((null rebel)
           (setf (cell-contents cell) ())
           (setf (cell-rule cell) ())
           (setf (cell-state cell) @puppet)
           (awaken-all (node-cells cell) @nogood))
          (t (usurper rebel)
             (setf (cell-contents cell) ())
             (setf (cell-rule cell) ())
             (setf (cell-state cell) @slave)
             (decrement (node-contra cell))
             (awaken cell @added)
             (dolist (c (rep-cells (cell-repository cell)))
               (select! (cell-state c)
                 ((@slave) (awaken c @added))
                 ((@king))
                 ((@rebel)
                  (cond ((equal (cell-contents c) (cell-contents rebel))
                         (setf (cell-state c) @friend)
                         (decrement (node-contra cell)))
                        (t (enqueue (list @node c rebel) *contra-queue*))))
                 ((@dupe)
                  (and (equal (cell-contents (cell-contents c))
                              (cell-contents rebel))
                       (setf (cell-state c) @slave)))
                 ((@puppet @friend)
                  (lose "Impossible cell state for ~S." c))))))
    (dolist (q fcellsets)
      (dolist (f (cdr q))
        (forget f cell (car q))))))
```

TABLE 6-31. Forgetting a Friendless King (Very Hairy!).

If a king to be forgotten has no friends, then several things happen (performed by the function `forget-friendless-king` in Table 6-31). First of all, it is the king that affect nogood sets (nogood sets being per-node instead of per-cell), so if a king disappears then all the nogood sets of the node for the disappearing value must be checked. If all the other nodes but one in a nogood set have their associated values, then the disappearance of this king might unblock an assumption for the lone node not having its paired value; rules for that node (called the `unique-loser` in the code) must be awakened.

After the nogood awakenings are taken care of, then all the consequences of forgetting the king must be enumerated. The variable `fcellsets` is a list of buckets; each bucket is headed by a slave of the king, and contains immediate consequences of that slave. These consequences will be forgotten in turn, but not until the state of the current node has been resolved. (It was much easier to write the `forget` function if it could be assumed that every node encountered was in a consistent state, rather than in a half-forgotten state.) The function `forget-consequences` (Table 6-30) enumerates the consequences of a cell by examining all the pins of that cell's constraint and finding those pins for which the given cell was a trigger. (By the way, this enumeration of consequences is also done when a rebel is forgotten—see Table 6-29.)

While the consequences are being enumerated, it is noted whether there is any rebel. If there is not, then the king becomes a puppet, and the node loses its value, whereupon `@nogood` rules must be given a chance to run. (This would be the obvious place to run `@forget` rules, but, as already noted, this is unnecessary.) Otherwise, one rebel is chosen arbitrarily (the last one seen) to become the new king. It usurps the old king, which becomes a slave. The total number of rebels for the node is decreased by one. Then every other cell of the node must be examines. Slaves are awakened to the new value. Other rebels either become friends (in which case the count of rebels is decremented) or remain rebels (in which case a contradiction with the new king is enqueued). Dupes of rebels which are to become friends are turned into slaves; they need not be awakened, as they already knew of the "correct" value. Puppets cannot occur in node which has a value, and by supposition the old king was friendless, so no friends can be encountered.

The function `retract` (Table 6-30) is now the user interface to the retraction mechanism. It calls `*retract` to do the work on the given cell's true-supplier (in that way the user can say `(retract centigrade)` to specify retraction of the `default` cell connected to `centigrade`, for example). Then `run?` is called to allow queued tasks to be processed.

The function `change` will change the value of a default or parameter cell. If the value is forbidden by a nogood set, it asks the user before blundering onward. It then retracts the old value, installs the new one (by pretending to do a `setc`-type operation), and then runs the task scheduler. (This implementation is extremely simple-minded. Although it checks the nogood sets, if the value is found to be contradictory it does not immediately enqueue a contradiction on the basis of the nogood set (which would not be hard to do). Instead, the computation will truly blunder onward

```
(defmacro the (x y) `(*the ',x ,y))

(defun *the (name con)
  (require-constraint con)
  (or (lookup name con) (lose "~S has no part named ~S." con name)))

(defun lookup (name thing)
  (require-constraint thing)
  (let ((names (ctype-vars (con-ctype thing)))
        (cells (con-values thing)))
    (let ((n (array-length names)))
      (do ((j 0 (+ j 1)))
          ((= j n) ())
        (and (eq (aref names j) name) (return (aref cells j)))))))
```

TABLE 6-32. Referring to Pins Using `the`.

until the contradiction is rediscovered. I was feeling lazy the day I wrote this code. A more complicated idea would be to take advantage of the fact that a value was not being retracted, but merely changing. This would involve running rules while the `forget` process was only half done, and would require great care. However, it is certainly what people do when adjusting constrained values.)

6.3.23. The `lookup` Functions Scans the Constraint-types's `vars` Array

The `the` macro, and its utility function `*the`, are now primarily for user interface since the implementation now deals internally with pin-numbers rather than pin-names. The `lookup` function scans the vars array of the constraint-type, and if the name is found returns the corresponding component of the constraint's values array.

```
(statistics-counter equatings "Number of calls to ==")

(defun == (cell1 cell2)
  (require-cell cell1)
  (require-cell cell2)
  (statistic equatings)
  (let ((x1 (memq cell1 (cell-equivs cell2)))
        (x2 (memq cell2 (cell-equivs cell1))))
    (cond ((or x1 x2)
           (or (and x1 x2 (eq (cell-repository cell1) (cell-repository cell2)))
               (lose "EQUIVS lists not consistent for ~S and ~S." cell1 cell2)))
          ((not (eq cell1 cell2))
           (push cell1 (cell-equivs cell2))
           (push cell2 (cell-equivs cell1))))))
  (or (eq (cell-repository cell1) (cell-repository cell2))
      (let ((r1 (cell-repository cell1))
            (r2 (cell-repository cell2))
            (cb1 (node-boundp cell1))
            (cb2 (node-boundp cell2)))
        (let ((r (merge-values cell1 cell2))
              (rcells (append (rep-cells r1) (rep-cells r2))))
          (let ((newcomers (if cb1 (if cb2 '() (rep-cells r2))
                               (if cb2 (rep-cells r1) '()))))
            (xr (if (eq r r1) r2 r1)))
            (setf (rep-cells r) rcells)
            (dolist (cell (rep-cells xr)) (setf (cell-repository cell) r))
            (let ((fcells (alter-nogoods-rep xr r)))
              (setf (rep-nogoods r)
                    (merge-nogood-sets (rep-nogoods r) (rep-nogoods xr)))
              (awaken-all fcells @nogood))
            (awaken-all newcomers @added)              (run?)
            'done)))))
```

TABLE 6-33. Equating of Cells and Recording Equatings Explicitly.

6.3.24. Equatings are Recorded Explicitly and Initialize Links

When an equating is done (using `==`), it must be explicitly recorded even if it is redundant by transitivity. (See Table 6-33.) If this exact equating between these exact two cells has already been done, then it need not be recorded twice; otherwise each cell is added to the other's `equivs` list. (Therefore the equatings are recorded redundantly, in that each equating is recorded twice, one in each cell. This is mostly for pleasant symmetry and error-checking, and is not crucial to the implementation.)

Once this is done, then the rest of the work is relevant only if the cells are currently of two different nodes (determined by comparing their repositories). If they are of different nodes, then `merge-values` is called to compare the values. In this version, `merge-values` will not only compare the values and detect contradictions, but also rearrange the cell links and do other housekeeping. It returns as its value the repository of the node which is to provide the supplier for the merged node. The rest of `==` is pretty much as before. The set of newcomers is determined, the node structure is updated, the nogood sets are merged, cells may be awakened on account of the nogood sets, and the newcomers are awakened (they could be awakened right after they are determined, for awakening merely enqueues now; but I was lazy and left the code similar to previous versions —it doesn't hurt).

```
(defun merge-values (cell1 cell2)
  (require-cell cell1)
  (require-cell cell2)
  (let ((r1 (cell-repository cell1))
        (r2 (cell-repository cell2)))
    (cond ((not (node-boundp cell1))
           (let ((s (rep-supplier r1)))
             (or (eq (cell-state s) @puppet)
                 (lose "Valueless node had a non-@PUPPET supplier ~S." s))
             (setf (cell-state s) @slave))
           (point-links-toward cell1)
           (setf (cell-link cell1) cell2)
           r2)
          ((not (node-boundp cell2))
           (let ((s (rep-supplier r2)))
             (or (eq (cell-state s) @puppet)
                 (lose "Valueless node had a non-@PUPPET supplier ~S." s))
             (setf (cell-state s) @slave))
           (point-links-toward cell2)
           (setf (cell-link cell2) cell1)
           r1)
          (t (let ((r (cond ((eq (node-rule cell1) *constant-rule*) r1)
                            ((eq (node-rule cell2) *constant-rule*) r2)
                            ((ancestor cell1 cell2) r1)
                            ((ancestor cell2 cell1) r2)
                            ((plusp (rep-contra r2)) r1)
                            (t r2))))
               (if (eq r r1)
                   (merge-two-values r r2 cell1 cell2)
                   (merge-two-values r r1 cell2 cell1)))))))
```

TABLE 6-34. Merging Values and Arranging Cell Links.

The function `merge-values` (Table 6-34) is responsible for deciding which node will provide the repository (and thus the supplier) for the merged node. It is also reponsible for installing a new cell link. The new link will always be between the two cells given to `==`; this guarantees that links follow paths laid down by explicit equatings. My initial impulse was to link the deposed supplier to the surviving supplier, because then all the other links need not be changed to preserve the property that the links lead eventually to the supplier. However, this fails to preserve the property of following explicit equatings. The solution is that the given cell of the node not providing the repository must usurp its own supplier. Then link paths from all cells of that node will lead to that given cell, and thence to the other given cell, and so to the surviving supplier. The function `point-links-toward` is used instead of `usurper` (see Table 6-19 (page 231)) to avoid changing the cell states (it doesn't much matter in the cases occuring in Table 6-34, because the deposed supplier has been made into a slave).

```
(defun merge-two-values (r xr cell xcell)
  (let ((val (cell-contents (rep-supplier r))))
    (cond ((and (zerop (rep-contra xr))
                (equal (cell-contents (rep-supplier xr)) val))
           (setf (cell-state (rep-supplier xr)) @friend))
          (t (ctrace "Contradiction when merging ~S and ~S." cell xcell)
             (dolist (c (rep-cells xr))
               (select! (cell-state c)
                 ((@slave)
                  (cond ((not (equal (cell-contents (node-supplier c)) val))
                         (setf (cell-state c) @dupe)
                         (setf (cell-contents c) (rep-supplier xr)))))
                 ((@rebel @king @friend)
                  (cond ((equal (cell-contents c) val)
                         (setf (cell-state c) @friend))
                        (t (setf (cell-state c) @rebel)
                           (increment (rep-contra r))
                           (enqueue (list @node c (rep-supplier r))
                                    *contra-queue*))))
                 ((@dupe)
                  (and (equal (cell-contents (cell-contents c)) val)
                       (setf (cell-state c) @slave)))
                 ((@puppet) (lose "Puppet ~S in a bound node." c)))))))
  (point-links-toward xcell)
  (setf (cell-link xcell) cell)
  r)
```

TABLE 6-35. Merging Two Nodes with Values and Handling Conflicts.

If both cells have values, then one is chosen on the basis of certain criteria, some of them heuristic. Constants are preferred for surviving kings. Barring that, the avoiding of circular dependency structures is paramount. If that does not resolve the issue, then nodes with internal contradictions are less desirable than consistent nodes. Once a repository has been chosen, then the rest of the work is handed off to `merge-two-values` (Table 6-35).

If the node whose king is being deposed (represented by `xr` and `xcell`) is free of contradiction, and the two values agree, then the situation is particularly easy and is handled as a special case. The deposed king becomes a a friend of the surviving king, and his old friends and slaves automatically become friends and slaves of the surviving king, and that is that. Otherwise a contradiction has occurred—either the deposed king or one of his rebels must disagree with the surviving king. All the cells of the `xr` node are processed. Slaves to a disagreeing old king become dupes. (Note that the phrase `(cell-contents (node-supplier c))` is used rather than the seemingly equivalent `(node-value c)`; this is done because `node-value` performs an important error-check that must be circumvented here because the node is temporarily in a bad situation.) Rebels, friends, and the king may become either friends or rebels, depending on the values involved. Dupes may become slaves if their associated rebels become friends. When all this is done, and contradictions have been enqueued, the cell links are set up and the chosen repository returned.

```
(defun alter-nogoods-rep (xr r)
  (let ((fcells '()))
    (dolist (bucket (rep-nogoods xr))
      (dolist (nogood (cdr bucket))
        (let ((z (assq r (cdr nogood)))
              (xz (assq xr (cdr nogood))))
          (cond ((null xz)
                 (lose "Funny nogood set ~S for bucket ~S of repository ~S."
                       xr (car bucket) nogood))
                ((null z)
                 (setf (cdr nogood)
                       (add-nogood-pair r (cdr xz) (delassq xr (cdr nogood)))))
                ((equal (cdr z) (cdr xz))
                 (setf (cdr nogood) (delassq xr (cdr nogood))))
                (t (dolist (pair (cdr nogood))
                     (setq fcells (append (rep-cells (car pair)) fcells))
                     (let ((buck (assoc (cdr pair) (rep-nogoods (car pair)))))
                       (or buck (lose "Nonexistent bucket: ~S." pair))
                       (setf (cdr buck) (delq nogood (cdr buck)))
                       (or (cdr buck)
                           (setf (rep-nogoods (car pair))
                                 (delrassq '() (rep-nogoods (car pair))))))))))))
    fcells))

(defun add-nogood-pair (rep val nogoodlist)
  (require-repository rep)
  (cond ((null nogoodlist) (list (cons rep val)))
        ((node-lessp (car (rep-cells rep)) (car (rep-cells (caar nogoodlist))))
         (cons (cons rep val) nogoodlist))
        (t (cons (car nogoodlist) (add-nogood-pair rep val (cdr nogoodlist))))))

(defun merge-nogood-sets (s1 s2)
  (cond ((null s1) s2)
        ((null s2) s1)
        ((< (caar s1) (caar s2))
         (cons (car s1) (merge-nogood-sets (cdr s1) s2)))
        ((> (caar s1) (caar s2))
         (cons (car s2) (merge-nogood-sets s1 (cdr s2))))
        (t (cons (cons (caar s1) (merge-nogood-buckets (cdar s1) (cdar s2)))
                 (merge-nogood-sets (cdr s1) (cdr s2))))))

(defun merge-nogood-buckets (b1 b2)
  (cond ((null b1) b2)
        ((member (car b1) b2) (merge-nogood-buckets (cdr b1) b2))
        (t (cons (car b1) (merge-nogood-buckets (cdr b1) b2)))))
```

TABLE 6-36. Altering and Merging of Nogood Sets.

The code for altering and merging of nogood sets is unchanged, because the representation of nogood sets is the same. For completeness the code is reproduced here in Table 6-36.

```
(defun ancestor (cell1 cell2)
  (require-cell cell1)
  (require-cell cell2)
    (or (eq (cell-repository cell1) (cell-repository cell2))
        (select! (cell-state cell2)
          ((@king @rebel) (ancestor-triggers cell1 cell2))
          ((@friend) (ancestor-triggers cell1 (node-supplier cell2)))
          ((@slave) (and (node-boundp cell2)
                         (ancestor-triggers cell1 (node-supplier cell2))))
          ((@puppet) ())
          ((@dupe) (ancestor-triggers cell1 (cell-contents cell2))))))

(defun ancestor-triggers (cell1 cell2)
  (require-cell cell1)
  (require-cell cell2)
  (do ((tns (rule-triggers (cell-rule cell2)) (cdr tns)))
      ((null tns) ())
    (and (ancestor cell1 (aref (con-values (cell-owner cell2)) (car tns)))
         (return t))))
```

TABLE 6-37. Testing Ancestorhood.

The tracing of ancestors by the function `ancestor` is similar in spirit if not in implementation to previous versions. The code appears in Table 6-37. Note that a dependency chain might actually wind through a single node more than once, if the node contains rebels. It might wind in through a dupe, out through a rebel; in through a slave, out through the king; in through another dupe, and so on. The low priority accorded to rules triggered by rebels and dupes is intended to avoid such occurrences, but it can legitimately happen.

```
(defun dissolve (cell) (*dissolve cell) (run?))

(defun *dissolve (cell)
  (require-cell cell)
  (fast-expunge-nogoods cell)
  (let ((supplier (node-supplier cell))
        (cells (node-cells cell)))
    (ctrace "Dissolving ~{~<;/|   ~2,72:;~S~>~^, ~}."
            (forlist (c cells) (cell-goodname c)))
    (dolist (c cells)
      (setf (cell-link c) ())
      (setf (cell-equivs c) '())
      (or (eq c supplier)
          (let ((r (gen-repository)))
            (select! (cell-state c)
              ((@friend @rebel)
               (setf (cell-state c) @king))
              ((@dupe @slave)
               (or (node-boundp supplier)
                   (setf (cell-state c) @puppet)))
              ((@king @puppet)
               (lose "@KING or @PUPPET ~S was not the supplier." c)))
            (setf (rep-supplier r) c)
            (setf (cell-repository c) r)
            (push c (rep-cells r)))))
    (setf (node-cells supplier) (list supplier))
    (setf (node-contra supplier) 0)
    (and (node-boundp supplier)
         (let ((fcells '()))
           (dolist (c cells)
             (select (cell-state c)
               ((@slave @dupe)
                (setf (cell-state c) @puppet)
                (setq fcells (nconc (forget-consequences c) fcells)))))
           (dolist (f fcells) (forget f)))))
  'done)
```

TABLE 6-38. Dissolving a Node.

6.3.25. Node Disconnections Can be Done by Dissolving and Reconnecting

When a node is dissolved, things are a little complicated, because friends and rebels can become kings. On the other hand, the former nonsense about restoring values to default and constant cells pleasantly vanishes here. The function `dissolve` (Table 6-38) tears all the cells of a node apart and generates new repositories for each one but the supplier. (A supplier does not have to be chosen artificially for a valueless node, because there is always a supplier, even if only a puppet.) The cell links and recorded equivalences are obliterated. If the node had had a supplier, then any cells which had been slaves or dupes will no longer have values, and so are subject to the forgetting

process. Therefore they are left marked as slaves or dupes until late in the process, until their consequences have been recorded for forgetting, whereupon they become puppets.

```
(defun detach (cell) (*detach cell) (run?))

(defun *detach (cell)
  (require-cell cell)
  (dolist (c (cell-equivs cell))
    (setf (cell-equivs c) (delq cell (cell-equivs c))))
  (setf (cell-equivs cell) ())
  (reconstruct-node cell))

(defun disconnect (cell) (*disconnect cell) (run?))

(defun *disconnect (cell)
  (require-cell cell)
  (dolist (c (cell-equivs cell))
    (setf (cell-equivs c)
          (unionq (remq c (cell-equivs cell))
                  (delq cell (cell-equivs c)))))
  (setf (cell-equivs cell) ())
  (reconstruct-node cell))

(defun disequate (cell1 cell2) (*disequate cell1 cell2) (run?))

(defun *disequate (cell1 cell2)
  (require-cell cell1)
  (require-cell cell2)
  (and (eq (cell-repository cell1) (cell-repository cell2))
       (let ((x1 (memq cell1 (cell-equivs cell2)))
             (x2 (memq cell2 (cell-equivs cell1))))
         (cond ((and x1 x2)
                (setf (cell-equivs cell1) (delq cell2 (cell-equivs cell1)))
                (setf (cell-equivs cell2) (delq cell1 (cell-equivs cell2)))
                (and (or (eq cell1 (cell-link cell2))
                         (eq cell2 (cell-link cell1)))
                     (reconstruct-node cell1)))
               ((or x1 x2)
                (lose "Inconsistent EQUIVS lists for ~S and ~S." cell1 cell2))))))
  'done)

(defun reconstruct-node (cell)
  (require-cell cell)
  (let ((equivs '())
        (*run-flag* ()))
    (dolist (c (node-cells cell))
      (dolist (e (cell-equivs c))
        (push (cons c e) equivs)))
    (*dissolve cell)
    (dolist (q equivs)
      (== (car q) (cdr q))))
  (run?))
```

TABLE 6-39. Detaching, Disconnecting, and Disequating Cells.

Rather than implementing all the special cases for disconnecting, detaching, and disequating, which are rather horrendous in their details, for ease of implementation I borrowed an idea from

L. Peter Deutsch: to change the connections of just a few cells, simply dissolve the whole node and then re-assert all the equatings except the ones to be abolished. This carries a time penalty, but makes implementation *much* easier.

Table 6-39 contains the code for `detach`, `disconnect`, and `disequate`. The first is defined to pretend that all equatings involving a given node had never taken place. The function `*detach` removes the cell from the equivs lists of all cells it had been equated to, erases the equatings of the given cell, and then calls `reconstruct-node` to do the dirty work.

The function `disconnect` is defined to remove itself from the node but otherwise leave the node intact, so it must add new equatings among all the things to which it had formerly been connected, to ensure that they do not become disconnected. The work function `*detach` removes the given cell from equivs lists, set-unions its own equivs list into its former buddies' equivs lists, erases its own connections, and then calls `reconstruct-node`.

The function `disequate` must undo any equating between the two given cells. Nothing need be done if they had not already been equated, but if they had then `*disequate` deletes each from the other's equivs list (after some error-checking), but only needs to reconstruct the node if deleting the equating affected the node's cell-links structure.

The interesting part is in `reconstruct-node`. It is defined to take a node whose equivs lists have been messed with and make the node structure consistent with those lists. It makes up a list of equatings to be done, dissolves the node, and then calls `==` to do each equating. (Because the equatings are recorded redundantly, as described in §6.3.24, twice as many calls as necessary are made to `==`; but this doesn't hurt anything.) This is done with `*run-flag*` bound to `()` to prevent tasks from running until the node is reconstructed—no use in computing values on the basis of a false network structure!

```
(defmacro mark-node (cell) `(setf (node-mark ,cell) t))
(defmacro unmark-node (cell) `(setf (node-mark ,cell) ()))
(defmacro markp (cell) `(node-mark ,cell))

(defun fast-expunge-nogoods (cell)
  (require-cell cell)
  (fast-expunge-nogoods-mark cell)
  (fast-expunge-nogoods-unmark cell))

(defun fast-expunge-nogoods-mark (cell)
  (require-cell cell)
  (cond ((not (markp cell))
         (mark-node cell)
         (and (not (null (node-nogoods cell)))
              (awaken-all (node-cells cell) @nogood))
         (setf (node-nogoods cell) '())
         (dolist (c (node-cells cell))
           (and (cell-owner c)
                (doarray (v (con-values (cell-owner c)))
                  (fast-expunge-nogoods-mark v)))))))

(defun fast-expunge-nogoods-unmark (cell)
  (require-cell cell)
  (cond ((markp cell)
         (unmark-node cell)
         (dolist (c (node-cells cell))
           (and (cell-owner c)
                (doarray (v (con-values (cell-owner c)))
                  (fast-expunge-nogoods-unmark v)))))))
```

TABLE 6-40. Fast Expunging of Nogood Information.

When a node is dissolved, this implementation follows previous implementations in simply expunging all the nogood information in the entire network. Now that `premises` computes the precise equivalences involved in a contradiction, it would not be so difficult to add this information to a nogood set when it was formed, and to cross-reference nogood sets in each node containing an equivalence mentioned in a nogood set. Then when a node was dissolved, only relevant nogood sets need be expunged. However, this involves saving a great deal of information as data structures, which may not be worth it, and so I have not investigated this technique. Note that `fast-expunge-nogoods-mark` (Table 6-40) awakens every cell it encounters for reason `@nogood` , which can take a while to process. Fortunately, there are not that many rules which awaken on that condition.

```
(defmacro destroy (symbol) `(*destroy ',symbol))

(defun *destroy (symbol)
  (require-symbol symbol)
  (and (boundp symbol)
       (let ((val (symeval symbol)))
         (cond ((cell-p val)
                (cond ((and (globalp val) (eq (cell-name val) symbol))
                       (*detach val)
                       (makunbound (cell-id val))
                       (makunbound symbol))
                      (t (lose "Illegal re-declaration of ~S." symbol))))
               ((constraint-p val)
                (cond ((eq (con-name val) symbol)
                       (forarray (p (con-values val)) (*detach p))
                       (makunbound symbol))
                      (t (lose "Illegal re-declaration of ~S." symbol))))
               ((or (constraint-type-p val) (repository-p val) (rule-p val))
                (lose "Illegal re-declaration of ~S." symbol))
               (t (makunbound symbol)))))
  'done)
```

TABLE 6-41. Destroying the Value of a Global Name.

6.3.26. Destroying a Variable or Constraint Detaches It from Everything

At last we may discuss the function `*destroy` referred to in §6.3.9. It is used by `create` and `variable` (Table 6-11 (page 217)) as well as by `destroy` (Table 6-41). The function `*destroy` takes a LISP symbol, and if that symbol has a value examines that value. If the value is a cell, then the cell must be global and have the symbol as its name; otherwise the user must be trying to destroy one of the generated unique debugging id names. The cell is detached, and the symbol made to have no value (the LISP function `makunbound` removes the value from a symbol). Similarly, if the value is a constraint, then if the symbol is the constraint's name, the constraint's pins are all detached. It is illegal to destroy the name for a constraint-type, or the id for a repository or rule. A name not used to name any of the system data structures may be destroyed.

```
(defprim (adder +) (c a b)
  (c (a b) (+ a b))
  (b (a c) (- c a))
  (a (b c) (- c b)))

(defprim (multiplier *) (c a b)
  (c (a) (if (zerop a) 0 @dismiss))
  (c (b) (if (zerop b) 0 @dismiss))
  (c (a b) (* a b))
  (b (a c) (if (and (not (zerop a)) (zerop (\ c a)))
               (// c a)
               @dismiss))
  (a (b c) (if (and (not (zerop b)) (zerop (\ c b)))
               (// c b)
               @dismiss)))

(defprim (maxer max) (c a b)
  (c (a b) (max a b))
  (b (a c) (cond ((< a c) c)
                 ((> a c) @lose)
                 (t @dismiss)))
  (a (b c) (cond ((< b c) c)
                 ((> b c) @lose)
                 (t @dismiss))))

(defprim (minner min) (c a b)
  (c (a b) (min a b))
  (b (a c) (cond ((> a c) c)
                 ((< a c) @lose)
                 (t @dismiss)))
  (a (b c) (cond ((> b c) c)
                 ((< b c) @lose)
                 (t @dismiss))))

(defprim (equality =) (p a b)
  ((p) (if (or (= p 0) (= p 1)) @dismiss @lose))
  ((p &nogoodbeg) () (resolve-among '(0 1)))
  (p (a b) (if (= a b) 1 0))
  (b (p a) (if (= p 1) a @dismiss))
  (a (p b) (if (= p 1) b @dismiss)))

(defprim gate (p a b)
  ((p) (if (or (= p 0) (= p 1)) @dismiss @lose))
  ((p &nogoodbeg) () (resolve-among '(0 1)))
  (p (a b) (if (= a b) @dismiss 0))
  (b (p a) (if (= p 1) a @dismiss))
  (a (p b) (if (= p 1) b @dismiss)))
```

TABLE 6-42. Definition of Primitive Constraint-types (i).

```
(defprim (lesser <) (a b)
  ((a b) (if (< a b) @dismiss @lose)))

(defprim (lesser! <!) (p a b)
  ((p) (if (or (= p 0) (= p 1)) @dismiss @lose))
  ((p &nogoodbeg) () (resolve-among '(0 1)))
  (p (a b) (if (< a b) 1 0)))

(defprim (lesser? <?) (p a b)
  ((p) (if (or (= p 0) (= p 1)) @dismiss @lose))
  ((p &nogoodbeg) () (resolve-among '(0 1)))
  (p (a b) (if (< a b) @dismiss 0)))

(defprim (?lesser ?<) (a b)
  ((a &nogoodbeg) (b) (if (forbiddenp (- b 1)) @dismiss (- b 1)))
  ((b &nogoodbeg) (a) (if (forbiddenp (+ a 1)) @dismiss (+ a 1)))
  ((a b) (if (< a b) @dismiss @lose)))

(defprim (?lesser! ?<!) (p a b)
  ((p) (if (or (= p 0) (= p 1)) @dismiss @lose))
  ((p &nogoodbeg) () (resolve-among '(0 1)))
  ((a &nogoodbeg) (b p)
   (let ((guess (if p (- b 1) b))) (if (forbiddenp guess) @dismiss guess)))
  ((b &nogoodbeg) (a p)
   (let ((guess (if p (+ a 1) a))) (if (forbiddenp guess) @dismiss guess)))
  (p (a b) (if (< a b) 1 0)))

(defprim (?lesser? ?<?) (p a b)
  ((p) (if (or (= p 0) (= p 1)) @dismiss @lose))
  ((p &nogoodbeg) () (resolve-among '(0 1)))
  ((a &nogoodbeg) (b p)
   (if (and p (not (forbiddenp (- b 1)))) (- b 1) @dismiss))
  ((b &nogoodbeg) (a p)
   (if (and p (not (forbiddenp (+ a 1)))) (+ a 1) @dismiss))
  (p (a b) (if (< a b) @dismiss 0)))
```

TABLE 6-43. Definition of Primitive Constraint-types (ii).

```
(defprim (?maxer ?max) (c a b)
  (c (a b) (max a b))
  ((c &nogoodbeg) (a) (if (forbiddenp a) a @dismiss))
  ((c &nogoodbeg) (b) (if (forbiddenp b) b @dismiss))
  (b (a c) (cond ((< a c) c)
                 ((> a c) @lose)
                 (t @dismiss)))
  (a (b c) (cond ((< b c) c)
                 ((> b c) @lose)
                 (t @dismiss))))

(defprim (?minner ?min) (c a b)
  (c (a b) (min a b))
  ((c &nogoodbeg) (a) (if (forbiddenp a) a @dismiss))
  ((c &nogoodbeg) (b) (if (forbiddenp b) b @dismiss))
  (b (a c) (cond ((> a c) c)
                 ((< a c) @lose)
                 (t @dismiss)))
  (a (b c) (cond ((> b c) c)
                 ((< b c) @lose)
                 (t @dismiss))))

(defprim signum (s a)
  ((s) (if (or (= s -1) (= s 0) (= s 1)) @dismiss @lose))
  ((s &nogoodbeg) () (resolve-among '(-1 0 1)))
  (a (s) (if (zerop s) 0 @dismiss))
  (s (a) (cond ((plusp a) 1) ((minusp a) -1) (t 0))))

(defprim (assumption assume) (pin)
  ((pin &nogoodbeg) () (if (forbiddenp *info*) @dismiss *info*)))

(defprim oneof (pin)
  ((pin &nogoodbeg) () (choose-from *info*))
  ((pin) (if (member pin *info*) @dismiss @lose)))

(defprim firstoneof (pin)
  ((pin &nogood) () (choose-from *info*)))
```

TABLE 6-44. Definition of Primitive Constraint-types (iii).

6.3.27. Primitive Constraints Are Uniformly Defined by `defprim`

In Chapter Five, the definition of most primitive constraints was done via the `defprim` construct, but the "strange" constraint-types `assumption`, `oneof`, and `firstoneof` were defined "manually", that is, by jerry-rigging the rule and constraint-type structures. Here we have a more general `defprim` that can accommodate rules which depend on nogood information.

The format of rule definitions was discussed in §6.3.13. Each rule specification has an optional output pin-name and keywords, a list of trigger pin-names, and a body. The definitions of `adder`, `multiplier`, `maxer`, `minner`, `equality`, and `gate` appear in Table 6-42. They are pretty

much as in previous versions, with three exceptions. One is that `setc` and `contradiction` are not used, but instead either an integer or one of the two flags `@lose` and `@dismiss` is computed. Another is that `equality` and `gate` each have a new rule, the second one. The first rule of each states that `p` must be 0 or 1, or else a contradiction occurs. The second rule says that if one of the two values 0 and 1 is forbidden by a nogood set, then the other one can be deduced and tentatively asserted. (The function `resolve-among` checks the nogood sets. It does not really do resolution—it merely checks whether one unique value of a set is possible, and if so asserts it; it is the value which will cause resolution if it fails.) The third exception is that if in place of the name is a list of two symbols, then the first is the name and the second is the `ctype-symbol` to be used by `tree-form` when extracting an algebraic expression from the network. If no separate `ctype-symbol` is supplied, the name is used. (This feature is primarily to make the output prettier. One might ask why the name is not always used, for if one wants adders to be called "+" in algebraic forms one could always just use the name `+` instead of `adder`. The answer is that the name is used for interacting with the LISP system, and the LISP system already uses the name `+` for something else. Thus this feature is a compromise with LISP, yielded in exchange for all the advantages using LISP provides.)

Table 6-43 defines some new primitive constraint-types. Definitions for them appeared in §6.1. A `lesser` device enforces a numerical less-than relationship between its two pins `a` and `b`—its rule signals a contradiction if this is not so. The `?lesser` device is similar, but also will try a heuristic guess at the value of one pin if the other is known. The first two rules are assumption (`&nogoodbeg`) rules which define the heuristic that in the absence of better information the two pins might as well have adjacent integers as values. This device is useful for expressing geometrical spacing constraints, for example. One might specify that one object must be somewhere to the left of (have an x-coordinate less than that of) another object; then in the absence of better information they will be right next to each other. The form `forbiddenp` is a predicate true iff the given value is disallowed by existing nogood sets.

The device types `lesser!` and `lesser?` are to `lesser` as `equality` and `gate` are to `==`, in effect. Type `lesser!` provides an extra pin `p` which specifies whether the `lesser` relationship between `a` and `p` is true or false. Type `lesser?` uses not a biconditional but an implication; if `p` is 1 then the `lesser` relationship holds, but if `p` is 0 the relationship may or may not hold. The device types `?lesser!` and /z?lesser! have the same relationships to the type `?lesser`. (I have found that having three such versions of almost any constraint-type is useful. A more advanced constraint language might just automatically provide every constraint-type with two extra pins `p?` and `p!` which are initially assumed to be 1. This would be one way for a constraint

```
(defun assume (value)
  (let ((a (gen-constraint assumption (gen-name 'assumption))))
    (setf (con-info a) value)
    (the pin a)))

(defun oneof (valuelist)
  (let ((a (gen-constraint oneof (gen-name 'oneof))))
    (setf (con-info a) valuelist)
    (the pin a)))

(defun firstoneof (valuelist)
  (let ((a (gen-constraint firstoneof (gen-name 'firstoneof))))
    (setf (con-info a) valuelist)
    (the pin a)))
```

TABLE 6-45. The `assume`, `oneof`, and `firstoneof` Constructs.

network to control itself—by turning the `p?` pin on and off to turn constraints on and off.[5])

The constraint-types `?maxer` and `?minner` (Table 6-44) are like `?lesser` – each is willing to make a guess on the basis of partial information. In this case, if one of `a` and `b` is known and the other not, then `c` is assumed to be the same as the known pin.

The constraint-type `signum` illustrates a situation where a pin is confined to a value set of more than two elements. The pin `s` must be one of -1, 0, or 1, reflecting the sign of the pin `a`. The first rule checks the value space; the second allows deduction of a value if the other two are forbidden; the third deduces $a = 0$ from $s = 0$; and the fourth is the obvious definition of `signum` as a function of `a`.

After these odd definitions, those of `assumption`, `oneof`, and `firstoneof` are not very surprizing. The one rule for `assumption` says that the rule need not be invoked unless the output pin has no value, in which case the assumed value is asserted unless forbidden. The first rule for `oneof` chooses among the possibilities and returns one (`choose-from` is like `resolve-among` except that it always returns some one choice or else performs resolution; `resolve-among` will fail to return a choice unless it is forced). The second rule checks that a value computed elsewhere is in its value set.

The one rule for `firstoneof` is a `&nogood` rules rather than a `&nogoodbeg` rule. That means that it will let a nogood set stop a value, but not the output pin. It chooses a value on the basis of nogood sets alone, and then returns it. If the output pin already has a value, it can jolly well cause a contradiction and create a nogood set, whereupon the rule, when run again, will then admit a different choice.

5. Luc Steels provides a similar facility in his constraint system [Steels 1980], where by convention every constraint has an extra "enable" pin. The name of this pin is the name of the constraint itself, and so he speaks of using the constarint itself as a value. I view the constraint and its enable pin as distinct things, and mean something else by using a constraint as a value. This is discussed in the Conclusions chapter.

The implementation of the `assume`, `oneof`, and `firstoneof` constructs is pretty much as in Chapter Five, except that the rules involved need not be explicitly awakened; the function `gen-constraint` (Table 6-11 (page 217)) takes care of that.

```
(defmacro defprim (namespec vars . rules)
  (let ((name (if (atom namespec) namespec (car namespec)))
        (symbol (if (atom namespec) namespec (cadr namespec))))
    `(progn 'compile
            (declare (special ,name))
            (setq ,name (make-constraint-type))
            (setf (ctype-name ,name) ',name)
            (setf (ctype-symbol ,name) ',symbol)
            (setf (ctype-vars ,name) (array-of ',vars))
            (setf (ctype-added-rules ,name) (array-n ,(length vars)))
            (setf (ctype-forget-rules ,name) (array-n ,(length vars)))
            (setf (ctype-nogood-rules ,name) (array-n ,(length vars)))
            (defmacro ,(symbolconc name "-VARNUM") (varname)
              `(posq ',varname ',',vars))
            (defmacro ,(symbolconc name "-BINDCELLS") body
              `(let ,',(forlist (var vars)
                         `(,(symbolconc var "-CELL")
                           (aref (con-values *me*) ,(posq var vars))))
                 ,@body))
            ,@(do ((r rules (cdr r))
                   (bit 1 (lsh bit 1))
                   (defs '() (cons `(defrule ,name ,bit
                                      ,@(if (null (cddr (car r)))
                                            (cons () (car r))
                                            (car r)))
                                   defs)))
                  ((null r) defs))
            '(,name primitive))))
```

TABLE 6-46. Definition of Primitives.

```
(defprim gate (p a b)
  ((p) (if (or (= p 0) (= p 1)) @dismiss @lose))
  ((p &nogoodbeg) () (resolve-among '(0 1)))
  (p (a b) (if (= a b) @dismiss 0))
  (b (p a) (if (= p 1) a @dismiss))
  (a (p b) (if (= p 1) b @dismiss)))
```

expands into:

```
(progn 'compile
       (declare (special gate))
       (setq gate (make-constraint-type))
       (setf (ctype-name gate) 'gate)
       (setf (ctype-symbol gate) 'gate)
       (setf (ctype-vars gate) (array-of '(p a b)))
       (setf (ctype-added-rules gate) (array-n 3))
       (setf (ctype-forget-rules gate) (array-n 3))
       (setf (ctype-nogood-rules gate) (array-n 3))
       (defmacro gate-varnum (varname)
           `(posq ',varname '(p a b)))
       (defmacro gate-bindcells body
           `(let ((p-cell (aref (con-values *me*) 0))
                  (a-cell (aref (con-values *me*) 1))
                  (b-cell (aref (con-values *me*) 2)))
              ,@body))
       (defrule gate 16 a (p b) (if (= p 1) b @dismiss))
       (defrule gate 8 b (p a) (if (= p 1) a @dismiss))
       (defrule gate 4 p (a b) (if (= a b) @dismiss 0))
       (defrule gate 2 (p &nogoodbeg) () (resolve-among '(0 1)))
       (defrule gate 1 () (p) (if (or (= p 0) (= p 1)) @dismiss @lose))
       '(gate primitive))
```

TABLE 6-47. Expansion of the Definition of `gate`.

Table 6-46 shows the new definition of the LISP macro `defprim`. Among other things, it assigns id-bits to each of the rules (each id-bit is a distinct power of two). It also defines two macros *name*-`varnum` and *name*-`bindcell`, references to which will be generated by `defrule`. One converts a pin-name into a pin-number, and the other generates the binding of names of the form *pin-name*-`cell` to the corresponding cells, which is done in every rule. Using macros in this way instead of a global data base causes the information to be transmitted correctly at either LISP compile time or LISP interpretation time. Table 6-47 shows the LISP code into which the `defprim` definition of `gate` expands.

```
(defmacro defrule (typename bit output-stuff trigger-names body)
  (let ((rulename (gen-name typename 'rule))
        (ctype (symeval typename))
        (output-name (cond ((null output-stuff) ())
                           ((atom output-stuff) output-stuff)
                           (t (car output-stuff))))
        (keywords (cond ((atom output-stuff) ())
                        (t (cdr output-stuff)))))
    (require-constraint-type ctype)
    `(progn 'compile
            (declare (special ,rulename))
            (defun ,rulename (*me*)
              (let ((*rule* ,rulename)
                    (*info* (con-info *me*)))
                (,(symbolconc typename "-BINDCELLS")
                  (let ((*outvar* ,(if output-name
                                       (symbolconc output-name "-CELL")
                                       ())))
                       ,@(forlist (var trigger-names)
                           `(,var (cell-value ,(symbolconc var "-CELL")))))
                    ,body))))
            (let ((rule (make-rule)))
              (setq ,rulename rule)
              (setf (rule-code rule) ',rulename)
              (setf (rule-ctype rule) ,typename)
              (setf (rule-outvar rule)
                    ,(if output-name
                         `(,(symbolconc typename "-VARNUM") ,output-name)
                         ()))
              (setf (rule-triggers rule)
                    (list ,@(forlist (var trigger-names)
                              `(,(symbolconc typename "-VARNUM") ,var))))
              (setf (rule-bits rule)
                    ,(+ (if (memq '&nogood keywords) @rule-nogood 0)
                        (if (memq '&nogoodbeg keywords) @rule-nogoodbeg 0)))
              (setf (rule-id-bit rule) ,bit)
              ,@(and output-name
                     `((push rule (aref (ctype-forget-rules ,typename)
                                        (,(symbolconc typename "-VARNUM")
                                         ,output-name)))))
              ,@(forlist (var trigger-names)
                  `(push rule (aref (ctype-added-rules ,typename)
                                    (,(symbolconc typename "-VARNUM") ,var))))
              ,@(and (or (memq '&nogood keywords) (memq '&nogoodbeg keywords))
                     `((push rule (aref (ctype-nogood-rules ,typename)
                                        (,(symbolconc typename "-VARNUM")
                                         ,output-name)))))
              '(,typename rule))))
```

TABLE 6-48. Definition of Rules.

Table 6-48 shows the new definition of the LISP macro `defrule`. It arranges to create the rule data structure and catalogue it in the constraint-type's rules tables.

```
(defrule gate 2 (p &nogoodbeg) () (resolve-among '(0 1)))
```

expands into:

```
(progn 'compile
       (declare (special gate-rule-69))
       (defun gate-rule-69 (*me*)
           (let ((*rule* gate-rule-69)
                 (*info* (con-info *me*)))
             (gate-bindcells (let ((*outvar* p-cell)) (resolve-among '(0 1))))))
       (let ((rule (make-rule)))
         (setq gate-rule-69 rule)
         (setf (rule-code rule) 'gate-rule-69)
         (setf (rule-ctype rule) gate)
         (setf (rule-outvar rule) (gate-varnum p))
         (setf (rule-triggers rule) (list))
         (setf (rule-bits rule) 2)
         (setf (rule-id-bit rule) 2)
         (push rule (aref (ctype-forget-rules gate) (gate-varnum p)))
         (push rule (aref (ctype-nogood-rules gate) (gate-varnum p))))
       '(gate rule))
```

```
(defrule gate 1 () (p) (if (or (= p 0) (= p 1)) @dismiss @lose))
```

expands into:

```
(progn 'compile
       (declare (special gate-rule-70))
       (defun gate-rule-70 (*me*)
         (let ((*rule* gate-rule-70)
               (*info* (con-info *me*)))
           (let ((p-cell (aref (con-values *me*) 0))
                 (a-cell (aref (con-values *me*) 1))
                 (b-cell (aref (con-values *me*) 2)))
             (let ((*outvar* nil)
                   (p (cell-value p-cell)))
               (if (or (= p 0) (= p 1)) @dismiss @lose)))))
       (let ((rule (make-rule)))
         (setq gate-rule-70 rule)
         (setf (rule-code rule) 'gate-rule-70)
         (setf (rule-ctype rule) gate)
         (setf (rule-outvar rule) nil)
         (setf (rule-triggers rule) (list (posq 'p '(p a b))))
         (setf (rule-bits rule) 0)
         (setf (rule-id-bit rule) 1)
         (push rule (aref (ctype-added-rules gate) (posq 'p '(p a b)))))
       '(gate rule))
```

TABLE 6-49. Expansions of the Definitions of Two `gate` Rules.

```
(defmacro forbiddenp (val) `(*forbiddenp ,val *outvar*))

(statistics-counter forbiddenp-sets "Number of nogood sets checked")
(statistics-counter forbiddenp-pairs "Number of nogood set pairs checked")

(defun *forbiddenp (val *outvar*)
  (do-named outer-loop
            ((x (cdr (assoc val (node-nogoods *outvar*))) (cdr x)))
            ((null x) ())
    (statistic forbiddenp-sets)
    (do-named inner-loop
              ((c (cdar x) (cdr c)))
              ((null c)
               (return-from outer-loop (car x)))
      (statistic forbiddenp-pairs)
      (and (not (eq (caar c) (cell-repository *outvar*)))
           (or (not (eq (cell-state (rep-supplier (caar c))) @king))
               (not (equal (cell-contents (rep-supplier (caar c))) (cdar c))))
           (return-from inner-loop)))))
```

TABLE 6-50. Checking Whether a Value is Forbidden by a Nogood Set.

Table 6-49 shows the expansions of two of the rules for the `gate` constraint-type. The first one does not have the occurrences of `gate-bindcells` and `gate-varnum` expanded, and the second one does. (The LISP function `posq` treats its second argument, a list, as a zero-origin array, and returns the index into that array of the first occurrence of its first argument, using an `eq` test.)

6.3.28. Checking the Nogood Sets Can Advise Rules about Forbidden Values

The utility macro `forbiddenp` used by many of the primitive's rule definitions is defined in Table 6-50. It calls the function `*forbiddenp` on the specified value and the cell for the output pin of the rule. (This is yet another example of providing a function for internal use and a macro that makes the interface pretty in common situations (rule definitions in this case).) It is essentially the check in the old code for `assumption-rule` in Table 5-2 (page 143). If a nogood set can be found that forbids the old value, it is a "killer" and is returned; if none is found then `()` is returned.

In a similar manner the macros `choose-from` and `resolve-among` interface to functions named `*choose-from` and `*resolve-among`. Each of them is based on the outer loop of the old `oneof-rule` in Table 5-3 (page 144). Each of them tests elements from the list of possibilities using `forbiddenp`, and for each forbidden value adds the killer to an accumulating list. Each of them signals a contradiction of none of the possibilities work (and then returns `@dismiss`, *not* `@lose`—the contradiction is enqueued by `signal-nochoice`, and it is not correct to blame this contradiction on the rule which invoked a macro, because it is a `@resolution`-type contradiction). The difference between them is that if a valid possibility is found `*choose-from` returns it

immediately, whereas `*resolve-among` checks them all, and returns `@dismiss` unless there is a unique choice.

```
(defmacro choose-from (choices)
  `(*choose-from ,choices *outvar*))

(defun *choose-from (choices *outvar*)
  (do ((v choices (cdr v))
       (killers '()))
      ((null v)
       (signal-nochoice choices *outvar* killers)
       @dismiss)
    (let ((ng (forbiddenp (car v))))
      (if ng (push ng killers) (return (car v))))))

(defun signal-nochoice (choices *outvar* killers)
  (ctrace "All of the values ~S for ~S are no good."
          choices
          (cell-goodname *outvar*))
  (let ((losers '()))
    (dolist (killer killers)
      (dolist (x (cdr killer))
        (or (eq (car x) (cell-repository *outvar*))
            (or (assq (rep-supplier (car x)) losers)
                (push (cons (rep-supplier (car x)) (cdr x)) losers)))))
    (enqueue (cons @resolution losers) *contra-queue*)))

(defmacro resolve-among (choices)
  `(*resolve-among ,choices *outvar*))

(defun *resolve-among (choices *outvar*)
  (do ((v choices (cdr v))
       (winners '())
       (killers '()))
      ((null v)
       (cond ((null winners)
              (signal-nochoice choices *outvar* killers)
              @dismiss)
             ((null (cdr winners)) (car winners))
             (t @dismiss)))
    (let ((ng (forbiddenp (car v))))
      (if ng (push ng killers) (push (car v) winners)))))
```

TABLE 6-51. Filtering a Set of Possibilities Using Nogood Sets.

The functions `choose-from` and `resolve` appear in Table 6-51. So does the function `signal-nochoice`, which performs the resolution step on a set of killer nogood sets. It produces a new resolvent set, and enqueues a `@resolution`-type contradiction task.

```
(defun why (cell)
  (require-cell cell)
  (cond ((not (node-boundp cell))
         (format t "~%;~S has no value." (cell-id cell))
         (let ((flag ()))
           (dolist (c (node-cells cell))
             (and (cell-owner c)
                  (dolist (rule (aref (ctype-forget-rules
                                       (con-ctype (cell-owner c)))
                                      (cell-name c)))
                    (let ((trigger-names
                           (forlist (tr (rule-triggers rule))
                             (aref (ctype-vars (con-ctype (cell-owner c))) tr))))
                      (format t "~:[  I could compute it~;~
                                  ; or~]"
                              flag)
                      (setq flag t)
                      (format t "~%;   from ~:[~2*~;pin~P ~{~S~^, ~} of ~]~
                                  ~S by rule ~S"
                              trigger-names
                              (length trigger-names)
                              trigger-names
                              (con-name (cell-owner c))
                              rule)))))
           (format t "~:[  I don't have any way to compute it.~;.~]" flag)))
        (t (format t "~%;The value ~S is in ~S because "
                   (cell-value cell) (cell-goodname cell))
           (select! (cell-state cell)
             ((@king @friend @rebel)
              (why-how cell))
             ((@slave)
              (format t "it is connected to ~S~%;   and "
                      (cell-goodname (node-supplier cell)))
              (why-how (node-supplier cell)))
             ((@dupe)
              (format t "it is connected to ~S~%;   and "
                      (cell-goodname (cell-contents cell)))
              (why-how (cell-contents cell)))
             ((@puppet) (lose "Bound node has a @PUPPET cell ~S." cell)))))
  'q.e.d.)
```

TABLE 6-52. Implementation of the `why` Function.

6.3.29. The `why` Function Prints Values Forbidden by Nogood Sets

The new definition of the `why` function appears in Table 6-52. As before, it divides into two cases depending on whether or not the given node has a value. If it does, then it dispatches on the cell-state of the given cell to determine just how it got its value. Note that `why` and all the other explanation functions are carefully written to be useful on contradictory networks—they handle rebels and dupes properly. They don't require consistency, merely well-foundedness.

```
(defun why-how (s)
  (if (null (cell-owner s))
      (format t "that is a constant.")
      (format t "~<~%;   ~0,72:;~S computed it~>~<~%;  ~0,72:; using rule ~S~>~
                          ~@[~%;   from: ~:{~S (~S)~:[~*~; = ~S~]~:^, ~}~]."
              (cell-owner s)
              (cell-rule s)
              (forlist (tr (rule-triggers (cell-rule s)))
                (let ((cell (aref (con-ctype (cell-owner s)) tr)))
                  (list (cell-id cell)
                        (aref (ctype-vars (con-values (cell-owner s))) tr)
                        (node-boundp cell)
                        (cell-value cell))))))
  (print-forbidden-values s))

(defun print-forbidden-values (s)
  (and (node-boundp s)
       (or (bit-test @rule-nogood (rule-bits (cell-rule s)))
           (bit-test @rule-nogoodbeg (rule-bits (cell-rule s))))
       (format t "~@[~%;Nogood sets currently forbid these values: ~{~S~^,~}.~]"
               (mapcan #'(lambda (x) (and (*forbiddenp (car x) s) (list (car x))))
                       (node-nogoods s)))))

(defun cell-goodname (cell)
  (require-cell cell)
  (cond ((globalp cell) (cell-name cell))
        ((or (eq (cell-rule cell) *constant-rule*)
             (eq (cell-rule cell) *default-rule*)
             (eq (cell-rule cell) *parameter-rule*))
         (list (cell-name cell) (cell-contents cell)))
        (t (list 'the (aref (ctype-vars (con-ctype (cell-owner cell)))
                            (cell-name cell))
                 (con-name (cell-owner cell))))))
```

TABLE 6-53. Explaining a True-Supplier, and Printing Forbidden Values.

The function `why-how` (Table 6-53) prints the constraint, rule, and triggers that were responsible for a computed value, and then calls `print-forbidden-values` to check for any values that are forbidden by nogood sets. Of course, the set of all possible values is infinite, being all the integers, but `print-forbidden-values` simply maps over the buckets of the nogoods component of a node, and for each bucket value checks to see whether it is forbidden. After all, a value cannot be forbidden if there is no bucket to hold a killer for it!

The function `cell-goodname` tries to pick a pretty name for a cell for printing purposes. This version never uses the cell-id, which the user isn't ever supposed to see anyway.

As an example, consider this interaction (with tracing turned off):

```
(test)                              ;set up a temperature conversion network
DONE
(== fahrenheit (default -40))
DONE
(why centigrade)
;The value -40 is in CENTIGRADE because it is connected to (THE B MULT)
;   and <MULT:MULTIPLIER> computed it using rule <B←MULTIPLIER-RULE-5(A,C)>
;   from: CELL-271 (A) = 9, CELL-269 (C) = -360.
Q.E.D.
```

As another example, suppose that the network for the four queens problem of §5.4.2 has been run in the new system.

```
(why q0)
;The value 1 is in Q0 because it is connected to (THE PIN ONEOF-250)
;   and <ONEOF-250:ONEOF> computed it
;   using rule <(PIN &NOGOODBEG)←ONEOF-RULE-44()>.
;Nogood sets currently forbid these values: 0.
Q.E.D.
(why q1)
;The value 3 is in Q1 because it is connected to (THE PIN ONEOF-253)
;   and <ONEOF-253:ONEOF> computed it
;   using rule <(PIN &NOGOODBEG)←ONEOF-RULE-44()>.
;Nogood sets currently forbid these values: 0,1,2.
Q.E.D.
```

Of course, a value is forbidden for `q`*n* only when it is assumed that all the other `q`*i* are fixed.

```
(defun why-ultimately (cell)
  (require-cell cell)
  (cond ((not (node-boundp cell))
         (format t "~%;~S has no value." (cell-id cell))
         (format t "~@[  Perhaps knowing the value of ~
                    ~{~<~%;  ~:15,72;~S ~>~^or ~}would help.~]"
                 (forlist (c (delq cell (desired-premises cell))) (cell-name c))))
        (t (format t "~%;The value ~S is in ~S because "
                   (cell-value cell) (cell-goodname cell))
           (select! (cell-state cell)
             ((@king @friend @rebel) (why-ultimately-how cell cell))
             ((@slave)
              (format t "it is connected to ~S~%;   and "
                      (cell-goodname (node-supplier cell)))
              (why-ultimately-how cell (node-supplier cell)))
             ((@dupe)
              (format t "it is connected to ~S~%;   and " (cell-contents cell))
              (why-ultimately-how cell (cell-goodname (cell-contents cell))))
             ((@puppet) (lose "Bound node has a @PUPPET cell ~S." cell))))))
  'q.e.d.)

(defun why-ultimately-how (cell s)
  (if (null (cell-owner s))
      (format t "that is a constant.")
      (multiple-value-bind (premises defaults params nogoods trees links)
          (fast-premises cell)
        (format t "it was ultimately derived~
                   ~@[ from:~:{~%;   ~S~@ == ~S~~:^,~}~].~
                   ~@[~%;These connections were involved:~
                          ~:{~%;   ~S == ~S~:^,~}.~]"
                (forlist (p premises)
                  (cons p (mapcan #'(lambda (c)
                                      (and (globalp c) (list (cell-name c))))
                                  (node-cells p))))
                (forlist (l links)
                  (list (cell-goodname (car l)) (cell-goodname (cdr l))))))))
```

TABLE 6-54. Implementation of `why-ultimately`.

6.3.30. The `why-ultimately` Function Prints Cell-Link Information

The function `why-ultimately` (Table 6-54) has been split into two functions in the same way that `why` was. The function `why-ultimately-how` prints not only the premises which support the quantity asked about, but also all the equating connections traversed by the computation (using the `links` information computed by `premises`). As an example, consider this explanation for the temperature conversion network:

```
(test)                          ;set up a temperature conversion network
DONE
(== fahrenheit (default -40))
```

```
(defun desired-premises (cell)
  (require-cell cell)
  (prog1 (desired-premises-mark cell) (desired-premises-unmark cell)))

(defun desired-premises-mark (cell)
  (require-cell cell)
  (cond ((and (not (node-boundp cell))
              (not (markp cell)))
         (mark-node cell)
         (do ((c (node-cells cell) (cdr c))
              (p '() (nconc (if (null (cell-owner (car c)))
                                (and (globalp (car c)) (list (car c)))
                                (desired-premises-constraint (car c)))
                            p)))
             ((null c) p)))))

(defun desired-premises-constraint (cell)
  (require-cell cell)
  (let ((p '()))
    (dolist (rule (aref (ctype-forget-rules (con-ctype (cell-owner cell)))
                        (cell-name cell)))
      (dolist (tr (rule-triggers rule))
        (setq p (nconc (desired-premises-mark
                        (aref (con-values (cell-owner cell)) tr))
                       p))))
    p))

(defun desired-premises-unmark (cell)
  (require-cell cell)
  (cond ((markp cell)
         (unmark-node cell)
         (dolist (c (node-cells cell))
           (and (cell-owner c)
                (doarray (pin (con-values (cell-owner c)))
                  (desired-premises-unmark pin)))))))
```

TABLE 6-55. Locating Desired Premises for an Unbound Cell.

```
DONE
(why-ultimately centigrade)
;The value -40 is in CENTIGRADE because it is connected to (THE B MULT)
;   and it was ultimately derived from:
;   <CELL-292 (DEFAULT-290) KING -40> == FAHRENHEIT.
;These connections were involved:
;   (THE B OTHERMULT) == (CONSTANT 5),
;   FAHRENHEIT == (DEFAULT-290 -40),
;   (THE C ADD) == FAHRENHEIT,
;   (THE B ADD) == (CONSTANT 32),
;   (THE A OTHERMULT) == (THE A ADD),
;   (THE C MULT) == (THE C OTHERMULT),
;   (THE A MULT) == (CONSTANT 9),
;   CENTIGRADE == (THE B MULT).
Q.E.D.
```

```
(defun what (cell)
  (require-cell cell)
  (cond ((not (node-boundp cell))
         (format t "~%;~S has no value.  I can express it in this way:~
                    ~:~%;    ~S = ~S~"
                 (cell-id cell) (tree-form cell t)))
        (t (format t "~%;The value ~S in ~S was computed in this way:~
                    ~:~%;    ~S ← ~S~"
                   (cell-value cell) (cell-goodname cell) (tree-form cell))))
  (print-forbidden-values (cell-true-supplier cell))
  'okay?)

(defprop assumption disliked treeformpref)
(defprop oneof disliked treeformpref)
(defprop firstoneof disliked treeformpref)

(defmacro nummark (cell)
  `(setf (cell-mark ,cell)
         (if (numberp (cell-mark ,cell)) (+ (cell-mark ,cell) 1) 1)))
(defmacro unnummark (cell) `(setf (cell-mark ,cell) ()))
(defmacro nummarkp (cell) `(numberp (cell-mark ,cell)))
(defmacro singlenummarkp (cell) `(equal (cell-mark ,cell) 1))

(defun tree-form (cell &optional (shallow ()))
  (require-cell cell)
  (nummark (cell-true-supplier cell))
  (prog2 (tree-form-trace cell shallow)
         (tree-form-gather cell shallow)
         (tree-form-unmark cell)))
```

TABLE 6-56. Implementation of `what`.

The code for `desired-premises` (Table 6-55) is essentially as before, with minor modifications for the new data structures involved.

6.3.31. The `what` Function Uses the Generalized Algebraic Form

The only change to the function `what` (Table 6-56) is the addition of a call to the function `print-forbidden-values` (defined in Table 6-53). All of the interesting changes are in the function `tree-form` and its cohorts.

I couldn't resist using property lists for *something* (this is LISP code, after all!), and so the property `treeformpref` on the name fo a constraint-type indicates whether that type of constraint may be used to express the value of a cell. Possible values are `disliked` and `forbidden`, though `forbidden` isn't used here. Types `assumption`, `oneof`, and `firstoneof` are disliked; they are not used in algebraic expressions if there is any better alternative.

```
(defun tree-form-trace-set (owner names shallow)
  (require-constraint owner)
  (do ((n names (cdr n))
       (queue '() (nconc (tree-form-tag (aref (con-values owner) (car n))) queue)))
      ((null n) (dolist (c queue) (tree-form-trace c shallow)))))

(defun tree-form-tag (cell)
  (require-cell cell)
  (let ((s (cell-true-supplier cell)))
    (and (not (progl (nummarkp s) (nummark s)))
         (list cell))))

(defun tree-form-trace (cell shallow)
  (require-cell cell)
  (cond ((node-boundp cell)
         (let ((s (cell-true-supplier cell)))
           (cond ((cell-owner s)
                  (or shallow
                      (tree-form-trace-set (cell-owner s)
                                           (rule-triggers (cell-rule s))
                                           shallow)))
                 (t (nummark s)))))                                    ;crock
        (t (let ((cells (node-cells cell)))
             (usurper (or (if shallow
                              (or (tree-form-shallow cell cells)
                                  (tree-form-deep cell cells shallow))
                              (or (tree-form-deep cell cells shallow)
                                  (tree-form-shallow cell cells)))
                          (progn (and (cell-owner cell)
                                      (tree-form-trace-set
                                          (cell-owner cell)
                                          (fortimes (j (array-length
                                                         (con-values
                                                             (cell-owner cell))))
                                            j)
                                          shallow))
                                 cell)))))))
```

TABLE 6-57. Tracing Out an Algebraic Expression in the Network.

The code for tracing out an expression (Table 6-57) is not changed much. It operates on the true-supplier of the given cell if it is bound. If the node has no value, then any supplier will do, and the existing puppet could be used. However, `tree-form-trace` tries as before to choose a "good" artificial supplier and lets it usurp the existing puppet. (It is somewhat of a "no-no" to have a probing utility such as `what` alter the network being probed—it violates the principle that debugging tools should avoid altering the object being debugged in unpredictable ways. This is a very tiny violation, though. If the node structure is sound, it doesn't matter which cell is the puppet.)

```
(defun tree-form-shallow (cell cells)
  (do ((c cells (cdr c)))
      ((null c) ())
    (and (not (eq (car c) cell))
         (globalp (car c))
         (return (car c)))))

(defun tree-form-deep (cell cells shallow)
  (do ((z cells (cdr z))
       (any ()))
      ((null z) any)
    (and (not (eq (car z) cell))
         (cell-owner (car z))
         (let ((pref (get (ctype-name (con-ctype (cell-owner (car z))))
                          'treeformpref)))
           (cond ((eq pref 'dislike) (setq any (car z)))
                 ((not (eq pref 'forbidden))
                  (tree-form-trace-set
                      (cell-owner (car z))
                      (fortimes (j (array-length (con-values (cell-owner (car z)))))
                        j)
                      shallow)
                  (return (car z))))))))
```

TABLE 6-58. Determining a "Good" Artificial Supplier.

The new version of `tree-form-deep` (Table 6-58) uses the preferences expressed by the `treeformpref` property to avoid using a disliked constraint-type in an algebraic expression. A forbidden one is never, ever used; a disliked one is used only if there aren't any others.

```
(declare (special *cuts* *allcuts* *extra-equations*))

(defun tree-form-gather (cell shallow)
  (require-cell cell)
  (do ((*cuts* (list cell))
       (*allcuts* (list cell))
       (equations '())
       (*extra-equations* '()))
      ((null *cuts*) (nreverse (append *extra-equations* equations)))
    (let ((cut (pop *cuts*)))
      (push (list (cell-goodname cut) (tree-form-chase cut shallow t))
            equations))))

(defun tree-form-chase (cell shallow top)
  (require-cell cell)
  (let ((s (cell-true-supplier cell)))
    (cond ((and shallow (node-boundp cell)) (cell-value cell))
          ((and (not top) (not (singlenummarkp s)))
           (cond ((and (null (cell-owner s))
                       (not (null (cell-rule s))))
                  (do ((c (node-cells s) (cdr c)))
                      ((null c) (cell-contents s))
                    (cond ((good-global (car c) s)
                           (cond ((not (memq (car c) *allcuts*))
                                  (push (car c) *allcuts*)
                                  (push (list (cell-name (car c)) (cell-contents s))
                                        *extra-equations*)))
                                  (return (cell-name (car c)))))))
                 (t (let ((best (do ((c (node-cells s) (cdr c)))
                                    ((null c) s)
                                  (and (good-global (car c) s)
                                       (return (car c))))))
                      (cond ((and (not (and (eq best s) (globalp s)))
                                  (not (memq best *allcuts*)))
                             (push best *allcuts*)
                             (push best *cuts*)))
                      (cell-goodname best)))))
          ((cell-owner s)
           (cond ((and (eq s cell) (not top)) (cell-goodname s))
                 (t (let ((args (forarray (v (con-values (cell-owner s)))
                                  (cond ((eq v s) '%)
                                        ((and (node-boundp s)
                                              (not (member (cell-name v)
                                                           (rule-triggers
                                                               (cell-rule s)))))
                                         '?)
                                        (t (tree-form-chase v shallow ()))))))
                      (nconc (list (ctype-symbol (con-ctype (cell-owner s))))
                             (if (eq (car args) '%) (cdr args) args)
                             (and (con-info (cell-owner s))
                                  (list (con-info (cell-owner s)))))))))
          ((globalp s) (cell-name s))                ;???
          (t (cell-contents s)))))
```

TABLE 6-59. Constructing the Traced-out Algebraic Expression.

```
(defun good-global (c s)                          ;is c a good global for naming s?
  (and (not (eq c s))
       (globalp c)
       (or (and (or (eq (cell-state s) @king)
                    (eq (cell-state s) @puppet)
                    (eq (cell-state s) @slave)
                    (eq (cell-state s) @friend))
                (eq (cell-state c) @slave))
           (and (eq (cell-state s) @rebel)
                (eq (cell-state c) @dupe)
                (eq (cell-contents c) s))
           (and (eq (cell-state s) @dupe)
                (eq (cell-state c) @dupe)
                (eq (cell-contents c) (cell-contents s))))))

(defun tree-form-unmark (cell)
  (require-cell cell)
  (let ((s (cell-true-supplier cell)))
    (cond ((nummarkp s)
           (unnummark s)
           (and (cell-owner s)
                (doarray (pin (con-values (cell-owner s)))
                  (tree-form-unmark pin)))))))
```

TABLE 6-60. Checking for a Good Global Name, and Unmarking, for `tree-form`.

The new version of `tree-form-chase` (Table 6-59) uses the % convention—as it translates the pins of some constraint, it notes which one was the output pin and substitutes a % for it. Also, for any pin that was not a trigger of the rule that computed the value, if any, it substitutes a ?, as before. When constructing the final form, if the first argument expression is % it is omitted.

The predicate `good-global` (Table 6-60) takes a two cells of a node, and is true if the first is thought to be a good choice as a name for the second. For this to be the case it must be different from the second, must be the cell for a global variable, and must take its value from the same place the second one does. (It might seem that the clause `(eq (cell-state c) @slave)` ought rather to be

```
(or (eq (cell-state c) @slave)
    (eq (cell-state c) @friend))
```

—however, a global variable never has its own value, and so can never be a friend.)

The function `tree-form-unmark` runs around as before, resetting all the marks.

As an example of `what`, consider this interaction:

```
(test)                          ;set up temperature conversion network
DONE
(== fahrenheit (parameter -40))
DONE
```

```
(what centigrade)
;The value -40 in CENTIGRADE was computed in this way:
;   CENTIGRADE ← (* (* (+ FAHRENHEIT % 32) 5) 9 %)
;   FAHRENHEIT ← -40
OKAY?
```

This algebraic form may be unfamiliar, but it correctly conveys the network structure used to compute the value.

```
(== fahrenheit (default 32))
;;; These are the premises that seem to be at fault:
;       <CELL-415 (DEFAULT-413) [OPPOSED] KING 32>,
;       <CELL-412 (PARAMETER-410) REBEL -40 AGAINST 32> == FAHRENHEIT.
;;; Choose one of these to retract and RETURN it.
;BKPT Choose Culprit
(return fahrenheit)                ;uniquely identifies the culprit
DONE
```

Note that returning `fahrenheit` uniquely identifies the culprit even though it and both premises are all in the same node (cf. §6.3.20 and Table 6-27 (page 243)). It could also have been specified by `(return parameter-410)`.

```
(what centigrade)
;The value 0 in CENTIGRADE was computed in this way:
;   CENTIGRADE ← (* (* (+ FAHRENHEIT % 32) ?) 9 %)
;   FAHRENHEIT ← 32
OKAY?
```

Because 32 and "%" sum to `fahrenheit` (which is also 32), therefore "%" is zero, and so the other operand to the inner multiplication was not a trigger for the product. This other operand is therefore represented as a "?".

6.4. The New Improved Example

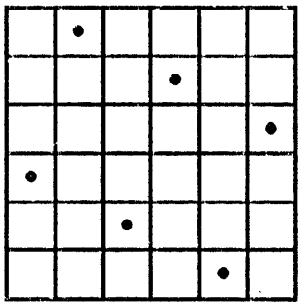
One advantage of using the queue-based control structure is that pending computations are stored explicitly as data structures rather than implicitly in the host-language control stack, which in many cases is of a finite size much smaller than the heap area—this is the case for many LISP implementations, including Lisp Machine LISP. Using the system of Part One, it was not possible to run the N queens problem for $N = 6$ because the LISP system stack would overflow. There is no problem with the current queue-based system, however.

The new language has the `disallow` construct, which allows cycling though a set of possibilities. In this example we will obtain all four solutions for the six queens problem. At each step we will ask for the statistics also.

```
(sixqueen)                                  ;start up the six queens problem
DONE                                          ;after much computation!
```

The LISP function `sixqueen` merely creates a large number of constraints analogous to those in Table 5-19 (page 166), Table 5-20 (page 166), Table 5-21 (page 167), and Table 5-22 (page 167).

```
(list q0 q1 q2 q3 q4 q5)
(<CELL-49 (Q0) SLAVE 1>
 <CELL-51 (Q1) SLAVE 3>
 <CELL-53 (Q2) SLAVE 5>
 <CELL-55 (Q3) SLAVE 0>
 <CELL-57 (Q4) SLAVE 2>
 <CELL-59 (Q5) SLAVE 4>)
```

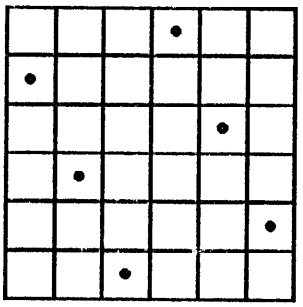


```
(stats)
;     248 = Repositories generated
;     248 = Cells generated
;       0 = Initialized cells
;      81 = Constraints generated
;    3895 = Iterations of top-level-loop queue scan
;    3742 = Rules enqueued
;    3156 = Added rules enqueued
;      51 = Forget rules enqueued
;     535 = Nogood rules enqueued
;    3495 = Attempts to run a rule
;    2785 = Successfully run rules
;    1291 = Rule runs which dismissed
;       0 = Rules which overrode other rules
;     143 = Rules which superseded other rules
;      30 = Usurpations
;     124 = Contradictions dequeued for processing
;     103 = @NODE contradictions dequeued for processing
;       0 = @CONSTRAINT contradictions dequeued for processing
;      21 = @RESOLUTION contradictions dequeued for processing
;     124 = Contradictions actually processed
```

```
;    124 = Nogood culprits automatically chosen
;    124 = Nogood sets installed
;    201 = Number of calls to ==
;  12547 = Awakenings
;   2546 = @ADDED awakenings
;      0 = @FORGET awakenings
;   9336 = @NOGOOD awakenings
;   1270 = Values forgotten
;   3778 = Number of nogood sets checked
;   5706 = Number of nogood set pairs checked
NIL
```

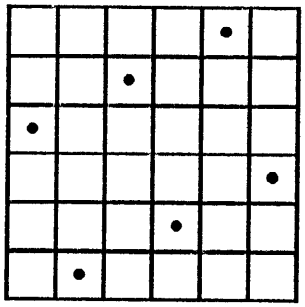
There have been 103 ordinary contradictions and 21 resolutions. Now we will disallow this particular solution, and force a search for another.

```
(disallow q0 q1 q2 q3 q4 q5)
DONE
(list q0 q1 q2 q3 q4 q5)
(<CELL-49 (Q0) SLAVE 3>
 <CELL-51 (Q1) SLAVE 0>
 <CELL-53 (Q2) SLAVE 4>
 <CELL-55 (Q3) SLAVE 1>
 <CELL-57 (Q4) SLAVE 5>
 <CELL-59 (Q5) SLAVE 2>)
```



```
(stats)
;    248 = Repositories generated
;    248 = Cells generated
;      0 = Initialized cells
;     81 = Constraints generated
;   5294 = Iterations of top-level-loop queue scan
;   5278 = Rules enqueued
;   4318 = Added rules enqueued
;     51 = Forget rules enqueued
;    909 = Nogood rules enqueued
```

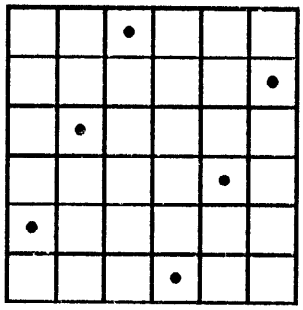
```
;   4847 = Attempts to run a rule
;   4046 = Successfully run rules
;   1800 = Rule runs which dismissed
;      0 = Rules which overrode other rules
;    267 = Rules which superseded other rules
;     30 = Usurpations
;    170 = Contradictions dequeued for processing
;    128 = @NODE contradictions dequeued for processing
;      0 = @CONSTRAINT contradictions dequeued for processing
;     42 = @RESOLUTION contradictions dequeued for processing
;    170 = Contradictions actually processed
;    170 = Nogood culprits automatically chosen
;    170 = Nogood sets installed
;    201 = Number of calls to ==
;  20332 = Awakenings
;   3483 = @ADDED awakenings
;      0 = @FORGET awakenings
;  15764 = @NOGOOD awakenings
;   1898 = Values forgotten
;   8913 = Number of nogood sets checked
;  13603 = Number of nogood set pairs checked
NIL


(disallow q0 q1 q2 q3 q4 q5)
DONE
(list q0 q1 q2 q3 q4 q5)
(<CELL-49 (Q0) SLAVE 4>
 <CELL-51 (Q1) SLAVE 2>
 <CELL-53 (Q2) SLAVE 0>
 <CELL-55 (Q3) SLAVE 5>
 <CELL-57 (Q4) SLAVE 3>
 <CELL-59 (Q5) SLAVE 1>)
```



```
(stats)
;    248 = Repositories generated
```

```
;    248 = Cells generated
;      0 = Initialized cells
;     81 = Constraints generated
;   7159 = Iterations of top-level-loop queue scan
;   7404 = Rules enqueued
;   5876 = Added rules enqueued
;     51 = Forget rules enqueued
;   1477 = Nogood rules enqueued
;   6652 = Attempts to run a rule
;   5733 = Successfully run rules
;   2469 = Rule runs which dismissed
;      0 = Rules which overrode other rules
;    428 = Rules which superseded other rules
;     30 = Usurpations
;    229 = Contradictions dequeued for processing
;    159 = @NODE contradictions dequeued for processing
;      0 = @CONSTRAINT contradictions dequeued for processing
;     70 = @RESOLUTION contradictions dequeued for processing
;    229 = Contradictions actually processed
;    229 = Nogood culprits automatically chosen
;    229 = Nogood sets installed
;    201 = Number of calls to ==
;  31887 = Awakenings
;   4707 = @ADDED awakenings
;      0 = @FORGET awakenings
;  25468 = @NOGOOD awakenings
;   2755 = Values forgotten
;  17166 = Number of nogood sets checked
;  26378 = Number of nogood set pairs checked
NIL
```

```
(disallow q0 q1 q2 q3 q4 q5)
DONE
(list q0 q1 q2 q3 q4 q5)
(<CELL-49 (Q0) SLAVE 2>
 <CELL-51 (Q1) SLAVE 5>
 <CELL-53 (Q2) SLAVE 1>
 <CELL-55 (Q3) SLAVE 4>
 <CELL-57 (Q4) SLAVE 0>
 <CELL-59 (Q5) SLAVE 3>)
```



```
(stats)
;    248 = Repositories generated
;    248 = Cells generated
;      0 = Initialized cells
;     81 = Constraints generated
;   8991 = Iterations of top-level-loop queue scan
;   9523 = Rules enqueued
;   7321 = Added rules enqueued
;     51 = Forget rules enqueued
;   2151 = Nogood rules enqueued
;   8427 = Attempts to run a rule
;   7466 = Successfully run rules
;   3123 = Rule runs which dismissed
;      0 = Rules which overrode other rules
;    660 = Rules which superseded other rules
;     30 = Usurpations
;    285 = Contradictions dequeued for processing
;    172 = @NODE contradictions dequeued for processing
;      0 = @CONSTRAINT contradictions dequeued for processing
;    113 = @RESOLUTION contradictions dequeued for processing
;    285 = Contradictions actually processed
;    285 = Nogood culprits automatically chosen
;    285 = Nogood sets installed
;    201 = Number of calls to ==
;  45184 = Awakenings
;   5870 = @ADDED awakenings
;      0 = @FORGET awakenings
;  36872 = @NOGOOD awakenings
;   3602 = Values forgotten
;  32911 = Number of nogood sets checked
;  51809 = Number of nogood set pairs checked
NIL
```

At this point all possible solutions have been generated. Disallowing this one causes a "hard-core contradiction", after which the final statistics are as follows.

```
(stats)
;     248 = Repositories generated
;     248 = Cells generated
;       0 = Initialized cells
;      81 = Constraints generated
;   21387 = Iterations of top-level-loop queue scan
;   29538 = Rules enqueued
;   18027 = Added rules enqueued
;      51 = Forget rules enqueued
;   11460 = Nogood rules enqueued
;   20283 = Attempts to run a rule
;   19256 = Successfully run rules
;    8606 = Rule runs which dismissed
;       0 = Rules which overrode other rules
;     763 = Rules which superseded other rules
;      30 = Usurpations
;     825 = Contradictions dequeued for processing
;     197 = @NODE contradictions dequeued for processing
;       0 = @CONSTRAINT contradictions dequeued for processing
;     628 = @RESOLUTION contradictions dequeued for processing
;     825 = Contradictions actually processed
;     824 = Nogood culprits automatically chosen
;     824 = Nogood sets installed
;     201 = Number of calls to ==
;  218830 = Awakenings
;   16150 = @ADDED awakenings
;       0 = @FORGET awakenings
;  190396 = @NOGOOD awakenings
;    9887 = Values forgotten
;  162765 = Number of nogood sets checked
;  250279 = Number of nogood set pairs checked
NIL
```

It is useful to compare this to the results of the LISP program of Table 5-18 (page 162), which uses chronological backtracking.

```
(queens 6)
Solution: (1,3,5,0,2,4) after 140 contradictions and 25 backtracks.
Solution: (2,5,1,4,0,3) after 334 contradictions and 64 backtracks.
Solution: (3,0,4,1,5,2) after 408 contradictions and 79 backtracks.
Solution: (4,2,0,5,3,1) after 602 contradictions and 118 backtracks.
Total of 742 contradictions and 149 backtracks.
DONE
```

The "backtracks" of this program correspond to resolution steps, and so we may compare numbers directly:

	LISP Contradictions	Constraint Contradictions	LISP Backtracks	Constraint Resolutions
After first Solution	140	103	25	21
After second Solution	334	128	64	42
After third Solution	408	159	79	70
After fourth Solution	602	172	118	113
After exhaustion	742	197	149	628

It easy to see that the non-chronological (constraint) version examines many fewer positions before arriving at a new solution. The number of resolution steps is about the same, unless there are no more solutions, in which case it must do about the same amount of work as the chronological version to prove that there is no solution (this is not surprising).

6.5. The New Improved Summary

This chapter has presented a complete re-implementation of the constraint language developed in Part One. A few new features (such as `disallow`) have been added to the language, but the primary emphasis has been placed on speed. The new system records multiple reasons for believing a value. It uses a task queue control structure for more flexibility in allocating computational resources. It pre-compiles tables of rules for primitive operators for fast run-time access and discrimination of rule to be executed. It hashes `constant` cells in order to share those with the same value among multiple uses. It uses a uniform algebraic notation in printing parts of the network as expressions.

Oh, once the opposition
Was completally opposed
To all the suppositions
That was **gen***'rally supposed:*
An' **now** *the superstitions*
That were tho't to be imposed
Are seen by composition
To be slightly decomposed!
—Walt Kelly (1952)
I Go Pogo

Chapter Seven

Correctness

THE CONSTRAINT SYSTEM described in the previous chapter is a large program, sufficiently complicated that there may well remain in it subtle errors. Nevertheless, the design is intended to make it simple(r) to argue that it is correct. All of the program state is made explicit in the form of LISP data structures, concerning which certain strong invariants may be stated.

I have not in any sense demonstrated that the system is correct. The system is an exercise in engineering; it is too large to be a reasonably manageable exercise in mathematics. However, in this chapter is outlined a series of statements which, if rigorously proved invariant over the processing of any queued task, would go a long way toward proving total correctness of the system. Some of these I *have* proved informally for certain classes of tasks, and consideration of these statements certainly aided in "eyeballing" the code for errors.

Parenthetical remarks provide definitions of terms and supplementary elucidation.

7.1. The Structure of Nodes

For every cell:

- its *id* is a LISP symbol (read-only). (The remark "(read-only)" about a component means that the component may never be altered once initialized.)
- its *repository* is (a pointer to)[1] a repository.

1. As in LISP *everything* is represented in terms of pointers, following this one reminder I shall not mention the presence of pointers. Nevertheless, the sharing of objects is very important!

- its *owner* is either `()` or a constraint (read-only).
- its *name* is a LISP symbol or an integer (read-only).
- its *state* is one of the six constants `@king`, `@puppet`, `@friend`, `@slave`, `@rebel`, `@dupe`. (When we say "a cell is a king" we mean that its *state* is `@king`, and similarly for the others.)
- its *contents* is either `()`, an integer, or a cell.
- its *rule* is either `()` or a rule.
- its *equivs* is a list of distinct cells. (By "a list of distinct things" we mean "a list of things in which no thing appears more than once". The list thus represents a set, and so the list may contain the elements in any order unless otherwise stated.)
- its *link* is a cell or `()`.
- its *mark* is `()`. (Thought apparently constant, this is *not* read-only! It may be used temporarily, but must always be reset to `()` before scheduling a new task.)

For every repository:

- its *cells* is a list of distinct cells.
- its *supplier* is a cell.
- its *id* is a LISP symbol (read-only).
- its *nogoods* is a list of buckets; each bucket is a pair whose *car* is an integer and whose *cdr* is a list of nogood sets. No two buckets in the *nogoods* of a single repository may contain the same integer. (We shall say that "a nogood set is in the *nogoods* of a repository" if the nogood set is a member of some list which is the *cdr* of some bucket of the *nogoods* of the repository.)
- its *contra* is an integer.

Relationships among cells and repositories:

- For every repository *r*, for every cell *c* in the *cells* of *r*, the *repository* of *c* is *r*.
- For every repository *r*, its *supplier* is a member of its *cells* list.
- The LISP value of the *id* of a repository is that repository.
- The LISP value of the *id* of a cell is that cell.
- If the *name* of a cell is a symbol, the LISP value of that symbol is the cell.
- The *link* of a cell is a member of the *cells* list of the *repository* of that cell.
- If a cell has a constraint for its *owner*, then its *name* is an integer.
- The members of the *equivs* list of a cell are all members of the *cells* list of the *repository* of that cell.
- If cell *a* is a member of the *equivs* list of cell *b*, then *b* is a member of the *equivs* of *a*.
- No cell is in its own *equivs* list.
- The cell which is the *supplier* of a repository must have `()` for its *link*.
- Any cell which is not the *supplier* of its *repository* must have a cell for its *link*.

- If cell *a* is the *link* of cell *b*, then *a* is in the *equivs* list of *b* (and therefore *b* is also in the *equivs* list of *a*).
- Consider the relationship between two cells *a* and *b* "cell *b* is the *link* of cell *a*"; the transitive closure of this relationship is an irreflexive partial order, that is, there are no cycles. (This plus the fact that precisely one cell of a node has a null link implies that the link structure forms a directed tree with all paths eventually converging at the supplier.)

Relationships among the states of cells and other things:

- No cell which is not a supplier is a king or puppet.
- The cell which is the *supplier* for a repository is a king or puppet. (By "the supplier of a cell" we mean the *supplier* of the *repository* of the cell. By "the king of a cell" we mean the *supplier* of the *repository* of the cell, which is known to be a king.)
- The supplier of a rebel, friend, rebel, or dupe is a king. (Conversely, if the *supplier* of a repository is a puppet, then all other members of that repository's *cells* list are slaves.)
- A slave or puppet has ( ) for its *contents* and its *rule*.
- A dupe has a cell for its *contents*, and that cell is a rebel and a member of the *cells* list of the dupe's *repository*. A dupe has ( ) for its *rule*.
- A king, friend, or rebel cell has an integer for its *contents*, and a rule for its *rule*.
- The *contents* of a friend is the same as the *contents* of its king.
- The *contents* of a rebel is different from the *contents* of its king.
- A king, friend, or rebel cell either has a constraint for its *owner*, or has one of the three special rules `*constant-rule*`, `*default-rule*`, or `*parameter-rule*` for its *rule*.
- The *contra* of a repository is equal to the number of rebels in its *cells* list.

We say that "a cell has a value" if the cell is a king, friend, rebel, or dupe, or if it is a slave and the *supplier* of the cell's *repository* is a king. The value of a king, friend, or rebel is its *contents*; the value of a dupe is the *contents* of its *contents* (a rebel); the value of a slave which has a value is the *contents* of its king.

7.2. Constraint-types and Constraints

For every constraint-type:

- its *name* is a LISP symbol (read-only).
- its *vars* is an array of distinct LISP symbols (read-only). (When a constraint-type is being discussed, a reference to *N* refers to the length of this array. An integer used to index this array, or any parallel array, is called a pin-number. The corresponding element of the *vars* array is called the pin-name for that pin-number.)

- its *added-rules* is an array of length N of lists of distinct rules (read-only).
- its *forget-rules* is an array of length N of lists of distinct rules (read-only).
- its *nogood-rules* is an array of length N of lists of distinct rules (read-only). (The three arrays *added-rules*, *forget-rules*, and *nogood-rules* of a constraint type are called the "rule arrays" of the constraint-type. The set of all rules appearing in any element of any rule array of a constraint-type is called the "rule set" for that constraint-type.)
- its *symbol* is a LISP symbol (read-only).

Every constraint-type is entirely read-only.

For every constraint:

- its *name* is a LISP symbol (read-only).
- its *ctype* is a constraint-type (read-only). (When a constraint is being discussed, a reference to N refers to the length of the *vars* array of the *ctype* of the constraint. Also, by an "instance" of a constraint-type we mean any constraint whose *ctype* is that constraint-type.)
- its *values* is an array of length N of distinct cells (read-only). (The set of cells which are elements of the *values* array of a constraint are called the "pins" of that constraint.)
- its *info* may be anything (normally read-only).
- its *queued-rules* is an integer.

Miscellaneous relationships:

- The LISP value of the *name* of a constraint-type is that constraint-type.
- The LISP value of the *name* of a constraint is that constraint.
- The *symbol* of a constraint-type has on its property list a `ctypename` property whose value is the *name* of the constraint-type.
- If a cell has a constraint for its *owner*, then the *name* of the cell is an integer j not less than 0 and less than N, and entry j of the *values* array of the *owner* of the cell is that cell. (It follows that all the cells in the *values* array of a constraint have distinct *name* components ranging from 0 to $N - 1$.)

7.3. Rules

For every rule:

- its *triggers* is a list of distinct integers (read-only).
- its *outvar* is either ( ) or an integer (read-only).
- its *code* is a LISP symbol (read-only).
- its *ctype* is a constraint-type (read-only).

- its *bits* is an integer 0, 1, 2, or 3 (read-only). (We say that a rule is a `&nogood` rule if this integer is 1 or 3, and a `&nogoodbeg` rule if it is 2 or 3.)
- its *id-bit* is a positive integer which is a power of two (read-only).

Every rule is entirely read-only.

Relationships among rule components:

- The LISP value of the *code* of a rule is that rule.
- The LISP function definition for the *code* of a rule is a LISP function of one argument. The value of this function (when given an appropriate argument, to be described later) is either an integer or one of the special constants `@lose` and `@dismiss`, and it may not be an integer if the *outvar* of the rule is `( )`.
- If the *outvar* of a rule is an integer, then it is not equal to any member of the *triggers* list.
- If the *outvar* of a rule is `( )` then the *bits* of the rule is zero.

Relationships between rules and constraint-types:

- If a rule is a member of any element of any rule array of a constraint-type, then the *ctype* of that rule is that constraint-type.
- The integer elements of the *triggers* of a rule, as well as the *outvar* of the rule if it is not `( )`, are each not less than 0 and less than *N*, the length of the *vars* of the *ctype* of the rule. (Thus each of these integers is a pin-number.)
- A rule is a member of element *j* of the *added-rules* array of its *ctype* if and only if *j* is a member of the rule's *triggers*.
- A rule is a member of element *j* of the *forget-rules* array of its *ctype* if and only if its *outvar* is *j*.
- A rule is a member of element *j* of the *nogood-rules* array of its *ctype* if and only if its *outvar* is *j* and its *bits* is non-zero.
- The *id-bit* components of all the rules of a constraint-type's rule set are distinct.

On the running of rules:

- The *code* of a rule may be applied only to an instance of the *ctype* of the rule. (When such an application is performed we say that the rule is "run" on the instance constraint.)
- When a rule is run on a constraint, the trigger cells for the rule must all have values. (By the "trigger cells" of a rule being run, we mean the set of cells which are elements of the *values* of the array and whose names (which are the corresponding indices into that array) are members of the *triggers* list of the rule.)
- The result computed by running a rule on a constraint may depend only on the values of the trigger cells and the *info* of the constraint; and also on the *nogoods* of the *repository* of the output cell (if any) provided that the *bits* of the rule is non-zero. (If the rule's *outvar* is not `( )` and the result of running the rule on a constraint is an integer, we say that "the rule produced

the integer for the output cell", where by "output cell" we mean that element of the *values* array of the constraint whose index in the array equals the *outvar* of the rule.)

- If a cell is a king, friend, or rebel, and its *owner* is a constraint, then its *rule* must be a rule in the rule set of the *ctype* of the *owner* of the cell, and that rule's *outvar* must be equal to the *name* (an integer) of the cell. Moreover, the triggers cells of the rule must all have values which would cause the rule, if run, to return the integer which is the *contents* of the cell. (Very important! This requirement implies that all values are well-founded upon premises.)

On the consistency of rules:

- Suppose that two rules in the rule set of some constraint-type have the same integer j for their respective *outvar* components, and that the set of elements of the *triggers* of the first is a subset (not necessarily a propert subset) of the set of elements of the *triggers* of the second. Consider some instance of that constraint-type, and suppose that every pin which is a trigger cell of the second rule (and therefore also of the first) has a value. Consider the values which would be produced by running either of the two rules in this same situation. If both values would be integers, then they must be the same integer unless at least one of the rules is a `&nogoodbeg` rule. (Indeed, in this situation one would *expect* the first rule, which gets less information, to be a `&nogoodbeg` rule.)
- Consider a constraint, and an ordered sequence with distinct elements $(r_1, r_2, \ldots, r_n)$ of at least two rules from the rule set of the *ctype* of the constraint. Suppose each of the rules has a non-null *outvar*, and that for each $1 \leq j < n$ the output pin of rule r_j is a trigger cell for rule r_{j+1}, and the output pin of rule r_n is a trigger cell for rule r_1. Suppose that all the trigger cells of all the rules have values, except those which are output pins of all rules except r_n. Now suppose that the *code* of rule r_1 is run, producing a value which is then assigned to the output pin of r_1; then rule r_2 is run, and so on. Then the value produced by r_n must agree in value with the value its output pin already had unless some r_j is a `&nogoodbeg` rule. (Example: one adder rule takes `a` and `b` and computes `c`; then if the rule that takes `b` and `c` to produce `a` were run, it ought to produce the same value for `a`. Note, however, that such a rule would not ordinarily actually be run (rules are not awakened by values computed by other rules of the same constraint), precisely because this consistency is taken for granted.)

7.4. Tasks and Queues

Properties of tasks and queues:

- A queue contains, among other things, a bag of tasks. Queues are of two kinds: rule queues and contradiction queues. (I am ignoring the `*rebel-queue*` here, for that is a minor variation.) Rule queues may contain only rule tasks, and contradiction rules only contradiction tasks.

- A rule task is a pair of a rule and a constraint. The constraint must be an instance of the *ctype* of the rule.
- Contradiction tasks are lists, and are of three kinds, distinguished by a special constant in the *car* of the list. A `@node` contradiction task has two cells in its *cadr* and *caddr*. A `@constraint` contradiction has a constraint for its *cadr*, and its *cddr* is an a-list associating cells with integers. A `@resolution` contradiction has a *cdr* which is an a-list associating cells with integers.
- At "task scheduling time", *any* task may be removed from any queue and executed. When that task has completed, it is task scheduling time again. (It is at this time that all of the invariants presented in this chapter must hold.)

Rule tasks:

- Consider any rule and any instance of that rule's *ctype*. Suppose that all the triggers cells for that rule have values. Then one of three situations must hold: (1) At least one of the trigger cells is a king, friend, or rebel (implying that its value was computed by some other rule for the same constraint). (2) A task pairing the rule with the constraint is in some rule queue. (3) Let x be the result of running the *code* of the rule on the constraint in that situation. If x is `@lose` then there must be a `@constraint` task on the contradiction queue mentioning the all trigger cells of the rule. If x is an integer then the output pin of the rule must be a king, friend, or rebel, and have that integer as its *contents*.
- If the value q for a constraint's *queued-rules* satisfies $(q \pmod{2^{j+1}}) \geq 2^j$ for some integer j (i.e. the 2^j-bit is set in the bit-vector represented by q), then there is in some rule queue a rule task pairing that constraint with the unique rule in the rule set of the constraint's *ctype* which has an *id-bit* component equal to 2^j.

Contradiction tasks:

- For every rebel cell there must be in some contradiction queue a `@node` contradiction task mentioning the rebel cell and its king.

7.5. Nogood Sets

- A nogood set is a list whose *car* is the LISP symbol `nogood` and whose *cdr* is an a-list associating distinct repositories with integers.
- A nogood set is in the *nogoods* of a repository if and only if the repository is mentioned in the a-list of the nogood set. More specifically, the nogood set will be in that (unique) bucket of the repository's *nogoods* whose *car* is the integer which the nogood set's a-list associates with the repository.
- If a nogood set exists, then it must be the case that if all values were removed from the network, excepting constant cells, and then default cells containing the integers specified in the nogood

set were added to the respective repositories, then by running appropriate rules a contradiction could be derived whose premises would be precisely the added default cells. (This invariant is stated very loosely, of course. The point is that a contradiction could be logically derived in a well-founded manner solely from the values in the nogood set.)

7.6. User Interface

Invariants of the previous sections are concerned with the internal workings of the system. There also need to be statements relating these internal workings to what the user types, to ensure that the work done by the system actually reflects the meaning understood by the user.

- Whenever a task is to be scheduled, it is permissible instead to process a user request (which may of course alter the network).
- If there are no tasks on any of the queues (in which case we say that the system is *quiescent*), then the network is free of contradiction, and the stated relationships hold among all quantities in the network. The network structure may then be understood to represent the sum total of all preceding user input, and the contents of cells to represent validly deduced values.
- A sequence of user inputs (other than information queries such as `why`) can be understood in terms of an equivalent sequence of inputs consisting only of `create`, `variable`, `==`, and `disallow` statements, in that order.

The remainder of this section would consist essentially of the language definition from §6.1, which is therefore not repeated here.

7.7. Summary

This chapter does not by any means indicate all the invariants which might possibly be stated. It does present a large number of invariants, some of them rather complex, which ought to be shown to hold.

One would also like to be able to exhibit a true multiprocessing implementation of a constraint language. If it were based on this implementation, one approach would be to identify the precise conditions under which two tasks could interfere with each other, and then arrange interlocks so that interfering tasks cannot run in parallel. Of course, the current implementation enforces such an interlock, a rather stringent one requiring that *no* two tasks run in parallel! But, for example, it is entirely plausible that many rule tasks could run in parallel, provided that they operated on constraints sufficiently separated within the network, or that appropriate interlocks were placed in `process-setc` with regard to the setting of cell contents and recording of nogood sets.

(Techniques for proving properties of parallel programs of this type are described in [Owicki 1975] and [Gries 1977].)

[This page intentionally left blank.]

Part Three

Abstraction

Well, then Fido got up off the floor
And he rolled over
And he looked me straight in the eye.
And you know what he said?

"Once upon a time
Somebody say to me."
(This is the dog talkin' now.)
"What is your
conceptual
continuity?

"Well, I told him right then," Fido said.
"It should be easy to see
The crux of the biscuit
Is the apostrophe."

Well, you know, the man who was talkin' to the dog
looked at the dog and he said
(Sort of starin' in disbelief).
"You can't say that!"

He said,
"It doesn't!
And you can't!
I won't!
And it don't!
It hasn't!
It isn't!
It even ain't!
And it shouldn't!
It couldn't!"
He told me, "No no no!"
I told him, "Yes yes yes!"
I said, "I do it all the time!"
(Ain't this boogie a mess?)

—Frank Zappa (1974)
"Stinkfoot"
Apostrophe

"The Old Oaken Bucket"
Already is written:
There's naught left to me
For an amphibrach fitten.

—Guy Lewis Steele, Sr.

A stitch in time is worth two in the bush
If you count them before you come to them.
Excepting February, which has twenty-eight.

—The Reverend Doctor Cuddles

Chapter Eight

Hierarchy

PREVIOUSLY PRESENTED VERSIONS of the constraint language have been "flat". In this chapter two forms of hierarchy are introduced. One stems from a macro-definition mechanism, which allows the user to define non-primitive constraint devices in terms of a network of other constraints. One may think of this as a trivial kind of subroutine mechanism, one which does not permit recursion. This mechanism introduces a calling hierarchy or an abstraction hierarchy, with complex things defined in terms of simpler things to many levels. The other form of hierarchy stems from permitting the user to write expressions in the nested algebraic syntax described in §6.2.5; this is a syntactic hierarchy, with complex expressions built from simpler ones. Both forms of hierarchy allow networks to be expressed much more concisely.

8.1. New Features for the Constraint Language

Here the changes to the constraint language are described. Besides the expression syntax and the macro mechanism, a special parsing and evaluating mechanism will be introduced which will relieve the restriction of the syntax to LISP-evaluable forms. The parser takes an S-expression representing a request for the constraint system, and reduces it to a set of simple statements. The evaluator acts on these statements, usually just by calling the LISP evaluator (since much of the system is derived from the version in Chapter Six), but not always. In addition, a simple iteration construct is provided.

8.1.1. The User Can Describe Networks Using the Expression Syntax

All nested expressions are considered to be abbreviations for a collection of `create` and `==` statement (indeed, in the implementation described later in this chapter, expressions are processed by constructing that equivalent collection and then processing the collection). The syntax will therefore be explained as such abbreviations.

Suppose that `sym` is the *symbol* for some constraint-type named `type`, and that the names of the pins for that constraint-type are `x`, `y`, `z`, . . .; then the expression

```
((sym name) a b c ...)
```

is a statement equivalent to these statements:

```
(create name type)
(== a (the x name))
(== b (the y name))
(== c (the z name))
   ⋮
```

Also, if any one of the argument forms is the symbol "%" then the expression is not a statement, but rather denotes the corresponding pin of the constraint instance. Thus, for example,

```
(== ((sym name) a % c ...) foo)
```

is equivalent to

```
(create name type)
(== a (the x name))
(== c (the z name))
   ⋮
(== (the y name) foo)
```

because the % was in the position corresponding to `(the y name)`.

There must be exactly as many argument forms as there are pins for the specified constraint-type (with two exceptions), and they are matched by order of appearance of argument forms in the mentioning expression and order of appearance of pin-names in the declaration of the constraint-type. If no argument form is %, then the expression is a statement and does not denote anything; if one if %, then the expression denotes the corresponding pin. No more than one argument form may be a %.

One exception to the rule is that one fewer argument form than the number of pins may be written, and no % written; in this case a % is taken to be an implicit argument form preceding all the others. The other exception is that an extra argument form may be written, which will become the *info* component of the constraint instance. In this way one can write `(assumption % 4)`, or simply `(assumption 4)`. The special routine `assume` is not needed in the implementation to construct an assumption; the necessary machinery falls out of this general notation.

As an example, a temperature conversion network may be described by the single statement

```
((+ add) fahrenheit ((* othermult) ((* mult) 9 centigrade) 5 %) 32)
```

which is entirely equivalent to the old definition

```
(create add adder)
(create mult multiplier)
(create othermult multiplier)
(== fahrenheit (the c add))
(== (the b add) (constant 32.))
(== (the a add) (the a othermult))
(== (the c othermult) (the c mult))
(== (the b othermult) (constant 5))
(== centigrade (the b mult))
(== (the a mult) (constant 9))
```

(Note that the `variable` declarations have been omitted. The parser arranges to perform an implicit `variable` declaration during the parsing process for any variable mentioned in an expression, *provided* that the variable has not already been so declared. To get the effect of redeclaring a variable, the user can just `destroy` it and then mention it again.)

If instead of a list `(sym name)` in the "operator position" of an expression, the user writes simply `name`, then the parser generates a name of the form `type-`*nnn*.

If in place of an argument form the user writes "?", then no `==` statement is generated for that argument position; it means "the corresponding pin is not connected to anything here".

8.1.2. The User Can Define Non-primitive Constraints

A form is provided for declaring new constraints in terms of old ones:

`(defcon` *name pin-names . body*`)`

says that *name* is the name of a new constraint-type whose *vars* is the set of names *pin-names*. Whenever an instance of *name* is to be created, a copy of the network described by the statements in *body* is constructed. Thereafter *name* may be used as any other constraint-type name in `create` statements.

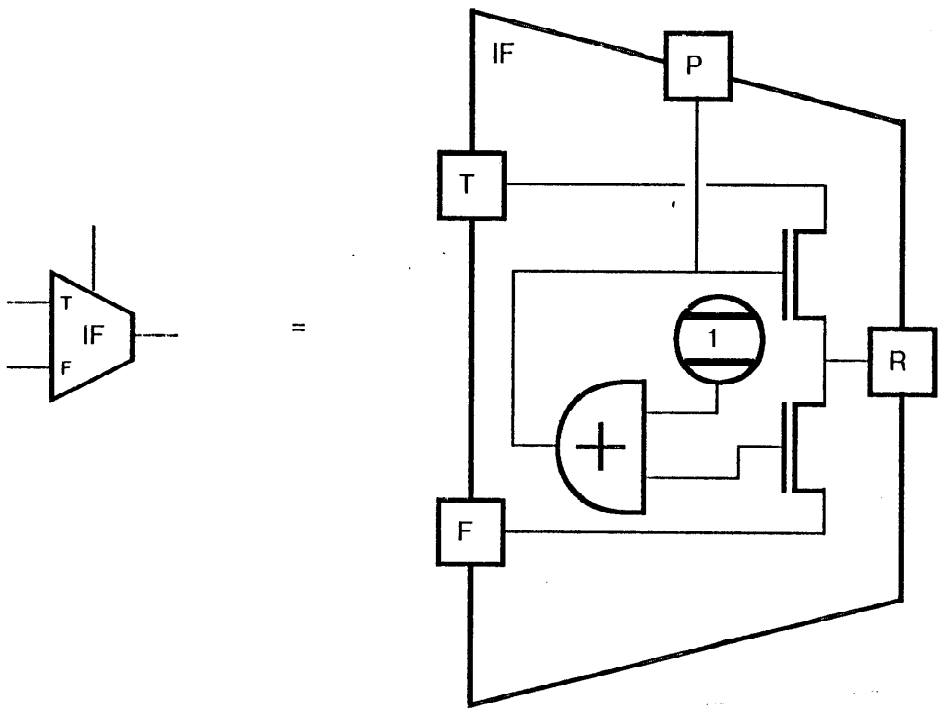


FIGURE 8-1. User Definition of the `if` Device.

As an example, we can define a "temperature converter device" which has two pins called `f` and `c` which enforces the Fahrenheit-to-Centigrade relationship between the two pins:

```
(defcon temp-converter (f c)
       ((+ add) f ((* othermult) ((* mult) 9 c) 5 %) 32))
```

If later we were to say `((temp-converter tc) fahrenheit centigrade)` then the usual relationship between `fahrenheit` and `centigrade` would hold, mediated by an instance of `temp-converter` called `tc`.

One can of course refer to the pins of such a constraint instance by saying, for example, `(the c tc)` to refer to the pin `c` of `tc`. One can also refer to the devices used in the instantiated network. The expression `(the add tc)` refers to the adder of the network for the instance `tc` of `temp-converter`; it follows that `(the b (the add tc))` is a pin which is connected to the constant 5. If `tc` had been part of another device `zed` then `(the b (the add (the tc zed)))` would name a pin. In this way we can use *pathnames* to refer to parts of parts of . . . parts of a complex constraint device.

As another example of a useful device, we can define an `if` device, a non-directional version if the standard **if-then-else-fi** programming-language construct (which connects a "result" pin to

one of two "source" pins; or, from another point of view, connects a source pin to one of two result pins!):

```
(defcon if (result test then else)
        (gate test then result)
        (gate (+ 1 test %) else result))
```

(See Figure 8-1.) This definition uses an adder to perform logical negation. With this, one can then write things like

```
(+ f (* (* 9 (if kelvin-flag (+ c % 273) c)) 5 %) 32)
```

to enable `c` to be either a temperature Kelvin or a temperature centigrade according the the flag `kelvin-flag`.

If any variables other than pins are mentioned in the *body* of a `defcon` definition, they are taken to be *local* to the definition; the variable is instantiated afresh for each instance of the containing definition. If it is desired that every instance be hooked up to some single instance of a global variable, then the construct (`global` *var*) may be used to refer to it.

8.1.3. Pathnames May be Written is Abbreviated Form

A pathname such as `(the b (the add (the tc zed)))` may be contracted to simply `(the b add tc zed)`. For even greater conciseness, it may be written simply as `zed.tc.add.b`. Here the path is written in the reverse order, with the name of the original object first and successive selectors following, separated by periods. If the name contains a leading period, then the name of the initial object is global; thus `.foo` is the same as `(global foo)`. This is of course similar to the component selection notation of many programming languages, and also to the file pathnames of Multics and UNIX, which use characters other than the period.

This facility is made possible by the introduction of the parser, which checks symbols in the input for periods in their names, and expands them into appropriate `the` and `global` constructs.

8.1.4. The `vector` Construct Provides Limited Iteration

The special form

(`(vector` *name*`)` *size interface common . body*`)`

defines and instantiates a special kind of constraint-type, a vector consisting of *size* copies of the network defined by *body*, placed side by side. The *size* must be an integer—this limited facility does

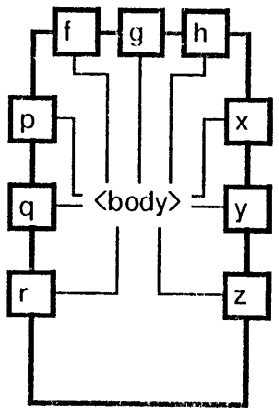


FIGURE 8-2. Pictorial Representation of the Body Prototype for a Vector.

not allow for variable-length vectors. The instances of *body* have names which are the integers from zero (inclusive) to *size* (exclusive).[1]

The *interface* describes how adjacent networks in the series are connected. It is a list of descriptors, and each descriptor is a list of four things:

(*leftedge left right rightedge*)

The *left* and *right* must be names; they are pin-names for the *body*. If two instances *x* and *y* of the *body* are adjacent, with *x* to the left of *y*, then the *right* of *x* is connected to the *left* of *y*. The instance of *body* at the left end of the row has its *left* connected to *leftedge*, which may be any expression denoting a cell, or ?; similarly for the *right* of the instance at the right end of the row and *rightedge*.

The list *common* is a list of names global to the `vector` construct which are to be made available to every instance of *body*. (This set is deducible from context, but to simplify the present implementation the user is required to declare these.)

As usual, if one writes `(vector ...)` instead of `((vector` *name*`) ...)` then a name is automatically generated.

Figure 8-2 shows a diagram representing the body of a vector defined as

```
((vector foo) 7 ((a p x i) (b q y j) (c r z k)) (f g h) <body>)
```

1. For technical reasons the names are actually LISP symbols whose print names are digit strings which look like the way the integer would print. Thus the first instance in a vector has the name `|0|` or `/0`, not `0`, to use Lisp Machine LISP syntax. As we shall see, when pathnames with periods are used this distinction is not apparent.

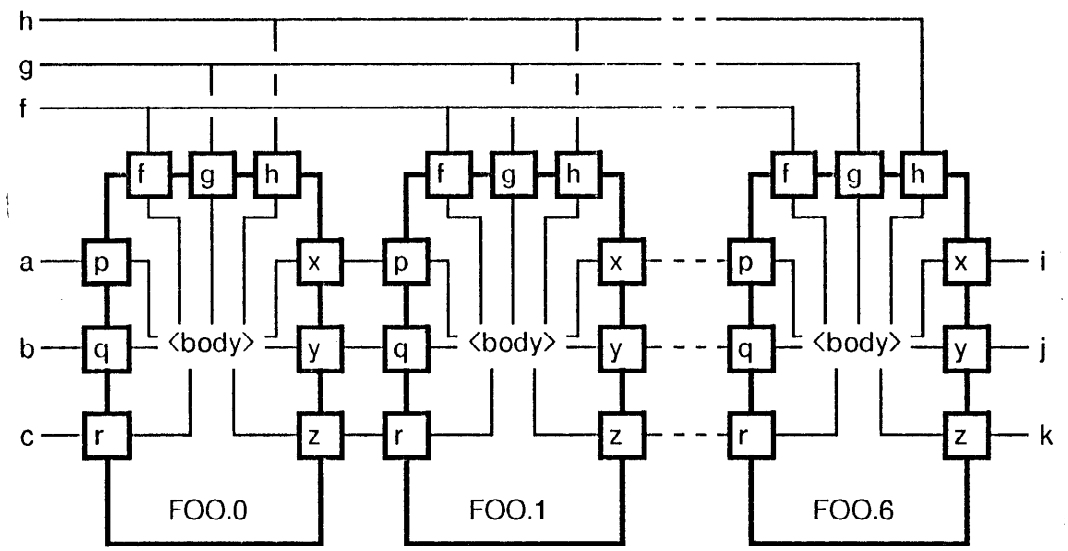


FIGURE 8-3. An Entire Vector, and Its Connections.

Figure 8-3 shows three of the seven instances of the body belonging to the vector, and their connections. The left-hand pins `p`, `q`, and `r` of each one are connected to the right-hand pins `x`, `y`, and `z` of the instance to the left. The leftmost instance has its left-hand pins connected to `a`, `b`, and `c`, while the rightmost has its right-hand pins connected to `i`, `j`, and `k`. All of them have the common pins `f`, `g`, and `h` connected to the external variables of the same name.

As a simple concrete example:

```
((vector foo) 4 ((input a b output)) () (+ b a a))
```

makes a length-four chain of adders like the one in Figure 3-2 (page 84). This statement is entirely equivalent to these declarations:

```
(DEFCON VECTOR-BODY-374 (A B) (+ B A A))
(DEFCON VECTOR-TYPE-373 (A B)
  ((VECTOR-BODY-374 |0|) A ?)
  ((VECTOR-BODY-374 |1|) (THE B |0|) ?)
  ((VECTOR-BODY-374 |2|) (THE B |1|) ?)
  ((VECTOR-BODY-374 |3|) (THE B |2|) B))
((VECTOR-TYPE-373 FOO) INPUT OUTPUT)
```

The *body* of the vector is made into a macro-constraint-type. Another macro-constraint-type is declared for the entire vector, which makes four instances of the body and makes the internal connections between adjacent instances. Finally, this latter macro-constraint-type is instantiated, the instance is named `foo`, and the edge connections to `input` and `output` are made. Note the

```
(deftype constraint-type
  (ctype-name ctype-vars ctype-added-rules ctype-forget-rules
   ctype-nogood-rules ctype-symbol (ctype-initfn ()))
  (format stream "<Constraint-type ~S>" (ctype-name constraint-type)))

(deftype constraint
  (con-name con-owner con-ctype con-values con-info (con-queued-rules 0))
  (format stream "<~{~A~^.~}:~S>"
          (con-pathname constraint) (ctype-name (con-ctype constraint))))

(deftype macro-constraint-type
  (mctype-name mctype-pins mctype-allvars mctype-creations mctype-connector)
  (format stream "<Macro-constraint-type ~S>" (mctype-name macro-constraint-type)))

(deftype macro-constraint
         (mcon-name mcon-owner mcon-mctype mcon-values mcon-devices)
  (format stream "<~{~A~^.~}:~S>"
          (con-pathname macro-constraint)
          (mctype-name (mcon-mctype macro-constraint))))
```

Compare this with Table 6-3 (page 204).

TABLE 8-1. Macro-constraint-types and Macro-constraints.

use of ? to indicate no connection, and the "numerical" names for the components of the vector. For example, there is an equating between `foo.0.a` and `foo.a`, which is in turn connected to `input`. Similarly, `foo.0.b` and `foo.1.a` are connected, as are `foo.1.b` and `foo.2.a`.

8.2. Implementation of Parsing and Macros

The code given here shows only the changes from the full system described in Chapter Six. First the new data types are described, then changes to previously existing mechanisms, and finally the new top-level loop and parser.

8.2.1. Macro-constraints Are Instances of Macro-constraint-types

User-defined macro-constraints are represented in a way very similar to ordinary constraints. Table 8-1 gives the data structure definitions for constraint-types and macro-constraint-types, for constraints and macro-constraints. The differences arise from the fact that a constraint has pins and rules, but a macro-constraint has a defining network. The interface information is similar, however.

The list (actually an array) of pins *vars* in a constraint-type becomes two in a macro-constraint-type: *pins* and *allvars*, the first being a subset of the second. The *allvars* is the set of all variables in

the defining network, while *pins* is the set of terminals, variables to which the "outside world" connects. Primitive constraints have no internal variables represented by cells, and so do not require an *allvars* set.

A macro-constraint-type does not have a *symbol* because in this simple implementation there is no means for printing a macro-constraint in algebraic form. It does not have tables of rules, for there are no rules. It does have, however, two components called *creations* and *connector*. The *creations* is a list of 3-lists; each 3-list describes one `create` operation to be performed when instantiating the network for an instance of the macro-constraint-type. The first element of such a 3-list is the name of the device; the second is the type (either a constraint-type or a macro-constraint-type) of the device; and the third is a datum to be installed in the *info* component of the device (if it is a primitive constraint). The *connector* is a function of one argument which, when applied to a macro-constraint instance, will make all the equatings necessary to wire up the network.

A constraint as well as a cell may now have an *owner*, which of course must be a macro-constraint. The owner of a cell may now be a constraint or a macro-constraint.

```
(deftype cell (cell-id cell-repository cell-owner cell-name
               (cell-contents ()) (cell-state @lose) (cell-rule ())
               (cell-equivs '()) (cell-link ()) (cell-mark ()))
  (progn (format stream "<~S (~{~A~^.~})" (cell-id cell) (cell-pathname cell))
         (select (cell-state cell)
           ((@puppet) (format stream " PUPPET>"))
           ((@slave) (format stream " SLAVE~@[ ~S~]>"
                             (select (cell-state (node-supplier cell))
                               ((@king) (node-value cell))
                               ((@puppet) ())
                               (otherwise
                                   (list 'bad-supplier
                                         (cell-state (node-supplier cell)))))))
           ((@king) (format stream "~@[~* [OPPOSED]~] KING ~S>"
                            (plusp (node-contra cell))
                            (cell-value cell)))
           ((@friend) (format stream "~@[~* [OPPOSED]~] FRIEND ~S>"
                              (plusp (node-contra cell))
                              (cell-value cell)))
           ((@rebel) (format stream " REBEL ~S AGAINST ~S>"
                             (cell-value cell)
                             (if (eq (cell-state (node-supplier cell)) @king)
                                 (node-value cell)
                                 (list 'bad-supplier
                                       (cell-state (node-supplier cell))))))
           ((@dupe) (format stream " DUPE ~S AGAINST ~S>"
                            (cell-value cell)
                            (if (eq (cell-state (node-supplier cell)) @king)
                                (node-value cell)
                                (list 'bad-supplier
                                      (cell-state (node-supplier cell))))))
           (otherwise (format stream " BAD STATE ~S>" (cell-state cell))))))
```

Compare this with Table 6-4 (page 206).

TABLE 8-2. New Printing Format for Cells.

Note the user of the function `con-pathname` in the printing code for constraints and macro-constraint. This causes a constraint to print like this:

```
<TC.ADD:ADDER>
```

which is the `add` device (an `adder`) of macro-constraint `tc`. Similarly, the printing format for cells is changed (Table 8-2) to something like:

```
<CELL-78 (TC.ADD.B) PUPPET>
```

for the `b` pin of that same adder.

The construction of pathnames is shown in Table 8-3. All that is necessary is to start from a given object and trace up the hierarchy of owners. The resulting pathname is a list of names, with the (global) name of the ultimate owner first, followed by successive selector names.

```
(defun cell-pathname (cell)
  (require-cell cell)
  (cond ((null (cell-owner cell)) (list (cell-name cell)))
        ((constraint-p (cell-owner cell))
         (nconc (con-pathname (cell-owner cell))
                (list (aref (ctype-vars (con-ctype (cell-owner cell)))
                            (cell-name cell)))))
        ((macro-constraint-p (cell-owner cell))
         (nconc (con-pathname (cell-owner cell))
                (list (aref (mctype-allvars (mcon-mctype (cell-owner cell)))
                            (cell-name cell)))))
        (t (lose "Bad cell owner ~S for ~S." (cell-owner cell) (cell-id cell)))))

(defun con-pathname (con)
  (cond ((constraint-p con)
         (con-pathname-1 (con-owner con) (con-name con)))
        ((macro-constraint-p con)
         (con-pathname-1 (mcon-owner con) (mcon-name con)))
        (t (lose "Not a constraint: ~S" con))))

(defun con-pathname-1 (owner name)
  (cond ((null owner) (list name))
        (t (require-macro-constraint owner)
           (nconc (con-pathname owner)
                  (list (car (aref (mctype-creations (mcon-mctype owner))
                                   name)))))))
```

TABLE 8-3. Construction of Pathnames for Cells and Devices.

```
(defun cell-goodname (cell)
  (require-cell cell)
  (cond ((globalp cell) (cell-name cell))
        ((or (eq (cell-rule cell) *constant-rule*)
             (eq (cell-rule cell) *default-rule*)
             (eq (cell-rule cell) *parameter-rule*))
         (list (cell-name cell) (cell-contents cell)))
        (t (cons 'the (cell-pathname cell)))))
```

Compare this with Table 6-53 (page 280).

TABLE 8-4. The Best Name for a Pin Is Its Pathname.

The function `cell-goodname` can be simplified by letting it use `cell-pathname` (Table 8-4).

8.2.2. Owners Can Now Be Constraints *or* Macro-constraints

There are many places in the code which check for owners of cells, and which formerly required such owners to be constraints. Now an owner can be a constraint or a macro-constraint. Error checks must be modified to permit either kind of owner. Rather than reprinting many lines of code just to show these simple modification, I will just describe the changes here.

Functions which used to require constraints and now must permit either constraints or macro-constraints: `gen-cell`, Table 6-6 (page 211).

Functions which tested the owner of a cell and assumed a non-null owner to be a constraint, and which must now test that it is in fact a constraint (rather than a macro-constraint): `awaken`, Table 6-28 (page 245); `why`(in the case that the cell has no value), Table 6-52 (page 279); `why-how`(it now tests that the owner is a constraint purely for error-checking purposes), Table 6-53 (page 280); `fast-expunge-nogoods-mark`and `fast-expunge-nogoods-unmark`, Table 6-40 (page 264); `desired-premises-constraint`and `desired-premises-unmark`, Table 6-55 (page 283); `tree-form-trace`(in the `progn` form), Table 6-57 (page 285); `tree-form-deep`, Table 6-58 (page 286); and `tree-form-unmark`, Table 6-60 (page 288).

Also, there is one change in `tree-form-chase` (Table 6-59 (page 287)), near the middle of the code:

```
((cell-owner s)
 (cond ((and (eq s cell) (not top)) (cell-goodname s))
       ... )
 ...
```

becomes

```
((cell-owner s)
 (cond ((or (and (eq s cell) (not top))
            (macro-constraint-p (cell-owner s)))
        (cell-goodname s))
       ... )
 ...
```

the effect being that if the chase comes to a cell owned by a macro-constraint then it must be a puppet, an artificial supplier, and so the chase may as well end there.

```
(defmacro create (name type &optional (info ())) `(*create ',name ,type ,info))

(defun *create (name type info)
  (prog2 (*destroy name)
         (gen-constraint type name () info)
         (run?)))

(defmacro destroy (symbol) `(*destroy ',symbol))

(defun *destroy (symbol &optional (forced ()))
  (require-symbol symbol)
  (and (boundp symbol)
       (let ((val (symeval symbol)))
         (cond ((cell-p val)
                (cond ((and (globalp val) (eq (cell-name val) symbol))
                       (*detach val)
                       (makunbound (cell-id val))
                       (makunbound symbol))
                      (t (lose "Illegal re-declaration of ~S." symbol))))
               ((constraint-p val)
                (cond ((or forced (eq (con-name val) symbol))
                       (forarray (p (con-values val)) (*detach p))
                       (makunbound symbol))
                      (t (lose "Illegal re-declaration of ~S." symbol))))
               ((macro-constraint-p val)
                (cond ((or forced (eq (mcon-name val) symbol))
                       (forarray (p (mcon-values val)) (*detach p))
                       (forarray (d (mcon-devices val)) (*destroy d t))
                       (makunbound symbol))
                      (t (lose "Illegal re-declaration of ~S." symbol))))
               ((or (constraint-type-p val)
                    (macro-constraint-type-p val)
                    (repository-p val)
                    (rule-p val))
                (lose "Illegal re-declaration of ~S." symbol))
               (t (makunbound symbol)))))
  'done)
```

Compare this with Table 6-11 (page 217) and Table 6-41 (page 265).

TABLE 8-5. Creating and Destroying Things.

8.2.3. Macro-constraints Can Be Created and Destroyed

The `create` form, which specifies a name for a constraint and the constraint-type to instantiate, now permits a third argument form to be supplied (Table 8-5). This third form, if present, is used to fill in the *info* component of a constraint.

The routine `*destroy` has been modified to be able to destroy macro-constraints, and not to destroy macro-constraint-types. Also, the new argument `forced` is a flag which forces constraints

and macro-constraints to be destroyed despite the error-check that their names be symbols; this is needed in order to recursively destroy sub-devices of a macro-constraint.

```
(defun gen-constraint (ctype name &optional (owner ()) (info ()))
  (statistic gen-constraint)
  (and owner (require-macro-constraint owner))
  (if owner (require-integer name) (require-symbol name))
  (cond ((constraint-type-p ctype)
         (let ((c (make-constraint)))
           (or owner (set name c))
           (setf (con-name c) name)
           (setf (con-ctype c) ctype)
           (setf (con-owner c) owner)
           (setf (con-values c)
                 (array-of (fortimes (j (array-length (ctype-vars ctype)))
                             (gen-cell j c))))
           (setf (con-info c)
                 (if (ctype-initfn ctype)
                     (funcall (ctype-initfn ctype) c info)
                     info))
           (doarray (bucket (ctype-forget-rules ctype))
             (dolist (rule bucket)
               (and (null (rule-triggers rule))
                    (enqueue-rule rule c @forget))))
           c))
        ((macro-constraint-type-p ctype)
         (let ((c (make-macro-constraint)))
           (or owner (set name c))
           (setf (mcon-name c) name)
           (setf (mcon-mctype c) ctype)
           (setf (mcon-owner c) owner)
           (setf (mcon-values c)
                 (array-of (fortimes (j (array-length (mctype-allvars ctype)))
                             (gen-cell j c))))
           (setf (mcon-devices c)
                 (array-of (fortimes (j (array-length (mctype-creations ctype)))
                             (let ((x (aref (mctype-creations ctype) j)))
                               (gen-constraint (cadr x) j c (caddr x))))))
           (funcall (mctype-connector ctype) c)
           c))
        (t (lose "~S not a constraint-type or macro-constraint-type." ctype))))
```

Compare this with Table 6-11 (page 217).

TABLE 8-6. Generating a Constraint or Macro-constraint.

The function `gen-constraint`, which is used by `create`, is now capable of instantiating either a constraint-type or a macro-constraint-type. Moreover, an instance may have an owner now (which must be a macro-constraint), and if so the name will be an integer rather than a symbol.

When a macro-constraint-type is instantiated, a macro-constraint is created and its *name*, *mctype*, and *owner* slots are filled in. Then a cell is generated for every variable of the macro-constraint, as determined by the *allvars* array of the *mctype*, and these are stored in a corresponding array in the *values* component. Next all the devices needed by the defining network are created (this will involve recursive calls to `gen-constraint`); these are stored in the *devices* array.

```
(defmacro the (x y) `(*the ',x ,y))
(defmacro my (x) `(the ,x *me*))

(defun *the (name con)
  (or (cond ((constraint-p con) (lookup name con))
            ((macro-constraint-p con) (macro-lookup name con))
            (t (lose "Not a constraint: ~S." con)))
      (lose "~S has no part named ~S." con name)))

(defun lookup (name thing)
  (require-constraint thing)
  (let ((names (ctype-vars (con-ctype thing)))
        (cells (con-values thing)))
    (let ((n (array-length names)))
      (do ((j 0 (+ j 1)))
          ((= j n) ())
        (and (eq (aref names j) name) (return (aref cells j)))))))

(defun macro-lookup (name thing)
  (require-macro-constraint thing)
  (let ((names (mctype-allvars (mcon-mctype thing)))
        (cells (mcon-values thing)))
    (let ((n (array-length names)))
      (do ((j 0 (+ j 1)))
          ((= j n)
           (let ((creations (mctype-creations (mcon-mctype thing)))
                 (devices (mcon-devices thing)))
             (let ((m (array-length creations)))
               (do ((k 0 (+ k 1)))
                   ((= k m) ())
                 (and (eq (car (aref creations k)) name)
                      (return (aref devices k))))))))
        (and (eq (aref names j) name) (return (aref cells j)))))))
```

Compare this with Table 6-32 (page 253).

TABLE 8-7. Looking Up Parts of a Macro-Constraint.

Finally, the connector function is applied to the macro-constraint instance in order to wire up the network.

8.2.4. The `the` Construct Can Refer to Parts of a Macro-Device

The `the` construction must now be able to locate named parts of either constraints or macro-constraints. The function `*the` now merely divides into two cases; for macro-constraints it calls `macro-lookup`, which searches first the *names* array and then the `creations` array of the macro-constraint-type. When a matching name is found, the corresponding element of the *values* or *devices* array of the macro-constraint is returned.

The `my` macro is included as a convenience; "my x" is the same as "the x of `*me*`". (See Table 8-7.)

```
(defconst @quit (list '@quit))          ;quit from consys loop
(defconst @nothing (list '@nothing))    ;something the consys loop won't print

(defun consys ()
  (let ((rubout-handler ())                          ;variables controlling the
        (read-preserve-delimiters ()))               ; Lisp Machine READ function
    (format t "~&;Welcome to The Constraint System.")
    (consys-loop "]:")))

(defun consys-loop (prompt)
    (do () (())                                      ;do forever (until explicit return)
      (format t "~%~A" prompt)
      (setq - (si:read-for-top-level))
      (and (eq - @quit) (return))
      (setq // (multiple-value-list (evaluate-input -)))
      (setq *** **)
      (setq ** *)
      (setq * (car //))                              ;save first value
      (dolist (value //)
        (cond ((not (eq value @nothing))
               (terpri)
               (funcall (or prin1 #'prin1) value))))
      (setq +++ ++)
      (setq ++ +)
      (setq + -)))
```

TABLE 8-8. The Top-Level "Read-Eval-Print" Loop for the Constraint System.

8.2.5. A "Read-Eval-Print" Loop Processes User Requests

We have discussed all the changes to previously existing code; these had primarily to do with the introduction of macro-constraints. The all-new code to be discussed has primarily to do with the newly introduced surface syntax. It includes a top-level processing loop and a parser. The loop is responsible for reading user input, parsing and evaluating it, printing any results, and then iterating. Parsing occurs in two stages: the LISP function `read` reads in a string of characters and produces a LISP S-expression, which is then further processed by the parser to be presented here.

The top-level loop is shown in Table 8-8. It is typical of LISP interaction loops, and uses the MACLISP/Lisp Machine LISP conventions for "interaction variables". The variable + always holds the last thing typed in by the user, and ++ and +++ the two things before that. The variable - has the expression being processed. The variable // has a list of all the values returned as a rsult of evaluating the last expression, and * has the first of these values, ** and *** beign earlier instances of *. These variables are purely for user convenience, so that he can refer to partial results without losing them if he forgot to set some variable to the computed value.

The important thing about this loop is that it handles the user interaction, prompting, reading, processing, and printing. The two constants `@quit` and `@nothing` provide special control. One causes the loop to terminate, reverting to the LISP system; the other is a "magic value" that will not be printed. This allows a request such as `==` or `why` to print nothing, rather than **DONE** or `Q.E.D.` (which some users find annoying). Though those trivial changes are not shown here, functions such as `why` should in fact be altered to return `@nothing` after printing a message.

```
(defun evaluate-input (input)
  (cond ((or (atom input) (eq (car input) 'the))
         (eval (parse-thing input)))
        ((and (symbolp (car input)) (get (car input) 'request))
         (eval (cons (car input)
                     (forlist (x (cdr input)) (parse-thing x)))))
        ((eq (car input) 'defcon) (define-macro input))
        ((eq (car input) 'destroy)
         (dolist (x (cdr input)) (*destroy x)))
        ((eq (car input) 'lisp) (eval (cadr input)))
        (t (multiple-value-bind (creations equations definitions)
               (parse-statements (list input) () ())
             (dolist (stmt creations) (eval stmt))
             (dolist (stmt equations) (eval stmt)))
           @nothing)))

(dolist (x '(stats reset-stats variable queue-stats reset-queues
             run? disallow change retract forget dissolve detach
             disconnect disequate why why-ultimately what))
  (putprop x t 'request))
```

TABLE 8-9. Discrimination of Input Forms.

8.2.6. User Input Forms Are Divided into Three Categories

The top-level loop calls `evaluate-input` (Table 8-9) to process the user input. This function categorizes the input form as a *statement* (a `create`, `==`, or `vector` form), a *request*, or a descriptor for a *thing* (which at the top level is interpreted as a request for the value of that thing). Atoms and `the`-forms are things. A list whose first element is a symbol with a non-null `request` property is a request (note the definitions of such properties by the `dolist` form in Table 8-9); such requests are assumed to take "things" as their arguments. The `defcon` and `destroy` requests are handled specially, because they take things other than "things" as arguments. A list whose first element is `lisp` is an escape, so that LISP expressions can be evaluated easily from within the `consys` loop; this is a special user convenience, not strictly speaking part of the supported constraint language. Any other form is taken to be a statement.

8.2.7. Defining a Macro Generates a Macro-Constraint-Type

The function `define-macro` parses a `defcon` request of the form described in §8.1.2. After the pieces of the form have been picked out and checked, two "environment" structures are created, one for cells and one for constraints. Each environment is a list cell whose *cdr* is an a-list (the a-list for devices is initially empty). The *car* is not used for anything; the use of a header cell allows new entries to be added by using a side-effect. Each a-list pair consists of a name and a flag;

```
(defun define-macro (input)
  (let ((name (if (atom (cadr input)) (cadr input) (caadr input)))
        (symbol (if (atom (cadr input)) (cadr input) (cadadr input)))
        (pins (caddr input))
        (body (cdddr input)))
    (require-symbol name)
    (require-symbol symbol)
    (let ((conenv (list 'conenv))
          (cellenv (cons 'cellenv (forlist (p pins) (require-symbol p) (list p)))))
      (multiple-value-bind (creations equations definitions)
          (parse-statements body cellenv conenv)
        (gen-macro-constraint-type name
                                   symbol
                                   pins
                                   (forlist (x (cdr cellenv)) (car x))
                                   creations
                                   equations)))))
```

TABLE 8-10. Processing a Macro Definition.

in `conenv` the flag indicates whether a `create` for that device has been encountered yet, and in `cellenv` the flag is unused. (The environments have the same structure so that common routines can process them.) Initially all the pin names are in `cellenv`.

The body of a macro definition should be a list of statements, and these are parsed in the given environments. (The parsing process may alter the environment structures.) The statement parsing produces three results: a list of `create` forms, a list of `==` forms, and a list of `defcon` forms (which result only from `vector` forms, and can be ignored (as they are here) because they are processed when generated). The creations and equations, along with all the names now in `cellenv` are passed to `gen-macro-constrain-type`.

```
(defun gen-macro-constraint-type (name symbol pins allvars creations equations)
  (require-symbol name)
  (*destroy name)
  (let ((ct (make-macro-constraint-type)))
    (set name ct)
    (putprop symbol name 'ctypename)
    (setf (mctype-name ct) name)
    (setf (mctype-pins ct) (array-of pins))
    (setf (mctype-allvars ct) (array-of allvars))
    (setf (mctype-creations ct)
          (array-of (forlist (c creations)
                      (or (eq (car c) 'create)
                          (lose "Non-creation ~S for GEN-MACRO-CONSTRAINT-TYPE." c))
                      (or (and (boundp (caddr c))
                               (or (constraint-type-p (symeval (caddr c)))
                                   (macro-constraint-type-p (symeval (caddr c)))))
                          (lose "Not a defined constraint-type ~S." (caddr c)))
                      (list (cadr c) (symeval (caddr c)) (cadddr c)))))
    (let ((codename (gen-name name 'connector)))
      (setf (mctype-connector ct) codename)
      (fset codename `(named-lambda ,codename (*me*)
                                    (let ((*run-flag* ()))
                                      ,@equations
                                      (run?))))
      (compile codename))
    ct))
```

TABLE 8-11. Generating a Macro-Constraint-Type.

This function (Table 8-11) generates a macro-constraint-type data structure. The *name*, *pins*, and *allvars* slots are filled in. The list of creations is pre-processed, in that the keyword `create` is removed, and the name of the type to be instantiated is replaced by the data structure for the type itself, which must be a constraint-type or a macro-constraint-type. Finally, the connector function is constructed from the list of equations. These equations will all refer to a local variable `x` as "`(my x)`", using the `my` macro of Table 8-7, so all that is needed is to execute these equations where the LISP variable `*me*` is defined. Also, the `*run-flag*` is bound to `()` to prevent propagation from occurring until the whole network is wired, to avoid wasted effort. Once a lambda-expression (actually a Lisp Machine LISP "named-lambda" expression) has been constructed, it is assigned to the "function cell" of a generated LISP symbol, and then the `compile` function is applied. The result is that the connector function is a compiled LISP function. (The call to `compile` could be omitted, and everything would still work, only more slowly.)

```
(declare (special *inputs* *equations* *creations* *definitions*))

(defun parse-statements (inputs cellenv conenv)
  (do ((*inputs* inputs)
       (*equations* '())
       (*creations* '())
       (*definitions* '()))
      ((null *inputs*)
       (dolist (x (cdr conenv))
         (or (cdr x) (format t "~&;Warning: constraint ~S not defined." (car x))))
       (return *creations* *equations* *definitions*))
    (let ((stmt (pop *inputs*)))
      (cond ((atom stmt) (lose "Internal error: atomic statement ~S." stmt))
            ((eq (car stmt) '==)
             (let ((things (forlist (z (cdr stmt))
                             (parse-thing z cellenv conenv ()))))
               (do ((th things (cdr th)))
                   ((null th))
                 (dolist (x (cdr th)) (push `(== ,(car th) ,x) *equations*)))))
            ((eq (car stmt) 'create)
             (push stmt *creations*)
             (and conenv
                  (let ((slot (assq (cadr stmt) (cdr conenv))))
                    (cond ((null slot) (push (cons (cadr stmt) t) (cdr conenv)))
                          ((null (cdr slot)) (rplacd slot t))
                          (t (lose ";Constraint ~S multiply created."
                                   (cadr stmt))))))))
            ((eq (car stmt) 'vector)
             (parse-vector (gen-name 'vector)
                           (cadr stmt) (caddr stmt) (cadddr stmt) (cddddr stmt)))
            ((and (not (atom (car stmt))) (eq (caar stmt) 'vector))
             (parse-vector (cadar stmt)
                           (cadr stmt) (caddr stmt) (cadddr stmt) (cddddr stmt)))
            (t (parse-constraint stmt t))))))
```

TABLE 8-12. Parsing Statements.

8.2.8. Statements Are Reduced to Simple Statements

The function `parse-statements` (Table 8-12) maintains a queue `*inputs*`, which is a queue of statements to be processed. The results will be a list of equations, a list of creations, and a list of (already processed) definitions (which is returned primarily so that `parse-statements` can be tested independently of the rest of the system and the results examined).

The `==` statement is generalized so that more than two things can be equated; each thing is directly equated to every other thing (so that equating n things results in $\frac{n(n-1)}{2}$ binary equatings). The things are all parsed using `parse-thing`.

A `create` statement is output to the `*creations*` list, and an entry is located or created in `conenv` if that environment is not null (the environments are null for top-level statements and non-null when parsing statements for a macro body).

```
(defun parse-constraint (form stmtp)
  (cond ((symbolp (car form))
         (parse-constraint-pins
             (car form) () (cdr form) stmtp))
        ((and (not (atom (car form)))
              (symbolp (caar form))
              (symbolp (cadar form))
              (null (cddar form)))
         (parse-constraint-pins
             (caar form) (cadar form) (cdr form) stmtp))
        (t (lose "Unknown form: ~S." form))))

(defun parse-constraint-pins (ctypesym userconname arguments stmtp)
  (require-symbol ctypesym)
  (and userconname (require-symbol userconname))
  (or stmtp (memq '% arguments) (push '% arguments))
  (let ((ctypename (get ctypesym 'ctypename)))
    (or (and ctypename (symbolp ctypename))
        (lose "~S is not the symbol for any constraint-type." ctypesym))
    (let ((ctype (symeval ctypename))
          (conname (or userconname (gen-name ctypename))))
      (or (constraint-type-p ctype)
          (macro-constraint-type-p ctype)
          (lose "Unknown constraint type: ~S." ctypename))
      (let ((pinarray (if (constraint-type-p ctype)
                          (ctype-vars ctype)
                          (mctype-pins ctype))))
        (let ((args (cond ((= (length arguments) (array-length pinarray)) arguments)
                          ((= (length arguments) (+ (array-length pinarray) 1))
                           (reverse (cdr (reverse arguments))))
                          (t (lose "Wrong number of arguments to ~S: ~S."
                                   ctypename arguments))))
              (info (and (not (= (length arguments) (array-length pinarray)))
                         (car (last arguments)))))
          (push `(create ,conname ,ctypename ,@(and info (list info))) *inputs*)
          (let ((result ()))
            (do ((j 0 (+ j 1))
                 (a args (cdr a)))
                ((= j (array-length pinarray)))
              (cond ((eq (car a) '%)
                     (cond (result
                                (lose "Multiple %'s to ~S: ~S." ctypename arguments))
                           ((not stmtp)
                            (setq result `(the ,(aref pinarray j) ,conname)))
                           (t (lose "Statement fed % to ~S: ~S."
                                    ctypename arguments))))
                    ((not (eq (car a) '?))
                     (push `(== (the ,(aref pinarray j) ,conname) ,(car a))
                           *inputs*))))
            result))))))
```

TABLE 8-13. Parsing an "Algebraic Expression".

Vectors are farmed out to `parse-vector`. All other forms are assumed to be network descriptions expressed in the nested algebraic form.

The function `parse-constraint` (Table 8-13) determines whether or not the constraint to be generated has been given a name by the user. The function `parse-constraint-pins` deals with the details of the `%` and `?` conventions, performs error checking, and then decomposes the form into equivalent `create` and `==` statements which are then enqueued on `*inputs*` for re-processing. The flag `stmtp` is true iff the given form is a statement (implying that no `%` is permitted), in which case `()` is returned. If `stmtp` is false, then the value is a name for the pin corresponding to the (explicit or implicit) occurrence of `%`.

```
(defun parse-thing (thing &optional (cellenv ()) (conenv ()) (simplep t))
      (cond ((numberp thing) `(constant ,thing))
            ((symbolp thing)
             (parse-recursive-symbol (get-pname thing) cellenv conenv thing))
            ((atom thing) (lose "Unknown atomic thing: ~S." thing))
            ((and (memq (car thing) '(default parameter))
                  (fixp (cadr thing))
                  (null (cddr thing)))
             thing)
            ((eq (car thing) 'global)
             (parse-global-symbol (cadr thing) t))
            ((and (eq (car thing) 'the) (cddr thing))
             (parse-the (cdr thing) conenv))
            ((not simplep)
             (parse-thing (parse-constraint thing ()) cellenv conenv ()))
            (t (lose "Non-simple thing: ~S." thing))))
```

TABLE 8-14. Parsing a Reference to a Thing.

```
(defun parse-recursive-symbol (pname cellenv conenv thing)
  (let ((pos (string-reverse-search-char #/. pname)))
    (cond ((null pos)
           (parse-simple-symbol (or thing (intern pname))
                                (if thing cellenv conenv)
                                (not (null thing))))
          ((zerop pos)
           (parse-global-symbol (intern (substring pname 1)) (not (null thing))))
          (t `(the ,(intern (substring pname (+ pos 1)))
                   ,(parse-recursive-symbol
                        (substring pname 0 pos) cellenv conenv ()))))))
```

TABLE 8-15. Parsing a Pathname Written with Periods.

8.2.9. Pathnames with Periods Are One of Many Forms of Reference

The function `parse-thing` reduces a reference to a "thing" to either a simple variable name, a pathname, or a `constant` or similar form. A number is converted to a `constant` form, so that one may write `(+ x 3)` rather than `(+ x (constant 3))`. Symbols are examined for periods by `parse-recursive-symbol`. A `default` or `parameter` form stands as written. A `global` form refers to a global variable, and is given to `process-global-symbol`. A `the` form has its own processor. Anything else is regarded as a nested algebraic expression, which must have an explicit or implicit `%` in it; `parse-constraint` is used to parse that form and return the name of a pin, which is then (for generality) re-processed by `parse-thing`.

```
(defun parse-simple-symbol (sym env cellp)
  (require-symbol sym)
  (cond ((null env) (parse-global-symbol sym cellp))
        ((assq sym (cdr env)) `(my ,sym))
        (t (push (list sym) (cdr env)) `(my ,sym))))
```

TABLE 8-16. Parsing a "Simple" (Ha!) Symbol.

Atomic symbols are pulled apart by `parse-recursive-symbol` (Table 8-15). If a "." is found in the print name of the symbol, then the symbol is divided into two parts, the part before the *last* "." and the part after it. The part after is the selector for a `the` form, and the part before is recursively parsed. As an efficiency trick, `thing` is passed in, so that if the name contains no "." then the thing can be returned directly without the expense of a call to `intern`. Also, a leading "." indicates a global symbol, as discussed in §8.1.3.

When not within a macro body, all symbols are global. Within a macro body, a simple symbol (not explicitly made global by a leading "." or use of the `global` form) is local, implying that it must be entered into the environment and that it should be referred to as "my" (that is, the macro's) symbol. The function `parse-simple-symbol` (Table 8-16) performs these operations.

```
(defun parse-global-symbol (sym cellp)
  (require-symbol sym)
  (cond ((not (boundp sym))
         (cond (cellp
                  (putprop sym t 'special)        ;compiler nonsense
                  (set sym (gen-cell sym)))))
        ((not cellp)
         (or (constraint-p (symeval sym))
             (macro-constraint-p (symeval sym))
             (format t "~&;Warning: ~S has a non-constraint value ~S."
                     sym (symeval sym))))
        (t (or (and (cell-p (symeval sym))
                    (globalp (symeval sym)))
               (format t "~&;Warning: ~S has a non-cell value ~S."
                       sym (symeval sym)))))
  sym)
```

TABLE 8-17. Parsing a Global Symbol.

```
(defun parse-the (list conenv)
  (if (null (cdr list))
      (parse-simple-symbol (car list) conenv ())
      `(the ,(car list) ,(parse-the (cdr list) conenv))))
```

TABLE 8-18. Parsing a `the` Expression.

If a global symbol has no (LISP) value, then a cell is automatically generated for it in `parse-global-symbol` (Table 8-17). This relieves the user of the constraint language from having to declare all his variables. If it does have a value, then it had better be the desired kind of object (a cell or a constraint, as determined by the flag `cellp`).

The only purpose of the function `parse-the` (Table 8-18) is to allow extended pathnames of the form `(the a b c ...)`. This could just as easily have been put into the LISP macro definition of `the`, but I thought it would be more appropriate to do it here, as part of the move away from dependence on the LISP evaluator.

8.2.10. Vectors Are Easily Defined in Terms of Macros

The function `parse-vector` (Table 8-19) reduces a `vector` statement to equivalent statements and two macro-constraint-type definitions, one for the body and one for the vector of the body. The `define-macro` function of Table 8-10 is applied to the two definitions to cause the macro-constraint-types to exist immediately; unfortunately, this must be done before the other statements can even be parsed properly. An example of the results of processing a `vector` form appears in §8.1.4.

```
(defun parse-vector (vectorname size interface common body)
  (require-integer size)
  (let ((vectortypename (gen-name 'vector-type))
        (bodyname (gen-name 'vector-body))
        (vectordevices (fortimes (j size) (intern (format () "~D" j)))))
    (do ((x interface (cdr x))
         (lowpins '() (cons (cadar x) lowpins))
         (highpins '() (cons (caddar x) highpins)))
        ((null x)
         (let ((bodydef
                 `(defcon ,bodyname ,(append lowpins highpins common) ,@body))
               (vectordef `
                   (defcon ,vectortypename ,(append lowpins highpins common)
                     ,@(do ((d vectordevices (cdr d))
                            (low lowpins (forlist (h highpins) `(the ,h ,(car d))))
                            (s '() (cons `((,bodyname ,(car d))
                                           ,@low
                                           ,@(if (cdr d)
                                                 (forlist (ignore highpins) '?)
                                                 highpins)
                                           ,@common)
                                         s)))
                           ((null d) (reverse s))))))
           (define-macro bodydef)
           (define-macro vectordef)
           (push bodydef *definitions*)
           (push vectordef *definitions*))
         (push `((,vectortypename ,vectorname)
                 ,@(forlist (x interface) (car x))
                 ,@(forlist (x interface) (cadddr x))
                 ,@common)
               *inputs*)))))
```

TABLE 8-19. Parsing a `vector` Statement.

8.3. Example of the Use of Macro-Constraints

Here we will define a macro-constraint-type which will constrain one number to be the `gcd` of two others, using the formulation of §1.1.1. It will use a vector, the body of which performs one step in the simplified (using subtraction rather than division) Euclidean algorithm. We will start in the LISP system, and enter `consys`.

```
(consys)
;Welcome to The Constraint System.
]:(defcon gcd7 (x y g)
    ((vector v) 7 ((x qin qout end1) (y rin rout end2)) (g)
     (== rout qin)
```

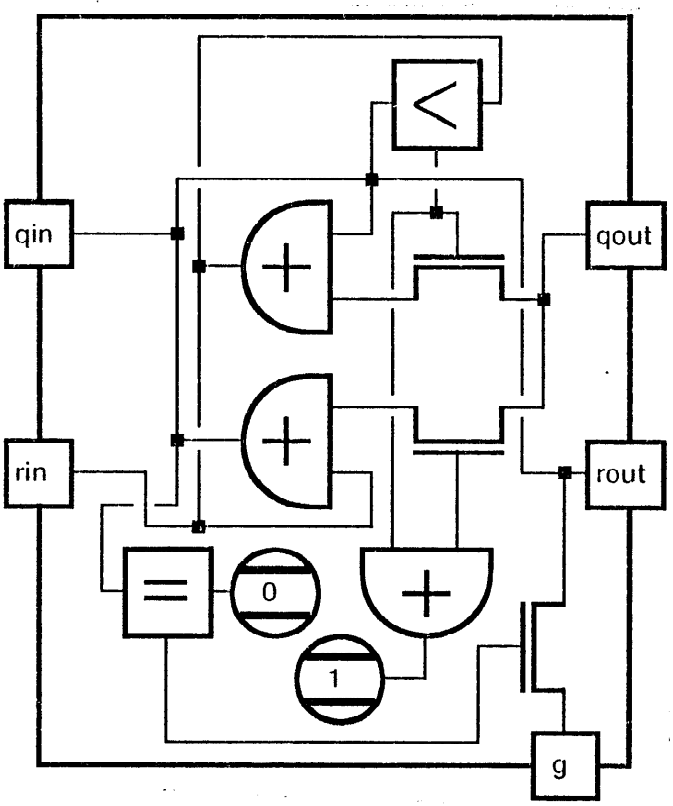


FIGURE 8-4. One Stage of a GCD Computation.

```
      (<! p qin rin)
      (gate p qout (+ rin qin %))
      (gate (+ 1 p %) qout (+ qin rin %))
      (gate (= qout 0) rout g)))
<Macro-constraint-type GCD7>
```

The "]:" is the (rather silly-looking) prompt for user input to the constraint system. Our first input defines a macro-constraint-type `gcd7` which constrains `g` to be the greatest common divisor of `x` and `y` *provided* that it can be found in seven substraction steps or fewer. (We will have more to say later about this arbitrary limitation!) It was certainly written with a functional view in mind, and so I shall describe it that way; but it is written in a constraint language, and so of course can "run backwards" to the extent permitted by its structure and the local propagation technique.

An instance of `gcd7` is a vector of seven steps. Each step (see Figure 8-4) has two pins on the left called `qin` and `rin` and two on the right called `qout` and `rout` (which connect to the `qin` and `rin` of the next step to the right). In each stage, `rout` is equal to `qin`, and `qout` is equal to either the difference between `qin` and `rin` or the difference between `rin` and `qin`. The value

of `p` determines which one is used; `p` is derived by comparing `qin` and `rin` in such a way that `qout` will be the *positive* difference. Finally, `g` is equal to `rout` if `qout` is zero. It is easy to see the the gcd of `qout` and `rout` is the same as that of `qin` and `rin`, and that if `qout` is zero then `rout` must in fact be the gcd.

The boundary conditions are that the `qin` and `rin` of the leftmost stage are equated to `x` and `y` respectively. The variables `end1` and `end2` could have been question marks; they are not used for anything in particular, but using names will allow us to examine them.

Now let us create an instance of `gcd7` called `foo`, and set `x` to **6** and `y` to **10**.

```
]:((gcd7 foo) 6 10 answer)
]:answer
<CELL-79 (ANSWER) SLAVE 2>
```

Note that no result was printed for the creation of the instance (because the top-level loop saw the "magic value" `@nothing`). The result `answer` is indeed **2**.

We can now look at various cells of the network and inspect their values.

```
]:foo.end1
<CELL-81 (FOO.END1) SLAVE 2>
]:foo.end2
<CELL-83 (FOO.END2) SLAVE 0>
```

The two final values are 2 and 0, which is consistent with a gcd equal to 2.

```
]:foo.v.0.qout
<CELL-473 (FOO.V.0.QOUT) SLAVE 4>
]:foo.v.1.qout
<CELL-413 (FOO.V.1.QOUT) SLAVE 6>
]:foo.v.2.qout
<CELL-353 (FOO.V.2.QOUT) SLAVE 2>
]:foo.v.3.qout
<CELL-293 (FOO.V.3.QOUT) SLAVE 4>
]:foo.v.4.qout
<CELL-233 (FOO.V.4.QOUT) SLAVE 2>
]:foo.v.5.qout
<CELL-173 (FOO.V.5.QOUT) SLAVE 2>
]:foo.v.6.qout
<CELL-109 (FOO.V.6.QOUT) SLAVE 0>
```

This is the sequence of intermediate values computed. (Compare this with sequences in §1.1.1). Note how easily we can refer to parts of parts of a vector, using the pathname notation. (We can also examine constraints as well as cells:

```
]:foo.v
<FOO.V:VECTOR-TYPE-65>
```

```
]:foo.v.4
<FOO.V.4:VECTOR-BODY-66>
```

though that is not officially part of the language.)

Now let us examine how the value for `answer` was deduced.

```
]:(why answer)
;The value 2 is in ANSWER because it is connected to (THE FOO V |6| GATE-73 B)
;   and <FOO.V.6.GATE-73:GATE> computed it by rule <B←GATE-RULE-21(P,A)>
;   from: CELL-113 (P) = 1, CELL-115 (A) = 2.
```

Again, no "return value" (`Q.E.D.`) is printed now because `why` returns `@nothing`. The output could be cleaned up a little by changing `cell-goodname` to return not a `the`-style pathname but a name with periods in it.

```
]:(why-ultimately answer)
;The value 2 is in ANSWER because it is connected to (THE FOO V |6| GATE-73 B)
;   and it was ultimately derived.
```

(It says that it was "ultimately derived" because only constants were involved, and constants are now omitted from the set of premises for a deduction.) A number of connections were involved:

```
;These connections were involved:
;   (THE FOO V |6| EQUALITY-74 B) == (CONSTANT 0),
;   (THE FOO V |6| GATE-70 B) == (THE FOO V |6| ADDER-71 B),
;   (THE FOO V |5| GATE-68 B) == (THE FOO V |5| ADDER-69 B),
;   (THE FOO V |4| GATE-70 B) == (THE FOO V |4| ADDER-71 B),
;   (THE FOO V |3| GATE-68 B) == (THE FOO V |3| ADDER-69 B),
;   (THE FOO V |2| GATE-70 B) == (THE FOO V |2| ADDER-71 B),
;   (THE FOO V |1| GATE-68 B) == (THE FOO V |1| ADDER-69 B),
;   (THE FOO V |0| GATE-70 B) == (THE FOO V |0| ADDER-71 B),
;   (THE FOO V |0| ADDER-72 C) == (CONSTANT 1),
;   (THE FOO X) == (CONSTANT 6),
;   (THE FOO V RIN) == (THE FOO X),
  [Forty connections omitted.]
;   (THE FOO V |6| ADDER-72 A) == (THE FOO V |6| P),
;   (THE FOO V |6| GATE-70 P) == (THE FOO V |6| ADDER-72 B),
;   (THE FOO V |6| QOUT) == (THE FOO V |6| GATE-70 A),
;   (THE FOO V |6| EQUALITY-74 A) == (THE FOO V |6| QOUT),
;   (THE FOO V |6| GATE-73 P) == (THE FOO V |6| EQUALITY-74 P),
;   (THE FOO V |6| G) == (THE FOO V |6| GATE-73 B),
;   (THE FOO V G) == (THE FOO V |6| G),
;   (THE FOO G) == (THE FOO V G),
;   ANSWER == (THE FOO G).
```

Now let us get up the courage to ask what the entire computation was! (I have taken the liberty of reformatting the output by inserting white space and line breaks.)

```
]:(what answer)
;The value 2 in ANSWER was computed in this way:
;   ANSWER ← (GATE (= (GATE (+ 1 (<! (THE FOO V |5| GATE-68 A)
                                     (THE FOO V |4| GATE-70 A))
                                  %)
                            %
                            (+ (THE FOO V |5| GATE-68 A)
                               (THE FOO V |4| GATE-70 A)
                               %))
                         0)
                   (THE FOO V |5| GATE-68 A)
                   %)
;   (THE FOO V |4| GATE-70 A) ←
       (GATE (+ 1 (<! (THE FOO V |3| GATE-68 A) (THE FOO V |2| GATE-70 A)) %)
             %
             (+ (THE FOO V |3| GATE-68 A) (THE FOO V |2| GATE-70 A) %))
;   (THE FOO V |2| GATE-70 A) ←
       (GATE (+ 1 (<! (THE FOO V |1| GATE-68 A) (THE FOO V |0| GATE-70 A)) %)
             %
             (+ (THE FOO V |1| GATE-68 A) (THE FOO V |0| GATE-70 A) %))
;   (THE FOO V |0| GATE-70 A) ← (GATE (+ 1 (<! 10 6) %) % (+ 10 6 %))
;   (THE FOO V |1| GATE-68 A) ←
        (GATE (<! (THE FOO V |0| GATE-70 A) 10)
              %
              (+ 10 (THE FOO V |0| GATE-70 A) %))
;   (THE FOO V |3| GATE-68 A) ←
        (GATE (<! (THE FOO V |2| GATE-70 A) (THE FOO V |1| GATE-68 A))
              %
              (+ (THE FOO V |1| GATE-68 A) (THE FOO V |2| GATE-70 A) %))
;   (THE FOO V |5| GATE-68 A) ←
       (GATE (<! (THE FOO V |4| GATE-70 A) (THE FOO V |3| GATE-68 A))
             %
             (+ (THE FOO V |3| GATE-68 A) (THE FOO V |4| GATE-70 A) %))
```

For all the use of "algebraic" notation, this is fairly hard to wade through! Part of the problem is that the explanation system doesn't take advantage of the macro-call hierarchy to produce summary explanations. One would like an explanation to go something like "The answer was computed by `foo.v.6` from its `qin` and `rin`, which it got from `foo.v.5`, and so on, down to `foo.v.0` which got its inputs from `foo.x` and `foo.y`."

8.4. Discussion of the Macro Language

In this chapter we have added to the constraint language an abstraction capability in the form of a simple macro mechanism, a limited iteration feature, and a front-end command processing

loop and parser to permit some useful syntactic abbreviations. I am pleased with the front end, for the most part, but the macro and iteration features are clearly deficient compared with what a useable constraint language requires. Here I discuss these deficiencies and possible solutions.

Every time a macro-constraint-type is instantiated, a complete copy is made of the defining network structure. This is wasteful of space; it is as if every time a constraint-type were instantiated a copy were made of all the rules. Now certainly new cells must be created for each macro-instance, because they hold values that are different for each instance. The reason a complete copy must be made is that presently the connectivity information is also stored in the cells. Cells point at devices and other cells, which in turn point back, and these back-pointers therefore require individual copies of each device data structure. It ought to be possible to re-design the data structures in such a way that the macro-constraint-type contains a single copy of the connectivity and device information as a full network with full back-pointers (presently that information is stored, but as directions for construction, not as a network). Then an instance would contain only an array of cells, which would determine their connectivity by referring to the "network template" in the macro-constraint-type. This is entirely analogous to the situation with constraints and constraint-types. (Of course, this idea trades time for space in requiring indirection to the constant, shared template. Indeed, Borning seems to do something similar to this in THINGLAB [Borning 1979]. But such sharing may not be desirable in a multi-processor constraint language implementation.)

A more important problem is that when a macro-constraint is created, all of its parts must first be fully instantiated (in the current implementation). This makes it impossible to write recursively defined constraints. One might like, for example, to write a factorial constraint:

```
(defcon factorial (f n)
  (= p n 0)
  (gate p f 1)
  (+ 1 p notp)
  (factorial (gate notp f %) (+ (gate notp n %) % 1)))
```

This cannot work in the current implementation because in the process of creating an instance of `factorial` another instance of `factorial` must be created, and so there is indefinite regress. This is a standard problem with any macro-type language. It is analogous to a programming language in which all procedure calls are replaced by the code for the called procedure (procedure integration) before any part of the program is run. What is needed is a way to instantiate a macro only partially, then compute using some of its parts, and then create the rest of its parts only when necessary. One possible heuristic is never to instantiate a sub-macro unless at least one argument has propagated to the call to it (the assumption being that it won't generate values without input—not always true when the `assume` construct is used!). This would allow the definition of `factorial` given above to operate properly. If one said `(factorial answer 3)`, then one instance of `factorial` would be made, containing an equality test, a gate, an adder, and

so on, plus a dummy instance (a "procedure *call*") of `factorial` that has not yet actually been created. The value 3 propagates from `n`, producing zero for `p`, and so `f` is not gated to 1, but instead `n` is gated to the second adder to calculate $3 - 1 = 2$. This is then visible on a pin of the dummy instance, and so the recursive instance of `factorial` is created at this point to replace the dummy instance (and this actual instance itself now contains another dummy instance). After two more steps there are four actual instances of `factorial` nested, with the innermost containing yet another dummy instance. This last one is not actually created, however, because `n` is zero and so the gates prevent any values from propagating to the pins of the dummy instance.

I have constructed a constraint system that operates in this manner, and it has successfully run recursive constraint networks such as the definition of `factorial` given above. It is not of the same "lineage" as the systems presented here, however, but an offshoot of earlier, less tractable versions, and so I do not present the code here. Also, that version did not have retraction capabilities; except for the ability to handle recursive constraints, it was approximately equivalent to the system of Chapter Three. (I attempted to add retraction capabilities, but that interacts in extremely complicated ways with dummy instances. I chose to abandon that path to concentrate on the use of dependencies and on dealing with networks containing multiple contradictions.)

An obvious problem with the `vector` construct is the restriction that the size of the vector be fixed. One would like to have the size specified by a true constraint variable. One could even set up a general `gcd` program that would use a vector of indefinite length; when the gcd computation was done, the size of the vector would have been determined by the computations of the body (trying to satisfy a boundary condition)! This would be very powerful. Implementing this is roughly the same as implementing recursive constraints; one needs a way to avoid creating instances of things until it is clear they they are really needed. In this case, no instances of a vector body would be created until one was needed. (This is analogous to a `while-do` loop: rather than creating all the copies of the loop body that will be needed at run time (unrolling the loop), before each time the loop body is executed a run-time check is made to determine whether it needs to be.) One difficulty that does arise with vectors is the situation where a vector 5 long is created, and then the value 5 is retracted and 3 substituted: two copies of the body must go away, or at least become ineffective (the latter course perhaps being more economical implementationally if there is a chance that the 3 may become a 5 again). This requires the ability to retract or suppress network connections or constraints; this ability is not provided by the current constraint system.

A song not for now you need not put stay ...
A tune for the was can be sung for today ...
The notes of the does-not will sound as the does ...
Today you can sing for the will-be that was.
—Walt Kelly (1953)
Ten Ever-Lovin' Blue-Eyed Years with Pogo

Chapter Nine

Compilation

THE PURPOSE OF COMPILATION is to trade more work now for less work later, by expending effort now to reduce an object to a form more easily dealt with later. In the case of our constraint system, we seek to reduce a macro-constraint definition to the definition of a primitive constraint.

I present here a simple compilation technique. While the idea is simple, the details are even more tedious than usual, and so I shall not present the code for the compiler here. The compiler is similar in flavor to the one described in [Borning 1979], and also bears some resemblance to the code-construction tecnhiques used in [Brown 1980].

The compiler takes a macro-constraint-type definition and creates an instance of it, in order to have a network structure on which to operate. It then performs a propagation-like process on the network.

Suppose the macro-constraint-type to have n pins. Then the compiler performs 2^n passes, one for each possible subset of the pins. (This exponential may seem horrendous, but I have compiled macro-constraints with nine pins in only a short time—less than thirty seconds.) For each subset, those pins are marked "given", and then pseudo-values are propagated throughout the network. A marker is actually a list of pins, and each given pin is marked with a list of itself. A rule may be used if markers are present on all its triggers, in which case the union of the marker sets is used to mark the output pin (because all those values went into the deduction of that value). If a marker reaches a pin, and the marker is the set of all the given pins, then a rule may be constructed relating the output pin to the input pins. This is done by tracing back through the "dependencies" maintained during the pseudo-propagation, welding together the LISP code for the various rules

used in the propagation process. (If a marker reaches a pin and is not the set of all given pins, no rule is constructed, because that rule will be obtained on another pass.) If two markers meet at a node, then a detector rule may be constructed that signals a contradiction if the two values are not equal.

Assumption cells (and `&nogood` rules in general) cause difficulties because their cells may not be "compiled out" and converted to LISP variables. This is because the cell structure is needed to record nogood sets. My solution to this (which I have not yet implemented—the current compiler simply doesn't handle `&nogood` rules) is to artificially move interior nodes supplied by `&nogood` rules to the "boundary" of the constraint, making them pseudo-pins. Then they can have regular cell structures, but they are not real pins in that they are not ordinary connection points.

As an example of the results of this compilation technique, consider our old standby, the temperature converter (this definition is taken from §8.1.2):

```
(defcon temp-converter (f c)
        ((+ add) f ((* othermult) ((* mult) 9 c) 5 %) 32))
```

The result of compilation is the following primitive constraint definition:

```
(DEFPRIM TEMP-CONVERTER (F C)
         (C (F)
            (PROG FOO ()
                  (RETURN (LET ((A (* (- 32 F) 5))
                                (C 9))
                            (IF (AND (NOT (ZEROP A)) (ZEROP ( C A)))
                                (// C A)
                                (RETURN-FROM FOO @DISMISS))))))
         (F (C)
            (PROG FOO ()
                  (RETURN (+ 32
                             (LET ((A (* C 9))
                                   (C 5))
                               (IF (AND (NOT (ZEROP A)) (ZEROP ( C A)))
                                   (// C A)
                                   (RETURN-FROM FOO @DISMISS)))))))))
```

There are two rules; one computes `c` from `f` and the other computes `f` from `c`. Notice that where possible straightforward LISP computations are used, as in `(* (- 32 f) 5)` in the rule for computing `c` from `f`. If a value is to be used more than once, then a `let` form is used to name the value. The `prog` forms are necessary so that if a `@dismiss` or `@lose` operation occurs, an immediate exit (via `return-from`) can be taken. If any part of a computation dismisses, then the whole thing dismisses; if any part loses, the whole thing loses.

Mister Middle in the meadow
Riddled 'round with rain.
Puzzle you the pitter-pat
What not goes up again?
Riddle you the little dew
And little do you do?
Little did is little done.
Tho' little did'll do.
—Walt Kelly (1959)
The Pogo Sunday Brunch

Chapter Ten

Conclusions

THE RESEARCH DISCUSSED IN THIS DISSERTATION has resulted in the construction of a constraint language system of fair complexity. It certainly does not have all the characteristics one could hope for; it is not even a combination of all the characteristics which have been separately achieved by previous systems. It is, however, a fairly efficient version that is perhaps closer to being viable for a multiprocessing implementation than any other so far.

The system presented here performs computations on networks of relationships by local propagation, using one-step local deductions. The history of the computation is maintained in the form of dependency information, indicating which values were derived from which others. This information can be used to explain the computation, in whole or by stages, and to guide the automatic or semi-automatic handling of contradictions or changes to the network parameters. This uses the technique of dependency-directed backtracking, which is shown to be superior to the usual chronological backtracking in many cases. An assumption mechanism is provided to allow guesses and default values, and a resolution mechanism plus recording of derived premises as nogood sets allows derived constraints to limit the explosion of combinatorial search.

The implementation of the system reflects the structure of the visual image of constraints as connected devices which gives this paradigm its intuitive power. This leads to some complexity because of the use of data structures with pointers to each other. However, once a network is constructed, propagation of values is very fast, and yet the structure of a network can be altered in mid-computation without invalidating the semantics of the language.

A primitive abstraction capability (macros) is provided, and a simple compiler for these macros can reduce them to primitive operators.

10.1. Comparisons with Other Work

10.1.1. SKETCHPAD Relaxed Constraints on Geometric Diagrams

The SKETCHPAD system [Sutherland 1963] was in many ways ahead of its time. It provided graphic display output, a user interface not limited by the "Model 33 bottleneck", and automatic satisfaction of constraints. A technique amounting to pre-compilation of local propagation paths was used (undoubtedly similar to the methods of Chapter Nine), which Sutherland called "the one-pass method". Where that failed, a relaxation method was used, with each constraint being represented by a simple subroutine which would calculate an error value as a measure of how "unhappy" that constraint was with the existing values. Explicit dependency information was not used; relaxation solved global conflicts. This was possible because the geometric domain of SKETCHPAD is continuous, and all the constraints were equalities among linear relationships of variables, so the arithmetic computations were well-behaved.

Macro-structures could be built within SKETCHPAD and instantiated. Such structures had some of the properties of primitive objects, such as designated points of attachment. Moreover, non-primitive objects could be identified (merged), in which case sub-parts would be recursively merged. The operation of merging was apparently irreversible, however.

Given that constraints were used as early as 1962, why were not these ideas explored further, rather than waiting ten to fifteen years? One might speculate that the ideas were tied to graphics, as constraints seem to be most suited for describing objects; and furthermore that the advent of timesharing suppressed the development of graphics for a long while (because smooth graphics support tends to require steady computational service and therefore a dedicated processor). This is pure speculation, of course; but witness the growth of graphics now that personal computers are widespread!

10.1.2. Data Flow Computations Use Parallel Directional Devices

The data flow languages and architectures proposed by Dennis and his colleagues [Dennis 1973] [Dennis 1975] [Arvind 1978] are very closely related to constraint languages in that the computation is organized as a network of processors operating in parallel on data which moves asynchronously along wires between the devices. In the data flow paradigm, however, wires are pre-assigned directions along which the data flows. The purpose of data flow is to express the sort of directional computations ordinary programming languages handle, without the extraneous

sequencing conditions which they impose by their over-restrictive control structures. The data flow network itself expresses the necessary and sufficient sequencing for the computation. A device computes a result as soon as its arguments are available.

Data flow networks do not record dependencies. As noted in Chapter Three, however, dependency information can consist largely of knowledge about in which direction data flowed along each wire—and this is fixed in a data flow network anyway.

Dennis has proposed specific architectures for executing data flow programs efficiently. [Dennis 1975] A constraint architecture is of necessity more complex because of its lack of prior commitment to the direction of data flow along any given wire. In effect, a single constraint network represents an entire class of data flow programs, one for each partitioning of the network's terminals into input and output terminals; the constraint system then, in effect, determines dynamically which data flow computation to perform. Therefore it is likely that advances in the theory of data flow languages may be used in the implementation of constraint systems. The language VAL [Ackermann 1979], which is an algorithmic language in the style of algebraic languages but permitting flexible handling of sets of values (in particular allowing a function to return more than one value without resorting to assignment of refeerence parameters) is of especial interest.

10.1.3. Waltz's Algorithm Filters Scene Labels by Local Propagation

A local propagation technique is used in [Waltz 1972] (and described also in [Winston 1974] and [Winston 1977]) to limit the combinatorial search for Huffman-style labellings of visual scenes represented as line drawings. A line drawing itself forms a network structure; the goal is to assign a labelling to each junction. The Waltz filtering algorithm propagates information only along lines between junctions, and computation is made at each junction only on the basis of information flowing in along these lines. Waltz found that this technique, while performing only local propagation and therefore unable to resolve global ambiguities, would in practical cases usually converge to an unique solution or one with very few alternatives to be resolved by global analysis. Moreover, it tends to converge quickly, in time closer to linear than exponential in the size of the network. Waltz also realized the possibilities for parallel computation in this formulation. (A movie which Waltz made, well-known in AI circles, shows graphically the information propagating from vertex to vertex, with ambiguity factors at each vertex rapidly decreasing from the thousands to number like 1 or 2.)

Waltz's representation has the advantage of simultaneously representing *all* valid states of the system; it has the disadvantage of maintaining no dependency information. This can be a problem if the network is ambiguous. For example, suppose that two vertices each have two possible labellings. One might think this would indicate four possible states of the network, but the labellings might be correlated in such a way that choosing a label for one vertex forces the choice for the

other. Waltz's system has no way to represent such correlations. (On the other hand, the propagation technique typically does reduce the number of cases to a number feasible to enumerate and explicitly check in order to eliminate miscorrelated cases, which was all Waltz needed.)

10.1.4. Semantic Networks Propagate Symbolic Tags

There is a line of research stemming from Quillian's work on semantic nets which deals with the propagation of symbolic tags, rather than computational quantities, within a network. The distinction I draw here is rather fuzzy, but in propagating symbolic tags it is the fact that two or more tags collide somewhere that is of interest, whereas with computational quantities the arrival of a single value at a node is of interest. That is, a value carries meaning of its own, while tags are not very interesting in themselves, but bear meaning derived from the places they were first inserted into the network.

Quillian's semantic nets [Quillian 1968] consisted of a set of nodes representing primitive concepts with pointers among them. The meaning of a node consists precisely of the sum total of its relationships to other nodes. Quillian's system could compare two concepts by propagating two symbolic tags from the concepts and analyzing the points where they met.

Grossman used constraint expressions to represent complex data base relationships, including but not limited to unions, intersections, and partitionings of sets. [Grossman 1976] His system used complicated, multiple-component tags. One problem with his system was that ever-increasing amounts of information are represented in the structure of individual tags rather than in the network, and the structure of a tag was not so easily amenable to analysis and propagation as the network.

Fahlman, on the other hand, explicitly uses only atomic tags, and a small number of them at that. The primary use of propagation in NETL is to use highly parallel techniques to quickly perform set intersection, which is one of the more difficult data base operations. The ability to propagate markers quickly in parallel enables the use, in effect, of templates and indirect pointers to represent virtual copies of things, rather than making explicit copies; while it takes time to follow indirect pointers (one reason I avoided them in the implementation of the macro mechanism of Chapter Eight), the parallel techniques of *netl* allow many such pointers to be traced simultaneously.

10.1.5. Freuder's Method Propagates by Synthesizing Higher-Order Constraints

In [Freuder 1976] a method is described for propagating constraints by synthesizing new ones. The method is roughly as follows. Suppose that a network has n nodes. Let each constraint on k nodes ($k \leq n$) be represented as an explicit set of k-tuples representing valid combinations

of values for those nodes. Then all possible subsets of these nodes are considered, in order of cardinality. As k ranges from 2 through n, the constraints of order k (those which relate exactly k nodes) are synthesized from those of order $k-1$. More precisely, the order-k constraint on a set of nodes J is synthesized by combining the k constraints of order $k-1$ on subsets of J. Hence this constitutes a sort of compilation process using a dynamic programming technique. When the algorithm is finished, the single order-n constraint is a set of solutions for the entire network.

The running time of this algorithm is uncertain. On the one hand, a network of n nodes will require synthesis of 2^n constraints. On the other hand, as Freuder indicates but does not elucidate, the entire set of constraints of order k will contain redundant information if the original constraints were all of smaller order. He says, as an example, that in an order-4 network with originally only binary constraints, only three ternary constraints need be synthesized. He does not give a general rule, however. He also suggests some general heuristics about which constraints to synthesize first.

One may observe that Freuder's representation of a constraint amounts to a collection of "good sets", combinations of permissible values. (He explicitly assumes that each variable ranges over a finite set of values. Note that the same assumption is true of the Waltz application: the set of possible labellings for each node is large but finite.) In constrast, the system presented in this dissertation initially assumes that all combinations are possible, and then uses nogood sets to rule out invalid combinations. When the universe of discourse is finite, this does not make much difference; but if it is infinite, then the choice of one representation or the other does matter. Indeed, in my system some contortion is needed to represent the fact that a node must take on one of a finite number of values; such a fact is enforced by a constraint which, when an invalid value ever appears, records that value in a nogood set. In effect, my system is biased towards infinite "good sets", on the principle that until a node is constrained at all it may take on any value. Indeed, Freuder uses, for efficiency, a special kind of constraint, which he calls the *non-constraint*, which is in effect the set of all possibilities. He suggests also that the propagation procedure be able to deal with complements of sets. It is worth investigating the characteristics of a constraint system combining explicit nogood sets with explicit good sets. [1]

10.1.6. PROLOG Uses Chronological Backtracking on Horn Clauses

The PROLOG language allows the programmer to write statements of predicate calculus in Horn clause form. A PROLOG statement is an implication whose antecedent is the conjunction of predicates and whose consequent is a single predicate form. A typical PROLOG statement is:

```
arrange(cons(X,L),tree(T1,X,T2)) :-
        partition(L,X,L1,L2), arrange(L1,T1), arrange(L2,T2).
```

1. The `oneof` mechanism of Chapter Five constitutes an explicit representation of "good sets", but in a manner not well integrated with the representation of nogood sets.

(This is taken from a program in [Warren 1977b] that converts between lists and binary trees.) This may be interpreted declaratively as the statement

$$\forall X \forall L \forall T_1 \forall T_2 \forall L_1 \forall L_2 \ (p(L, X, L_1, L_2) \wedge a(L_1, T_1) \wedge a(L_2, T_2)) \Rightarrow a(c(X, L), t(T_1, X, T_2))$$

However, the PROLOG language also provides an imperative interpretation. The term before the ":-" is considered to be a procedure declaration, and the terms to the right are statements of the procedure. Thus, the statement above may be read, "If you need to call procedure `arrange`, then its first argument must be a cons and its second a tree, and also you must execute three other procedure calls". Moreover, there may be more than one "declaration" of a "procedure"; when a procedure must be executed, its various declarations must be chosen among "non-deterministically". (The non-determinism is implemented using chronological backtracking, as in MICRO-PLANNER [???]. If a given declaration for a procedure doesn't work, the next declaration for that procedure is tried; if all declarations fail, then failure propagates back to the caller, which must then try a new declaration for the preceding term, or fail itself. See [Sussman 1972] for a critique of the method of chronological backtracking.)

The PROLOG language, like the constraint language, provides a secondary interpretation for its semantically declarative constructs which is used to limit and guide deductive mechanisms. As we have seen in Chapters Five and Six, the non-chronological backtracking mechanism is potentially more efficient than the chronological one used by PROLOG. The PROLOG implementation keeps dependency information internally (in a specialized form made possible by the nature of its backtracking mechanism, which allows use of a stack), because after merging variables during a "procedure call" it may later have to undo the merge on failure. However, this dependency information is not available to the user. Conditionals are handled by a resolution pattern-matching mechanism and explicit predicates, both of which succeed or fail. The PROLOG compiler manages to compile these failure mechanisms out in simple cases, reducing the pattern-matching to simple dispatches.

The best implementation of PROLOG [Warren 1977a] uses data-structure techniques similar to those of the constraint system described in this dissertation. When two variables are identified, one is chosen as the "repository" for the value, and the other is made to contain an indirect pointer to the first. (It is cleverly arranged that the "oldest" becomes the repository, so that a repository cannot be destroyed during backtracking if there is any indirect pointer to it. This is one of the tricks enabled by the use of chronological, stack-based backtracking.[2])

There are many good things about PROLOG, and it deserves more popularity in this country. If there were an implementation of PROLOG which had arrays, retained general user-accessible dependency information, permitted assumptions, and forsook chronological backtracking, it might

2. For a discussion of the effects of the stack implementation technique on the development of PROLOG, as well as a good general discussion of the pros and cons of the language, see [McDermott 1980].

be close to the ideal constraint language I have in mind. Interesting variations of PROLOG allow the use of non-Horn clauses [Eder 1976], and the specification of control information to influence the backtracking order in a rather neat and intuitive way [Clark 1980?].

10.1.7. THINGLAB Provides A Class Hierarchy and Uses Pathnames

The THINGLAB system [Borning 1979] is a constraint-based graphics system that is quite similar in its capabilities to SKETCHPAD [Sutherland 1963]. Its internal organization is different, however, and indeed somewhat more flexible. It is embedded within the SMALLTALK language system (a successor to that described in [Goldberg 1976]) in much the same way that my constraint system is embedded within the Lisp Machine LISP system. The SMALLTALK language is object-oriented; all computation conceptually occurs by one object passing messages to another. This is, therefore, already very similar to a constraint system, the primary (and very large!) difference being that the computation is directional in nature. The THINGLAB system implements adirectional constraint computations, and takes advantage of the SMALLTALK class hierarchy, which allows objects to inherit properties from other objects. Borning indicates that the class hierarchy is more useful than the SKETCHPAD instance mechanism because SMALLTALK instances can have individual state variables to parameterize each instance.

The THINGLAB system always compiles a network before beginning to satisfy it; this is done for speed. Both propagation and relaxation techniques are used for constraint satisfaction. While THINGLAB initially used only the error-computation minimization technique of SKETCHPAD, in its final form it also has local procedures (analogous to the rules of my system) for explicitly satisfying constraints; these were introduced in order to deal with non-numeric constraints (which I think might better characterized as "constraints on variables over a discrete domain").

No dependency information is retained by THINGLAB. Borning states that this causes more work to be done than necessary when a parameter is changed.

Internally, while constraint networks are always compiled, parts are always accessed at run time by following path names; direct connections are not used. Borning points out that this avoids the extensive use of back-pointers (complex pointer structures were used in SKETCHPAD and also in the system described in this dissertation), at some time penalty for following the paths on every access. Another advantage of symbolic pathnames is that the description of a constraint network need not be copied every time it is instantiated. This copying is of course a problem with the macro mechanism of Chapter Eight: every macro-constraint instance requires a complete copy of the defining network. This is necessary because not only must the cells of the instance point to the devices of the network, but the devices must point back to the cells. In Borning's system much less copying is done. On the other hand, a system which shares a single read-only description among many instances is, I think, less amenable to a multiprocessing implementation.

THINGLAB provides a beautiful graphical user interface, and ways of manipulating constraints either as pictures or as SMALLTALK programs. However, it is not a true programming language, nor was it intended to be; Borning labels it a "simulation laboratory".

10.1.8. EL and ARS Analyze Electrical Circuits by Local Propagation

The EL [Sussman 1975] and ARS [Stallman 1977] systems were the direct intellectual ancestors of the research in this dissertation, and also inspired much of the other work at M.I.T. to be described in the next few sections. These were programs for electrical circuit analysis (actually, ARS was an Antecedent Reasoning System in terms of which later versions of EL were implemented). The various implementations of EL all used local propagation (one-step local deductions) of currents and voltages to analyze circuits. This is of course a natural application for constraints, as the constraint network corresponds directly to the circuit diagram, and inspires a view of constraints as active devices. While Sussman and Stallman are quick to point out that local propagation does not work for many complex synergistic circuits, they also note that the technique often produces a solution much more quickly, directly, and intuitively than the usual technique of setting up node or branch equations and then solving a large set of simultaneous linear equations. In effect, in this application the local propagation technique automatically determines the best variable to eliminate at each step from a set of equations for which the coefficient matrix is sparse.

EL/ARS also kept track of dependency information, using it both for providing explanations for the user and for the handling of contradictions by retracting only relevant premises. It was for this system that the term *dependency-directed backtracking* was coined, as well as the notions of *in* and *out* facts and *nogood sets.*

There has been continued work on the application of constraints to circuit analysis and synthesis. [de Kleer 1978b]

10.1.9. Truth Maintenance Systems Are General Dependency Managers

From ARS developed the notion that there could or should be a general programming system or package which could deal with dependencies in a general way, much as a garbage collector deals with heap storage in a general way. A series of implementations of a language called AMORD were produced at M.I.T. [Doyle 1977] [de Kleer 1978a] The AMORD system provided an indexed data base mechanism for recording symbolically represented facts, and a dependencies manager called a *truth maintenance system* (*TMS*) for recording logical relationships among the (otherwise

uninterpreted) facts.[3]

Jon Doyle [Doyle 1978a] [Doyle 1978b] [Doyle 1979] and later David McAllester [McAllester 1978] [McAllester 1980] researched and implemented methods of separating more fully the Truth Maintenance System from the data base machinery. Each deals with *nodes*, small LISP data structures which represent abstract facts. Each fact may have a truth value associated with it. One of the important results of this work was realizing (or rediscovering) and institutionalizing the distinction between knowing something not to be true and not knowing something to be true. Thus a distinction is drawn between a fact being *out* (not believed) and being *false* (believed not). Doyle's TMS implementations allow facts to be either *in* or *out*, and deductions may be made on the basis of a fact's being *in* or *out*. Negation (falseness of a fact) is handled by having two nodes, one for a fact and one for its negation, and linking them with two rules stating that if one is known to be *in*, then the other must be *out*. Thus the two nodes have four states in all, of which one (both *in*) is forbidden. The other three states correspond to the fact being known true, known false, and unknown.

The ability to make deductions based on *not* knowing something leads to a peculiar logic. Doyle and McDermott have investigated the formal properties of such a logic. [McDermott 1979] This structure allows one to express assumptions, for example: one may have a rule stating that if the negation of a fact is not known to be true, then the fact may be deduced to be true.

McAllester's version of a Truth Maintenance System [McAllester 1978] handles the three states *true*, *false*, and *unknown* directly. It also handles assumptions (which McAllester calls defaults) in a special way (specially tagging certain nodes as being automatically retractable), which inspired the methods I have used in this dissertation. His system is a little more streamlined than Doyle's, and operates somewhat differently internally. A stripped-down re-implementation of this system has been used by Shrobe in the integrated-circuit design system DAEDALUS [Shrobe 1980].

I originally set out to use a version of McAllester's TMS in this research, and re-implementing it taught me a great deal. (As it turned out, my re-implementation turned out to be surprisingly similar to Shrobe's: we had both striven to excise the remaining vestiges of the TMS' being tied to a particular data base format! I find this encouraging; it indicates that a "pure" TMS package, properly implemented, will be useful in a number of applications.)

I decided, however, that it would be more useful not to have to store a value in a cell and then make a data structure to represent the fact that the cell had the value. Rather than having *facts* with states *true*, *false*, and *unknown*, it seemed simpler just to let *cells* have states consisting of

3. One of my best failures was an attempt to write a simple LISP compiler in an early version of AMORD. The idea was that once a program was compiled an incremental change to the program would require only an incremental amount of recompilation to produce new compiled code. The use of dependency-directed backtracking would ensure that parts of the old compilation effort which did not depend on altered pieces of the program would be preserved. The first version of AMORD had a data base organized around triples, much like LEAP, in order to gain some imagined speed from a clever implementation technique. (This was my fault, I believe.) It was the attempt to write something as large as a compiler in terms of triples that proved the organization to be excessively unwieldy. Later versions of AMORD have had a richer data base structure.

their natural domain (I chose integers) plus *unknown*. The resolution techniques McAllester uses generalize in the manner I have shown; one need only think of the TMS as operating on a many-valued logic, if one wishes. McAllester's TMS represents all constraints as clauses, sets of literals not all of which may hold (or, at least one of which must not hold, depending on which side of the deMorgan coin one looks); since this is the natural form for nogood sets, such sets are represented in the same way as any other constraint, a nice feature.

Doyle comments in [Doyle 1979] that it is useful to have a way to explicitly represent the situation where both a fact and its negation are believed, and so argues against the use of three-valued Truth Maintenance Systems. To be sure (Doyle argues), this situation is contradictory and so ideally is transient, but nonetheless may persist for some time and must be dealt with. My system deals with this issue in that all facts are of the form "cell x has value n", and there is an explicit way to represent the situation where equalities are violated. If one takes a cell and its value to mean represent the "fact" that the node (in my sense, not the TMS sense of the data structure representating a fact) of the cell has that value, then the various cells of a node can represent possible facts about that node. It remains to be seen whether this structure can persist when equalities (connections) themselves are also considered to be questionable and retractable facts.

McAllester uses his TMS in [McAllester 1980], in which he views deducing facts as a process of deriving better names for an object than you started out with. I have found his ideas tangentially useful in trying to choose good names for objects when producing explanations (cf. the function `cell-goodname` in my constraint system).

10.1.10. A Simple Constraint Language Was Designed Two Years Before This

In [Steele 1979] Sussman and I described a constraint language system which was the direct precursor of the present work. The language allowed creation and connection of devices, and provided a macro abstraction facility for defining devices in terms of complex networks. It performed local propagation of values, maintained dependencies, and provided for retraction in case of contradictions. It did not have an assumption facility and the corresponding nogood machinery, and had no provision for dealing with parts of the network as algebraic expressions, though the paper contains some discussion of the possibilities.

The system in [Steele 1979] did have one interesting and useful feature that the system presented here does not (though it would not be that difficult to add): equatings could be stated not only between two cells, but between two constraints of like type. Such an equating implied recursive equating all the corresponding parts (sub-devices, pins, and other variables) of the constraint. This of course is similar in intent to the merging facility of SKETCHPAD and THINGLAB. However, those two systems did not maintain dependency information. When dependencies are maintained, some record of the merging must be kept to enter into explanations. In the system of [Steele 1979],

this was done by letting the merged objects remain distinct and passing values back and forth over an explicit equality link. A more efficient technique would be to merge the two objects into a single one, with a notation as to how to undo the process (this is done in THINGLAB). Better yet, some analogous thing might be found to do for constraint objects what I have done here for cells, letting them share a central "repository" structure while maintaining other individual components.

10.1.11. Other Work Using Constraints

There are several lines of research into the applications of constraints. None of these, I believe, is aimed directly at the construction of a general-purpose constraint language, but the ideas in them are interesting and result from the pressures of a real application of the concept. While the work I have presented in this dissertation does not draw directly from all of these, yet it has been influenced by the existence of each one, through conversations I have had with the authors or papers they have written.

Ken Forbus has used constraints to perform qualitative analyses of situations experiments in classical mechanics. He has implemented in MACLISP an efficient and extended version of the constraint language used in [Steele 1979], which was written in SCHEME [Steele 1978a].

L. Peter Deutsch, while visiting M.I.T. for half a year, worked on the theory of constraints and had many conversations with me. His primary goal (as I understand it) was to write a constraint system suitable for supporting a text processor, which must have constraints on margins, paragraph sizes, and so on.

Howard Shrobe has used constraints in an integrated-circuit design program [Shrobe 1980], and applied dependency analysis to the understanding of computer programs [Shrobe 1979].

Luc Steels has been experimenting with constraint propagation within an actors/frame organization. [Steels 1979] [Steels 1980] Rather than using a centralized general Truth Maintenance System, he lets each constraint have its own arbitrary rules for restoring consistency. He "treats a constraint as a propositional object", but by this he does not mean what I do when I say that I would like a constraint to be an object of the language. In his language, treating the constraint as an object means that its value (a truth value) indicates whether or not the constraint is in force or not. (This is similar to my suggestion in §6.3.27 that every primitive constraint have two extra pins, one as a conditional control and the other as a biconditional control on whether or not the constraint is in force.) What I mean by letting a constraint be an object is that it can *be* a value, rather than *having* a value, to which constraints can be applied. Of course, `adder` and `maxer` constraints would not make sense applied to a constraint, but an `apply` operator or a `mapcar` sort of operator would make sense.

Richard Brown has written an impressive system [Brown 1980] which synthesizes numerical programs by constructing a network of constraints, propagating numerical and symbolic informa-

tion within it, and then extracting an algebraic description of relevant portions of the network in the form of an executable LISP program. Brown's terminology is not at all the same as mine; what I call a macro-device he calls a complex device, the term macro-device having a slightly different meaning for him: it is a complex device whose definition is a subnetwork which the system extracts from a given network in order to explain the relationship between a given set of nodes. Brown notes that a complex device is created for every macro-device extracted.

I observe that Brown's macro-devices are generally used in situations where some functional relationship must be found between two (or more) nodes so that some other part of the network can act as an operator on the function which expresses the relationship; that is, as a constraint on a constraint. I would find it more natural to let constraints be objects of the language, and express his bisection-search other strategies as constraints which take other constraints as arguments. To use a mathematical analogy, Brown's system (metaphorically) takes derivatives by accepting a notation such as $d\,f(x)/dx$, looks at the two places after the d's, and figures out the functional relationship between them, and then takes the derivative of that function. By contrast, I would prefer simply to write $d\,f$ and be done with it. To use a programming-language analogy, Brown's system uses Jensen's device, while I would prefer to use functional arguments. A fair amount of Brown's system is devoted to the heuristic extraction of macro-devices; this is necessarily heuristic. It amounts to a separation of level from meta-level after they have been mixed. I would view bisection-search as a "special form" rather than a "simple constraint". (See [Steele 1978a] and [Steele 1978b], on the semantics of LISP and SCHEME, from which I draw my analogies.)

10.2. Present and Future Work

10.2.1. Tables Can Be Done "The Obvious Way" or by "Algebra"

By a "table" I mean a data structure which has individual parts into which another object can be stored; this includes arrays, record structures, cons cells, character strings (i.e., those which can be modified), and so on. I use the word "table" to avoid meaning any one of these specifically. The interesting characteristic of a table is that given a table a and a selector k (a number, say), one can access a variable a_k which the table associates with that selector. It is important that the selector can be variable, i.e., computed.

One approach to implementing these involves using the obvious sort of internal data structure, say an array or a-list, pairing selector values with associated values. One quickly concludes that what must be associated with a selector is a cell, for a table can be partially specified, with some components having known values and others not. This structure complicates the propagation

process. Suppose that a constraint `(part x a s)` is provided which enforces the relationship that the s'th component of the table a contains the value (i.e., is equated to) x. Then if `s` becomes known, cells containing the table must be awakened, because they may be connected to other `part` constraints which might be interested in the new value. This is slightly strange internally if one isn't thinking carefully; in some sense the table has not changed—it is still the same data structure—but it has become "more known" than it was before. There is a spectrum of known-ness, rather than a dichotomy of known versus unknown.

There is also a problem when two tables "collide". Consider the following sequence of statements:

```
(part 43 a 1)
(part 69 b 2)
(== a b)
```

When the first `part` constraint is created, presumably (at least, I would do it this way) the variable `a` gets as its value a table whose part named 1 is 43. Similarly `b` has as value a table whose part 2 has the value 69. When `a` and `b` are equated, we should expect the two internal table structure to be logically merged, so that `a` and `b` both have as value the self-same table which has two defined parts named 1 and 2. This is implementationally difficult to do correctly, especially so that it may be undone. (This is a problem of merging structures which may not have like parts. By the way, this is why my system has always had a function called `merge-values`; the intention was that this routine would be responsible for merging tables.)

Notice that the representation of a constraint in my system is actually very much like a table. A constraint has named parts. One experimental constraint system I have implemented deals with a problem of the systems presented in this dissertation, which is that it is not truly order-independent with respect to user input, because `create` statements for a device must precede any references to parts of that device. This would seem reasonable; but then again, why should one not be able to refer to the `a` pin of a device not yet specified, expecting to plug it in later? This is an essential prerequisite to the ability to have the analogue of functional arguments: constraints that operate on other constraints. The experimental system I refer to allowed one to make such "forward references". If one referred to `(the b foo)` and `foo` was not yet defined to be a device, then it was defined to be a `b-device`, that is, a device whose only property is that it has a `b` pin; such a device has no rules. If one then later referred to `(the a foo)` then `foo` would also be defined to be an `a-device`, which definition would then be merged with the `b-device` definition to produce a definition of `foo` as an `a-b-device`. When eventually one said `(create foo adder)`, the `adder` definition would be merged with the other, and pins of like name identified. This was all very complicated and I was unable to combine it with retraction in a straightforward way.

The other way to deal with tables is by "algebra". Rather than having any special *value* which represents a table in a cell, we let the structure of the network represent the table. Thus, if several `part` constraints all have their `a` pins connected together, than they collectively constitute the relevant structure of the table, for they relate selectors to components. All that is needed is a rule of algebra that states that if `(part x a s)` and `(part y a v)` and `s` is equal to `v` then install an equating between `x` and `y`. Such an equating must be retractable, of course, in case `s` or `v` is retracted! It also requires the ability to alter the network on the basis of the computation (which is what I mean when I say "algebra").[4]

While the algebraic method may seem more appealing, the arithmetic (or explicit data structure) method has the advantage of consolidating, in the explicit table data structure, those connections which are of interest to the table identities, distinguishing them from other equatings to table components.

10.2.2. Recursive Constraint Definitions Require Conditional Expansion

As discussed in §8.4, the present macro mechanism does not allow the definition of recursive macros, because a macro is fully expanded when it is instantiated. Much better would be a mechanism analogous to a procedure call, where a macro-constraint is not instantiated until it is "called" (i.e., until at least one of its pins is known, or perhaps one of a set of combinations of pins specified in or derived from its definition). Gerald Sussman has also suggested that one might want to have a more explicit handle on the problem by having a special form, say

`(when` *var body*`)`

meaning that the network described by *body* ought not be constructed until such time as *var* takes on the value *true*. This is not required to be retractable, however; if *var* becomes false, the constructed network remains. This seems to me a rather brute-force approach, but it does work (I have tried it in one experimental system).

10.2.3. Explanations Should Take Advantage of the Macro-Call Hierarchy

In §8.3 I briefly mentioned the possibility of producing summary explanations by glossing over the details of the contents of a macro-device. It is essential that explanations of large computations be abbreviated to be comprehensible. It is both convenient and natural to use the hierarchy of the macro-call hierarchy to guide the summarization process.

---

4. I think that there is perhaps a plateau up to which I have had trouble scaling the cliff. If any one of tables, algebra, or meta-constraints could be handled properly in combination with dependencies and retraction, then the others would come through easily. But it is a difficult feat to get any one of these.

Doyle [Doyle 1978a] [Doyle 1979] notes that summaries can be logically represented in the form of conditional proofs, from which, once they are constructed, summary explanations are easily produced. However, it is not always clear when it is useful to construct such a conditional proof. In the context of the macro-call hierarchy, however, it is natural to summarize the definition of the macro-device as a single fact, so that one can say that the computed results depend on the input values plus this single fact, which in turn is supported by the conjunction of a large number of facts (the definitions and connections which define the macro-device).

The current constraint system does not really represent the fact that a value is used by or produced by a macro-constraint; the macro-call hierarchy in effect merely provides additional names for a node in the form of pins and variables of the macro-constraint. Such pins do show up in the dependency structure of the computation, in the list of connections provided by `why-ultimately`. What is needed is a mechanism to recognize this fact and omit details of the "insides" of a macro-constraint.

10.2.4. A Constraint Language Should be Meta-Circular

I have held as a goal toward the start, that I admit I have not even closely approached but insist has been a valuable guide in definition and implementation, that a general-purpose constraint language should be powerful enough to express an interpreter for itself. (This is the analogy to a law which I have often stated in bull sessions, and believe to be original with (though probably not unique to) me: A general-purpose programming language isn't, if it can't conveniently implement itself. An example of one which isn't is BASIC.)

First, there need to be appropriate primitive constraints. This causes the constraint language to be a meta-language for itself. For example, one might need

```
(pin p c n)
```

which causes `p` to be equated to the pin named `n` of the constraint `c` (compare this with the `part` constraint suggested in §10.2.1). One might also want

```
(call c x y z ...)
```

to mean that if the value of the variable `c` is a constraint `bar` of type `foo`, then it is as if one had written

```
((foo bar) x y z ...)
```

(This is analogous to the MACLISP function `funcall`. [Moon 1974])

Once the language is sufficiently powerful, then it can be used to express its own interpreter (possibly using certain features to express themselves, as when in a LISP interpreter written in LISP one writes something like

```
(cond ((eq fun-name 'cons) (cons (car arguments) (cadr arguments)))
      ((eq fun-name 'car) (car (car arguments)))
                  ⋮
```

in the definition of the `apply` function—it is this property that Reynolds calls *meta-circularity.* [Reynolds 1972])

10.2.5. Algebra Is Operating on the Network Structure

While local propagation can take one very far, there are many situations it cannot handle. I believe that the solution is to supplement local propagation with a very limited way of augmenting the network structure under computational control to instantiate algebraic identities. These correspond to several ways of viewing the same relationship, where the various points of biew are expressed as networks with differing structure (and so they express interesting semantic relationships betwen the different networks). This is to be distinguished from the ability of the current system to take several points of view concerning the same network structure; this is a more syntactic kind of algebra that can be performed without reference to the semantics of the constraints, but only using the topology of their connections. Ways of introducing multiple points of view are discussed in [Sussman 1977] and [Steele 1979]. Richard Brown's system [Brown 1980] in fact implements such algebraic augmentation of the network. One can imagine facilities for pattern-directed invocation of algebra rules that would trigger on specified network configurations when new connections were made.

10.2.6. The System May Need Control Advice from the User

Reality being what it is, sometimes the system's automatic control structures will thrash wildly without some advice from the user on in what order to perform computations. The priority queue structure of Chapter Six, for example, provides some automatic control heuristics, but the user may need to fine-tune these.

An approach I find intriguing might be borrowed from the IC-PROLOG system. [Clark 1980?] This version of PROLOG allows one to annotate the argument forms to "procedure calls" to indicate that a particular argument is a "lazy producer" of values or an "eager consumer" of them. To translate this concept to the constraint system: a pin of a constraint-type could be labelled by a ? (this pin is an eager consumer) or a ! (this pin is a lazy producer). Then rules with triggers labelled ? would have very high priority when such a trigger received a value; and rules with output pins labelled ! would have lowered priority. Moreover, if another constraint's rule has an output pin connected to an eager consumer cell, then that rule has high priority (and this overrides any !

annotation of that output pin). Using this annotation scheme one can express co-routines and such using recursively defined constraints.

Another way to control the order of computation is to let every constraint have an extra conditional control pin, as suggested in §6.3.27. Then a network of meta-constraints could selectively enable and disable constraints via this pin; a disabled constraint would not propagate, and enabling would allow rules to be awakened. This technique in particular might be of use in combination with algebraic techniques. Very often, in a redundantly specified network, one can determine that several subnetworks are duplicating each other's efforts. This could be detected by pattern-directed methods, and an algebra rule triggered that would disable the redundant versions of the network, or perhaps just give them low priority.

10.2.7. Techniques Are Needed for Run-time Storage Reclamation

One property of the recording of dependency information is that the entire history of the computation is maintained. This is an advantage, but it also poses a problem, in that it takes memory to hold the history. In a practical constraint system there will need to be ways to reclaim data structures which are unused or likely not to be used. The best candidates for reclamation are those structures which can be recomputed if necessary.

In a constraint system with an abstraction hierarchy, one could reclaim the dependency structures, and even the networks, for the bodies of macro-constraints, leaving behind the results that had been computed from the inputs. (If later the dependency structures are traced, the syustem can report: "I computed it using this macro-constraint, whose instance here I garbage-collected, but I assure you that it was a valid computation—and if you like I will reconstruct the proof.") If one is interested in reclaiming devices, a good choice might be devices whose pins all have values, for in some sense they have done all the work they can. (The appearance and disappearance of devices is analogous to the appearance and disappearance of incarnations of a procedure; the incarnation appears (perhaps in the guise of a frame on the run-time stack) when the procedure is needed, may survive for a while in a co-routining context, and then disappears when it is done.)

Another possibility is reclamation of nogood sets. There are at least two cases of interest. One is that resolution may produce a new nogood set that supersedes an old nogood set for some node, in that the set of pairs in the new nogood set is a proper subset of that of the old nogood set. In this case the old one can be reclaimed. Doing this efficiently would require a more complex indexing structure for nogood sets than I have used here, but would probably be worth a great deal. It would certainly decrease the time spent checking nogood sets when solving something like the N queens problem as in §6.4.

The other possibility for reclaiming nogood sets is necessarily heuristic. One would simply throw away nogood sets containing values which are (for some reason) considered to be unlikely

to recur. Such nogood sets can always be recomputed if necessary. Perhaps with each nogood set might be recorded some measure of the effort that was necessary to compute it, and cheap ones might be discarded before expensive ones. Care is needed, of course, to prevent thrashing caused by constantly reclaiming and recreating a nogood set.

10.3. Contributions of This Research

These are what I believe to be the new and original contributions of the research reported by this dissertation:

- I attempted to design a complete, general-purpose programming system organized around constraints. While this goal has not yet been met, yet I have made some progress toward it.
- The structure of the implementation matches the imagery of the paradigm. Data structures which are thought of as being directly connected (as for example in a diagram of a network) actually *are* connected.

  (If this point seems trivial, consider two implementations of LISP, one using the usual pointer representation, and one operating on S-expressions represented in "external form", as character strings. An implementation of the latter form may seem manifestly laughable, but consider: (1) McCarthy mentioned the possibility of such an implementation inone of the first early papers on LISP. [McCarthy 1960] (2) Church's lambda calculus [Church 1941], an intellectual precursor to LISP, was defined in terms of manipulations on strings, not trees, of symbols. The tree representation had to await computer imnplementation. (3) Certain program editors for LISP, notably the cousins EMACS [Stallman 1980] (the editor usually used by the MACLISP community [Moon 1974]) and ZWEI (the editor for Lisp Machine LISP [Weinreb 1979]), represent those programs as character strings and yet manipulate S-expressions, as if performing `car`, `cdr`, and `cons`operations, by manipulating these strings. For some purposes, such as pretty-printing, the character string representation is actually *more* convenient than the pointer representation. (Indeed, that is why we write LISP programs as character strings rather than box-and-arrow diagrams!) Therefore the string representation is not *obviously* impractical for all purposes. On the other hand, the pointer representation is certainly much more economical for *most* purposes, and this follows our intuitions about explicitly representing interesting relationships (such as car and cdr) directly. The points are that (a) alternative representations need to be explored, and (b) a representation which conforms to a pictorial image may seem more cumbersome at first but may be more efficient because interesting relationships are represented directly.)
- The structure of the implementation is closer to being suitable for implementation on multiple processors than any other constraint system. (The organization of THINGLAB [Borning] might on the surface appear to be equally suitable, but the internal use of pathnames rather than direct

pointers causes problems.) Considerable effort has been made to make small the quanta of necessarily indivisible computation.

- The research was conducted with the principles of *order-independence*, *locality*, and *monotonicity* always explicitly in mind. These principles are essential to the design of a constraint system. Order-independence prevents the system from relying on the form of the input; its computations should be derived from the content only. Locality and monotonicity are important to the conceptual simplicity and comprehensibility of the system.
- This system deals explicitly with the problem of behaving properly in the presence of contradictions. Just as Doyle and McAllester had to make the distinction between the truth of a fact and the belief in a fact, so I have noted the distinction between the *consistency* of a system and the *well-foundedness* of the system. Every computation, every deduction made by the system presented here is well-founded: each result is validly deduced from given premises. If the premises are inconsistent, then the conclusions may be contradictory, but they will nevertheless have correct justifications.

  It is important for a constraint system to be tolerant of contradictions; if a problem arises in a large system of relationships, the user may not want to deal with the problem immediately. He may want to investigate the problem, or work on distantly related parts of the network that may be affected by the contradiction only slightly or not at all. Another case, common when revising designs for engineered artifacts, is that the user wants to change several parameters at once; changing any one would produce a hopeless snarl of contradictions, but if the user can only tell the system, "Wait," then he can make the other changes to make the system consistent again.

  Previous Truth Maintenance Systems have been relatively intolerant of contradictions, insisting that each one be resolved immediately. The system I have presented is tolerant. The state of a network containing contradictions is well-defined, because it is well-founded, and the computation and explanation mechanisms can still behave reasonably and intuitively. Heuristics (lowered priority for computing consequences of values known to be in conflict) prevent the wasting of computational effort on deductions likely to be retracted soon.

To God alone be the glory. Amen.

The Party of the first Part
And the party of the next
Are partly participled
In a parsley-covered text.

Were you partial to a Party
That has parceled out its parts
With the Party that was second
In your polly-tickle heart?

Then parlay all your losings
On a horse that's running dark—
With lights-out you may triple
In a homer in the park.

—Walt Kelly (1952)
I Go Pogo

References

[Ackerman 1979]

Ackermann, William B., and Dennis, Jack B. VAL: *A Value-Oriented Algorithmic Language (Preliminary Reference Manual).* MIT/LCS/TR-218. M.I.T. Laboratory for Computer Science (Cambridge, June 1979).

[Arvind 1978]

Arvind; Gostelow, Kim P.; and Plouffe, Wil. *An Asynchronous Programming Language and Computing Machine.* Department of Information and Computer Science, University of California (Irvine, December 1978).

[Bobrow 1980]

Bobrow, Daniel G., and Winograd, Terry. Response to Knowledge Representation Questionnaire. *ACM SIGART Newsletter* 70 (February 1980), 85.

[Borning 1979]

Borning, Alan. THINGLAB: *A Constraint-Oriented Simulation Laboratory.* SSL-79-3. Xerox Palo Alto Research Center (Palo Alto, California, July 1979).

[Brown 1980]

Brown, Richard Henry. *Coherent Behavior from Incoherent Knowledge Sources in the Automatic Synthesis of Numerical Computer Programs.* Ph.D. Dissertation, M.I.T. (Cambridge, June 1980).

[Church 1941]

Church, Alonzo. *The Calculi of Lambda Conversion.* Annals of Mathematics Studies Number 6. Princeton University Press (Princeton, 1941). Reprinted by Klaus Reprint Corp. (New York, 1965).

[Clark 1980?]

Clark, K.L., and McCabe, F.G. *The Control Facilities of* IC-PROLOG. Department of Computing and Control, Imperial College (London, undated, circa 1980?).

[de Kleer 1978a]

de Kleer, Johan; Doyle, Jon; Rich, Charles; Steele, Guy L. Jr.; and Sussman, Gerald Jay. AMORD: *A Deductive Procedure System.* AI Memo 435. M.I.T. Artificial Intelligence Laboratory (Cambridge, January 1978).

[de Kleer 1978b]

de Kleer, Johan, and Sussman, Gerald Jay. *Propagation of Constraints Applied to Circuit Synthesis.* AI Memo 485. M.I.T. Artificial Intelligence Laboratory (Cambridge, September 1978). Also in *Circuit Theory and Applications* **8** (1980), 127–144.

[Dennis 1973]

Dennis, J.B. *First Version of a Data Flow Procedure Language.* Computation Structures Group Memo 93. M.I.T. Laboratory for Computer Science (Cambridge, November 1973). Revised as M.I.T. Project MAC TM-61 (May 1975).

[Dennis 1975]

Dennis, J.B., and Misunas, D.P. "A Preliminary Architecture for a Basic Data-Flow Processor." *Proc. Second Annual Symposium on Computer Architecture* (January 1975), 126–132.

[Doyle 1977]

Doyle, Jon, de Kleer, Johan, Sussman, Gerald Jay, and Steele, Guy L. Jr. "AMORD: Explicit Control of Reasoning." *Proc. AI and Programming Languages Conference* (Rochester, New York, August 1977). *ACM SIGPLAN Notices* **12**, 8, *ACM SIGART Newsletter* **64** (August 1977), 116–125.

[Doyle 1978a]

Doyle, Jon. *Truth Maintenance Systems for Problem Solving.* S.M. Dissertation, M.I.T. (Cambridge, May 1977). AI-TR-419. M.I.T. Artificial Intelligence Laboratory (Cambridge, January 1978).

[Doyle 1978b]

Doyle, Jon. *A Glimpse of Truth Maintenance.* AI Memo 461a. M.I.T. Artificial Intelligence Laboratory (Cambridge, November 1978).

[Doyle 1979]

Doyle, Jon. *A Truth Maintenance System.* AI Memo 521. M.I.T. Artificial Intelligence Laboratory (Cambridge, June 1978). Also in *Artificial Intelligence* **12** (1979), 231–272.

[Eder 1976]

Eder, Gottfried. *A* PROLOG*-like Interpreter for Non-Horn Clauses.* D.A.I. Research Report

26. Department of Artificial Intelligence, University of Edinburgh (Edinburgh, September 1976).

[Fahlman 1977]

Fahlman, Scott E. NETL: *A System for Representing and Using Real-world Knowledge.* Ph.D. Dissertation. M.I.T. (Cambridge, September 1977). Also published by M.I.T. Press (Cambridge, 1979).

[Fateman 1973]

Fateman, Richard J. "Reply to an Editorial." *ACM SIGSAM Bulletin* **25** (March 1973), 9–11.

[Floyd 1979]

Floyd, Robert W. "The Paradigms of Programming." 1978 ACM Turing Award Lecture. *Comm. ACM* **22**, 8 (August 1979), 455–460.

[Freuder 1976]

Freuder, Eugene C. *Synthesizing Constraint Expressions.* AI Memo 370. M.I.T. Artificial Intelligence Laboratory (Cambridge, July 1976).

[Goldberg 1976]

Goldberg, Adele, and Kay, Alan. SMALLTALK-*72 Instruction Manual.* Learning Research Group, Xerox Palo Alto Research Center (March 1976).

[Gries 1977]

Gries, David. "An Exercise in Proving Parallel Programs Correct." *Comm. ACM* **20**, 12 (December 1977), 921–930.

[Grossman 1976]

Grossman, Richard W. *Some Data Base Applications of Constraint Expressions.* S.M. Dissertation. M.I.T. (Cambridge, January 1976). Also TR-158. M.I.T. Laboratory for Computer Science (Cambridge, February 1976).

[Knuth 1973]

Knuth, Donald E. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley (Reading, Massachusetts, 1973).

[Kowalski 1974]

Kowalski, Robert. "Predicate Logic as Programming Language." Information Processing 74. North-Holland (1974).

[Kowalski 1979]

Kowalski, Robert. "Algorithm = Logic + Control." *Comm. ACM* **22**, 7 (July 1979), 424–436.

[Kowalski 1980]

Kowalski, Robert. Response to Knowledge Representation Questionnaire. *ACM SIGART Newsletter* 70 (February 1980), 44.

[McAllester 1978]

McAllester, David A. *A Three Valued Truth Maintenance System.* AI Memo 473. M.I.T. Artificial Intelligence Laboratory (Cambridge, May 1978).

[McAllester 1980]

McAllester, David A. *The Use of Equality in Deduction and Knowledge Representation.* S.M. Dissertation, M.I.T. AI-TR-550. M.I.T. Artificial Intelligence Laboratory (Cambridge, January 1980).

[McCarthy 1960]

McCarthy, John. "Recursive Functions of Symbolic Expressions and Their Computation by Machine — I." *Comm. ACM* 3, 4 (April 1960), 184–195.

[McDermott 1979]

McDermott, Drew V., and Doyle, Jon. *Non-Monotonic Logic — I.* AI Memo 468b. M.I.T. Artificial Intelligence Laboratory (Cambridge, August 1978, revised July 1979). Also in *Artificial Intelligence* 13 (1980), 41–72.

[McDermott 1980]

McDermott, Drew. "The PROLOG Phenomenon." *ACM SIGART Newsletter* 72 (July 1980), 16–20.

[Moon 1974]

Moon, David A. *MacLISP Reference Manual (Revision 0).* Project MAC, M.I.T. (Cambridge, April 1974).

[Owicki 1975]

Owicki, Susan Speer. *Axiomatic Proof Techniques for Parallel Programs.* Ph.D. Dissertation. TR 75-251. Department of Computer Science, Cornell University (Ithaca, New York, July 1975).

[Pratt 1977]

Pratt, Vaughan R. "The Competence/Performance Dichotomy in Programming." *Proc. Fourth ACM Symposium on Principles of Programming Languages (POPL)* (Los Angeles, January 1977), 194–200.

[Quillian 1968]

Quillian, M. Ross. "Semantic Memory." In Minsky, Marvin (ed.), *Semantic Information Processing.* M.I.T. Press (Cambridge, 1968).

[Reynolds 1972]

Reynolds, John C. "Definitional Interpreters for Higher Order Programming Languages." *Proc. ACM National Conference* (Boston, 1972), 717–740.

[Shrobe 1979]

Shrobe, Howard Elliot. *Dependency Directed Reasoning for Complex Program Understanding.* Ph.D. Dissertation, M.I.T. AI-TR-503. M.I.T. Artificial Intelligence Laboratory (April 1979).

[Shrobe 1980]

Shrobe, Howard. "Constraint Propagation in VLSI Design: DAEDALUS and Beyond." Abstracts from Spring 1980 M.I.T. VLSI Research Review. (Cambridge, May 1980).

[Sloman 1980]

Sloman, Aaron. Response to Knowledge Representation Questionnaire. *ACM SIGART Newsletter* 70 (February 1980), 86.

[Stallman 1977]

Stallman, Richard M., and Sussman, Gerald Jay. *Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis.* AI Memo 380. M.I.T. Artificial Intelligence Laboratory (Cambridge, September 1976). Also in *Artificial Intelligence* 9 (1977), 135–196.

[Stallman 1980]

Stallman, Richard M. EMACS *Manual for* ITS *Users.* AI Memo 554. M.I.T. Artificial Intelligence Laboratory (Cambridge, June 1980).

[Steele 1977]

Steele, Guy Lewis Jr. "Fast Arithmetic in MacLISP." Proceedings of the 1977 MACSYMA Users' Conference. NASA Sci. and Tech. Info. Office (Washington, D.C., July 1977), 215–224.

[Steele 1978a]

Steele, Guy Lewis Jr., and Sussman, Gerald Jay. *The Revised Report on* SCHEME: *A Dialect of* LISP. AI Memo 452. M.I.T. Artificial Intelligence Laboratory (Cambridge, January 1978).

[Steele 1978b]

Steele, Guy Lewis Jr., and Sussman, Gerald Jay. *The Art of the Interpreter; or, The Modularity Complex (Parts Zero, One, and Two).* AI Memo 453. M.I.T. Artificial Intelligence Laboratory (Cambridge, May 1978).

[Steele 1979]

Steele, Guy Lewis Jr., and Sussman, Gerald Jay. "Constraints." AI Memo 502. M.I.T. Artificial Intelligence Laboratory (Cambridge, November 1978). Invited paper. *Proceedings APL '79. ACM SIGPLAN STAPL APL Quote Quad* 9, 4 (June 1979), 208–225.

[Steels 1979]

Steels, Luc. *The* XPRT *Description System.* AI Working Paper 178. M.I.T. Artificial Intelligence Laboratory (Cambridge, January 1979).

[Steels 1980]

Steels, Luc. *The Constraint Machine* (draft). Schlumberger-Doll Research Lab (Ridgefield, Connecticut, May 1980).

[Sussman 1971]

Sussman, Gerald Jay; Winograd, Terry; and Charniak, Eugene. MICRO-PLANNER *Reference Manual.* AI Memo 203A. M.I.T. Artificial Intelligence Laboratory (Cambridge, December 1971).

[Sussman 1972]

Sussman, Gerald Jay, and McDermott, Drew Vincent. "Why Conniving is Better than Planning." AI Memo 255A. M.I.T. Artificial Intelligence Laboratory (Cambridge, April 1972). Also appeared as "From PLANNER to CONNIVER—A Genetic Approach." Proc. 1972 Fall Joint Computer Conference. AFIPS Press (Montvale, New Jersey, 1972), 1171–1179.

[Sussman 1975]

Sussman, Gerald Jay, and Stallman, Richard M. *Heuristic Techniques in Computer-Aided Circuit Analysis.* AI Memo 328. M.I.T. Artificial Intelligence Laboratory (Cambridge, March 1975). Also in *IEEE Transactions on Circuits and Systems* Vol. CAS-22 (11) (November 1975).

[Sussman 1977]

Sussman, Gerald Jay. SLICES: *At the Boundary between Analysis and Synthesis.* AI Memo 433. M.I.T. Artificial Intelligence Laboratory (Cambridge, July 1977).

[Sutherland 1963]

Sutherland, Ivan E. SKETCHPAD: *A Man-Machine Graphical Communication System.* M.I.T. Lincoln Laboratory Technical Report 296 (January 1963).

[Waltz 1972]

Waltz, David L. *Generating Semantic Descriptions from Drawings of Scenes with Shadows.* AI TR-271. M.I.T. Artificial Intelligence Laboratory (Cambridge, November 1972).

[Warren 1977a]

Warren, David H.D. *Implementing* PROLOG: *Compiling Predicate Logic Programs.* Two volumes. D.A.I. Research Reports 39 and 40. Department of Artificial Intelligence, University of Edinburgh (Edinburgh, May 1977).

[Warren 1977b]

Warren, David H.D., and Pereira, Luis. "PROLOG: The Language and Its Implementation Compared with LISP." *Proc. Symposium on Artifical Intelligence and Programming Languages*

(Rochester, New York, August 1977). *ACM SIGPLAN Notices* **12**, 8, *ACM SIGART Newsletter* **64** (August 1977), 109–115.

[Weinreb 1979]

Weinreb, Daniel, and Moon, David. *Lisp Machine Manual (Second Preliminary Version).* M.I.T. Artificial Intelligence Laboratory (Cambridge, January 1979).

[Winston 1974]

Winston, Patrick Henry. *New Progress in Artificial Intelligence.* AI TR-310. M.I.T. Artificial Intelligence Laboratory (Cambridge, June 1974).

[Winston 1977]

Winston, Patrick Henry. *Artificial Intelligence.* Addison-Wesley (Reading, Massachusetts, 1977).