A Constraint Language in C

by

William D. Cattey

Submitted in Partial Fulfillment of the requirements

for the Degree of

Bachelor of Science

at the

Massachusetts Institute of Technology

May, 1983

Signature of Author _____
Department of Electrical Engineering and Computer Science
May 6, 1983

Certified by _____
Gerald Jay Sussman
Thesis Supervisor

Accepted by _____
David Adler
Chairman, Departmental Committee on Theses

A Constraint Language in C

William D. Cattey

Massachusetts Institute of Technology

Gerald Jay Sussman

Thesis Advisor

Abstract

This thesis asserts that a constraint language can be written in the C programming language. It presents the data structures and low level procedures that would serve as the basis of a stand alone constraint language or logic simulator. Constraint systems are defined in intuitive terms. Advantages of modeling in the environment of a constraint language are described. Some history of other systems is presented. The description of the implementation covers such issues as efficient data organization, real time storage allocation, constraint systems as subprograms, and constraint systems as stand alone programming environments.

Contents

Appendicies

Figures

Introduction

To properly introduce this paper, three questions must be addressed: what a constraint language is, what local propagation is, and what a logic simulator is. After defining these terms, I will address the larger issues of why constraint languages and logic simulators are interesting to study, why I chose C as the implementation language, and what implementation tradeoffs I made. For the sake of completeness, I mention some advanced topics necessary for a truly beautiful stand alone system.

Originally I conceived of my implementation as a stand alone constraint language. Instead of producing a marginally useful stand alone language, and a poor discussion, I later elected to focus my effort on the discussion, and production of working code that would be the *basis* of a stand alone constraint language in C. Since a stand alone system is the most general way to apply the primitive data structures and procedures, it proved useful to design everything with a stand alone system in mind. The code documented in this paper represents working subsystems that can be gathered together and beautified into a stand alone system.

What is a constraint system?

It is vitally important to realize that nearly *anything* can be thought of as a constraint system. In the most general sense, a constraint system describes the ways things in a group are inter-related. If one or more of the things is an unknown quantity, the list of possible values is shortened by keeping in mind that the system must remain consistent for all correct values of the unknown.

In computer modeling, the purpose of a constraint system is to express some relationship, and then to *constrain* certain portions of the system until there is enough information to fill in some important unknown. Stated simply: You give it information a little bit at a time until it gives you the answer based on what can be inferred from the relationship. A constraint system consists of two distinct parts: the various quantities that have values, and the relationship those quantities have to each other.

Before going any further I would like to give the reader a few examples that will capture certain essential aspects of what a constraint system is in an intuitive way.

A set of simultaneous linear equations defines a type of constraint system. In both the equation set and the constraint system, there are variables which get values by filling in the knowns until the unknowns are filled consistently.

An electrical network is a constraint system. There is a method for solving electrical networks called Propagation of Constraints [de Kleer 78]. More on this bit of network theory is discussed in the section on History. Much of the nomenclature I have used throughout this paper is based on thinking about constraint systems as if they were physical circuits wired together.

The ubiquitous "spread sheet" programs for personal computers are constraint languages. Their problem solving domain is limited to mathematical expressions relating items in different "locations". The implementation described in this paper is designed to permit modeling in arbitrary domains.

Although abstract mathematics provides the most accurate understanding of what a constraint system is, people who are facile with abstract mathematics could easily learn and use a computer language that implemented models of constraint systems. It is my goal to provide a more intuitive and less mathematical description aimed at introducing modeling with a constraint language to those who would not ordinarily think of using it. I choose metaphor rather than mathematics as my prime method of presentation.

I would now like to introduce my nomenclature: For the purposes of this document, I will describe the process of modeling a constraint system as a way of wiring together abstract devices I call *units*. The system is the network of wired together units. Values are stored in objects I call *cells*. More nomenclature will be presented in the section on implementation. The terms unit, cell and wire are crucial to further discussion.

Now that I have defined a constraint language and a bit of intuitive nomenclature for it, I can capture the idea of local propagation simply. Values in a constraint system propagate along the wires between cells and units that are wired directly to each other. The propagation is local because the wires restrict the flow of data to be between adjacent objects. With local propagation there is no way to make the change of one value to affect all units in the network in a simultaneous or global way. Values pass locally over wires between units and cells or between units and units.

There are some problems with local propagation, most of them have to do with networks containing loops. If there is a loop in the network, the value goes around the loop to its starting point without noticing that it has been there before. Loops in a network can have one of three effects: Perhaps having a loop in the network has no effect at all because the propagation of values never traverses the loop. The presence of a loop may prevent cells from settling down to consistent values, if a unit produces an answer that radically changes its input. A loop may prevent some cells from ever getting values because the dependencies are such that no unit in the subnet get enough data to produce a value to propagate through the rest of the network.

The only way to prevent the effects of loops is to eliminate them. The elimination of loops turns out to be a hairy algebra problem. I believe that there are rigorous treatises on loop elimination in the libraries of abstract mathematicians. It is with considerable hand waving that I assert that the simplest way to eliminate loops is for an algebraic canonicalizer to help the definer of the network create loop free networks (called acyclic graphs in modern algebra). The proof that there is an acyclic graph that would solve every network that one would want to wire is well beyond the scope of this paper. It is clear, however that there is much interesting ground to explore even with loops and with those problems that can be expressed directly without inordinate amounts of algebra.

From this it is also possible to conclude that there are some problems that are silly to ex-

press with local propagation. Here, I let pragmatism rule: If it seems to hard to think of a network that expresses the problem, I should solve it with a method different from propagation of constraints. If the problem is already a network, I have already solved the topology problem and defining the abstract units is a one time problem for each type of function I want to model. I will say more on the advantages of modeling with a constraint language in the section on motivation.

What is a logic simulator?

A logic simulator is a refinement on a constraint system. Like a constraint system, there are abstract units wired together with values that get locally propagated among value cells and units. The domain is restricted to that of boolean logic. Theoretically one could simulate an entire computer system with a logic simulator. In practice it would take many computer cycles to run such a description.

Unlike a general constraint language, propagation delay of data between units is carefully monitored. A general constraint language is not supposed to care too much when values get to various places as long as the network is globally stable and gives consistent answers. A logic simulator contains a priority queue so that propagation delay can be simulated. The priority queue is filled to permit some values to propagate sooner or later than usual. Ordinarily the implementation would queue up propagations to most safely guarantee that values arrive at their expected destination.

Advanced Topics

Now that the three basic questions have been answered, I would like to touch briefly on some advanced topics. These are mentioned for the sake of presenting a picture of a complete stand alone constraint language system. As was mentioned earlier, the code documented in this paper is the basis for a stand alone system. To make it all into a stand alone system, some top level evaluator must be provided to enable a programmer to dynamically wire up networks.

The top level can get arbitrarily complex. Even in my original implementation strategy, I only planned five complementary pairs of top level commands:

Create & Destroy -- For new instances of old data types.
Connect & Disconnect -- For wiring.
Define & Expunge -- For packaging networks into new data types.
Set & Clear -- For assigning values to cells.
Value & Follow -- For looking at values and tracing propagations.

The ten commands represent the minimum necessary to wire up arbitrary networks, to manipulate values, and to package subnetworks into new abstract data types.

My only attempt at "user-friendliness" is making the command set simple and limited. There is no attempt at refined tracing, or complicated alteration to the form that the user types. Simplicity is the rule. Any network that the user chooses to wire is dutifully wired. Any mistake is complained about immediately with no attempt by the system to guess the user's intentions. Instead, I go out of my way to prevent the system from throwing away anything it does not understand. For example I choose to implement connect and disconnect and permit incremental change of a network rather than requiring re-entering of the whole thing. Such issues, although part of the advanced topic of building a stand alone language system, are basic to a reasonable implementation.

In the previous paragraphs I glossed over the details of the

Define and Expunge commands. They represent an extremely advanced feature to include in a stand alone constraint language. I believe that usable constraint systems will need to have a mechanism to permit sections of an existing network to be gathered into a more efficient structure that could be accessed as if it were a single unit. As I will discuss in more detail in the section on motivation, the abstracting units from the network is an important way of shrinking portions of the modeling problem.

Finally a canonicalizing front end would be an effective method of eliminating loops. It might permit complicated pieces of network to be wired on a simple macro command. A canonicalizer and algebraic manipulator could be either the whole front end or a filter that sends strings of commands to a simple front end. A canonicalizer is too difficult to implement as front end for the first version stand alone system. However in a fit of modularity and abstraction it occurs to me that a canonicalizer can later be written that issues commands to the simple front end I did implement.

Motivation

Why implement a constraint language? Why do it in C? These two questions are central to the motivation behind my thesis. C is Turing equivalent to all other computer languages. That means that it has already been proven that if an implementation exists on one programming language, one is possible in C. Since Steele and Sussman wrote several constraint languages in LISP, it is already proven that one *could* be written in C. However, the choice of C as an implementation language represents a collection of assertions about efficiency and desirability of C for this problem domain. This section will unify the ideas of the desirability of programming in a constraint language with the ideas of desirability of C as an implementation environment.

In the previous section, I defined a constraint system in intuitive terms. I also hinted at what a stand alone constraint language might have for some of its properties. I have not yet given a clear idea of what a constraint language for computer modeling might be. Nor have I clearly shown why such a language would be useful. First I would like to describe what I see when I think of an ideal stand alone constraint language.

Imagine a computer language in which, instead of expressing a program to perform a function as a sequence of linear steps or even as a recursion of subroutine calls, one defines a network of functional units wired together which models the behavior of some system. At some level, there might be lines of code executed sequentially or recursively, but the essence of my idealized picture of a constraint language is that the user wires up something that seems very much like the system being modeled. It would be inappropriate to call the person doing the modeling with the aid of the computer a programmer, because I would assert that program implies sequences of steps, a program. A constraint network captures the topology of a problem in a clean and vivid way. Topology is not clearly expressed in an ordinary computer program.

If one modeled amplifiers, there would be units called transistors with three pins that wire to resistors which have two pins, etc. To model an amplifier, one wires the units up as if one had soldered physical devices. The user interface, and the philosophy of modeling in this

way are central to my notion of a constraint language. Although I describe this as a user interface, we no longer consider a C compiler to be "the user interface" for the underlying machine language. It is my goal that the use of constraint languages will eventually reach this same level of acceptance.

In an analogous fashion to the way high level languages capture more than machine languages, constraint languages capture more than high level programming languages. Machine languages express operations, sequences and sometimes recursions. High level languages simplify the expression process by giving more abstract ways for programmers to express programs.

Modern software engineering is big on the ideas of modularity and abstraction. Making a program more abstract means making it seem more like the physical thing it models. Currently high level languages make programs more abstract by creating subprograms called modules that capture some abstract aspect in an elegant way. Ideally, constraint languages should increase abstraction through modular units to capture even more elegantly the properties of portions the thing being modeled. Additionally, they should do one thing that high level languages do not do: They should capture, in an elegant way, some of the topology of the thing being modeled.

To provide all the features of an idealized constraint language many things are necessary:

1) Bottom level data structures to support the foundation.
2) Primitive procedures to create and modify the data structures.
3) Higher level procedures to manipulate one and two above in an abstract way.
4) Top level procedures to create and wire abstract data types.
5) Top level procedures to set clear and view values.
6) Top level procedures to compile network into new abstract data types.
7) Procedures above top level to convert user input into forms most likely to get the desired answer.

As I mentioned in the introduction, the system as I originally conceived it had items one through

six, and that item seven, taking the form of a canonicalizer that could be added later.

For the purpose of this thesis only one two and three have been fully implemented. Four five and six require the creation of a parser and a symbol table manipulator, and I did write such code, but debugging, documenting and describing it would have jeopardized the ability of this paper to adequately present items one two and three. I chose to spend time writing such code because I wanted to taste as many implementation issues as I could. I wanted to be sure that the low level code I wrote could eventually be fully expanded into my ideal of a constraint language. As the appendix containing my implementation notes shows, the low level routines evolved a great deal as I more fully understood the farthest reaching issues.

Now that I have presented my idealized view of a constraint language, its usefulness, and its necessary parts, reasons for my chosing to study constraint language implementation should be clear: There is the potential to express computer models with a clarity unavailable through existing high level languages. The combination of modularity and topology aid abstraction to the point where, for many problem domains, the model looks amazingly similar to the physical object being modeled. The goal of this work is to bring the concepts of modeling with constraint languages to a wide variety of new people.

Now to the question of why I chose C as an implementation language. There are three central reasons behind my choice of C. It is widely available, it is philosophically correct, and it will permit a good implementation. The three phrases I chose to summarize the three ideas are meant to capture a lot of meaning in a little space. I will clarify them.

The first idea behind my choice of C was that it was a language available on a large number of machines. Many new computer facilities come up every day with C as one of their languages. The trade press overflows with articles on how UNIX systems (UNIX is a trademark of Bell Laboratories), which are implemented in C are taking over the world. While software houses struggle to implement Ada compilers for the Department of Defense, C is available now and is rapidly gaining acceptance. There is a high probability that, by the time Ada becomes

available, C will be too well established to permit wide acceptance of Ada.

In addition to gaining popularity, C fits onto a wide variety of machines. I have C running on my Z-80 microcomputer at home. There is a UNIX implemented in C for the IBM 370 and Amdahl 470 mainframes. The two major operating systems available for the Digital Equipment Corporation VAX line of computers, VMS and UNIX, both run C. Digital provides C with VMS to compete more successfully with UNIX. Even the ubiquitous Apple II has several C systems available for it. C is the language of choice when one wishes to specify a language that is available on the largest number of machines. The only language more widely accepted than C is BASIC, and the choice of C over BASIC is an implementation issue and a philosophical issue.

In addition to being available on many machines, C is portable. With care, I can write a program that will run, un-modified on more different machines than even the so called "standard" ANSI FORTRAN IV. Historically, FORTRAN was a language that people wished to make portable by establishing standards on paper that people would obey. C was made portable by developing a language in which the writing of portable programs was a natural easy thing to do. The difference is so striking that one can say that to write portable FORTRAN code one has to have the manual open at all times while the C code does the right thing by default in all but a few clearly defined cases. The portability issues were not defined by a standards committee, they were explored by porting the UNIX operating system to several machines of differing architectures.

The second idea has to do with the philosophy of writing and maintaining programs. Each programmer has a philosophy of what things to do to write the best programs. Mine is that programs should consist of small procedures grouped logically into modules which are gathered together to make a whole system. Although there are some who can write a whole program without breaking it up into procedures, I find that I can only remember a relatively small amount of information at a time. Therefore my programs tend to be very abstract, and very small and simple at any level. This is exactly the philosophy of programming in C. So totally is C commit-

ted to this philosophy that the I/O library is not part of the language, it is a separate module designed to be tailored to the needs of specific users while continuing to make portability easy.

As an example, Clu is an extremely modular and abstract language which I find a joy to use. Yet it does not have the same commitment to simple elegance that C does. Clu attempts to be all things to all people as a large, all-encompassing, language. C remains a simple language from which one can take what one needs and no more. Think of the difficulties one would face while modifying the I/O library of Clu to do special TV terminal functions more efficiently than currently possible. The ability to be abstract and simple at the same time captures the essence of my philosophy of programming. As a side effect, my programs are easier to maintain, and easier to understand by people who have never seen them before.

The UNIX environment encourages simple pragmatic generality. Efficiency comes from simplicity. The most prominent feature of the UNIX environment is that it does not "get in the way" while one programs. This sort of perspective was particularly helpful in my goal of bringing constraint modeling out of the laboratory and into the world.

The original constraint languages were created to demonstrate the methodology of modeling with constraint systems. Many strategy was implemented to determine what sorts of data structures could retain enough of the right kinds of information to make the system work best. The goal of getting something that worked overrode the goal of getting something that worked quickly. Since my implementation in C is a second generation constraint language, I had to take ideas from systems that worked and implement them in a system that would work quickly. The whole range of efficiency goals were made much more reachable when I used the UNIX perspective: simple, pragmatic, and general. I tried to make it my most prominent feature that the implementation assumptions I made would "not get in the way" of the user.

In this case there are really three classes of users of my implementation: Those studying my thesis with the goal of understanding constraint modeling languages, those using my code as the basis of their own constraint languages, and those actually modeling with my system. With

such an audience, generality through simplicity and careful choice of assumptions was the only reasonable method of implementation. Here the notions of philosophical correctness and good implementation come together.

My ideal of a good implementation is that it satisfies all three classes of users. Since C is widely available, and widely used, many people will be able to study constraint languages from this implementation. By doing things simply and with generality, the system should be easy to understand. By keeping in mind the philosophy of taking what one needs and throwing the rest away, I have produced code of value to many even if only small pieces of it turn out to be useful. For these reasons, writing in C has done much to help me make a good implementation, but there is more. The C language is considered to be a low level language with efficiency on par, although slightly lower than assembly language. By writing in an abstract style in C I have brought together understandability and efficiency.

Here is a simple example: Constraint languages usually have to look though lists of data to locate new values an to propagate them throughout the network. This means that the implementation language should be good at following pointers through long chains. Happily, the manipulation of pointers is something that C is particularly good at. Much of the efficiency in C programs comes from the excellent way it deals with pointers. Other ways in which C proved helpful and efficient will be covered in the section on implementation.

Although I have set ambitions goals for my programs, and their description, this paper, I have been helped by recognizing constraint languages as well worth studying, and C as a language well suited to bringing them to my intended audience.

History

In this chapter I will survey other work relevant to the study of constraint languages in general and my implementation in particular. There will be six subsections. In each of them I will call attention to specific points that illustrate or contrast with my work.

Section one will cover Ivan Sutherland's Ph.D. Thesis [Sutherland 63] entitled *SKETCHPAD: A Man-Machine Graphical Communication System.*

Section two will discuss Patrick Winston's Book [Winston 79] *Artificial Intelligence.* Chapter 3 is entitled "Exploiting Natural Constraints". It describes the use of constraints in computer vision and in natural language processing.

Section three traces constraints in the domain of electrical circuits through four papers from the M.I.T. Artificial Intelligence Laboratory.

Section four discusses Alan Borning's Ph.D. Thesis [Borning 81] *Thinglab -- A Constraint-Oriented Simulation Laboratory.*

Section five mentions Phyllis Koton's work with the description system Omega [Koton 81].

Section six introduces Guy Steele's Ph.D. Thesis [Steele 80] *The Definition and Implementation of a Computer Language based on Constraints,* the work that largely inspired this one. This section is a brief overview. I will say much more about Steele's thesis in the chapter on implementation.

Section seven will cover an article from *Byte* magazine on TK Solver [Williams 82], a constraint language for home computers.

1. SKETCHPAD

Sutherland's *SKETCHPAD* thesis was revolutionary for its time. Published in 1963, it contained ideas that should be incorporated into modern computer aided drawing systems. It also has become the foundation paper for the study of applying constraints to programming problems. Papers by Sussman, Steele, and Koton which I discuss later in this chapter, all refer to this work.

SKETCHPAD used the point plotting graphics display of the Lincoln Laboratory TX-2 to produce pictures for plotting. But the system was intended for more than the duplication of the functions of pencil and paper. Recognizing the differences between a computer generated image, and a line drawing on a sheet of paper, Sutherland produced a system that relied heavily on the computer's ability to remember special information about images, in addition to the images themselves.

For example: to draw two lines of equal length on a piece of paper, one gets out a ruler, decides length of the two lines, and draws them. If it turns out that the initial length was wrong, the lines are erased and redrawn. With *SKETCHPAD,* one draws two line segments using the DRAW command. Then one tells the computer to constrain them both to be the same length. Then no matter what length is chosen, the lines will always be the same length. This may seem a trivial point, but think of trying to draw 30 lines the same length, or 100 lines the same length, and having them not all fit on the paper, the first or second time they are drawn. Some modern drawing systems, conceived as replacements for pencil and paper, would also require re-drawing all the lines to re-size the lines, even though they are the same size with respect to each other.

Because of it's experimental nature, *SKETCHPAD* only contained 17 constraints. Although they were graphical constraints on size, distance, and position, Sutherland pointed out in his thesis that new constraints could be easily added and that they could have meanings not limited to the picture domain.

One other significant portion of the thesis is Sutherland's description of how he evaluated

expressions to satisfy constraints. He implemented two methods: the multiple iteration relaxation method, and the single pass "maze-solving" method. The single pass method looks for variables that can be completely evaluated and deleted from the list of unknowns. This process proceeds until all unknowns have been deleted from the list.

If they are not all satisfied, the relaxation method must be used. The relaxation method uses an error computation equation pre-defined with each constraint type. Variables are modified incrementally to minimize error. Although not explicitly stated, I presume there were capabilities for either converging to an error of zero and halting, or selecting arbitrarily small error values to establish the termination condition.

The rest of the thesis consisted of descriptions of the drawing system, the data structure, the program structure, the hardware, and proposals for future work. The descriptions, although historically interesting seem to focus on methods that are no longer appropriate. For example, his ring-type data structure could more easily be implemented in lisp with lists or in C with pointers and structures.

Sutherland's proposals for future study present an interesting pattern. Only recently have de Kleer and others used graphical input to circuit simulators. (I will be further discussing de Kleer's work elsewhere in this chapter.) There are now highly refined graphical displays, and yet there are few computer-aided drawing systems that meet or surpass pencil and paper except in certain carefully contrived problem domains. Those systems that do exist commonly contain recursive copying capabilities like those of *SKETCHPAD,* but I am not aware of any drawing system that makes any extensive use of graphical or general constraint satisfaction.

2. Patrick Winston on Constraints

Moving away from specific implementations for the moment, I would like to mention some things from Patrick Henry Winston's book *Artificial Intelligence.* Chapter three is entirely devoted to constraints in the domains of vision and speech. It is not immediately obvious that one could successfully use a constraint language in these domains.

The chapter is divided into two sections, the first one describes the application of constraints to a tiny subfield within vision called scene analysis, determining the meanings of the various lines in a drawing. Within a drawing, lines might represent boundaries between objects, cracks in objects, shadows of objects, or hints about the shapes of objects.

To perform scene analysis, the way lines come together must be enumerated. There are nine of them: L, ARROW, FORK, T, PSI, X, K, PEAK, and KA. Initially dealing with all nine junctions seemed too hard, so the number of junction types was reduced to four: L, ARROW, FORK, and T by assuming there were no shadows, and by looking at only those pictures with lines that intersect at three surfaces.

Simplifying to four types of junction made it possible to enumerate every way lines could come together from every possible viewpoint. Although there were 208 combinatorically possible configurations, there are only eighteen physically consistent ways lines could come together sticking out, sticking in, or as boundaries. If one drew a boundary around the outer edge of the objects in the picture, and then filled in all the cases for which only one answer was possible, the constraints quickly converges to describe every line in the drawing. This shows the beauty of scene analysis using constraints.

Then shadows were re-introduced. Surprisingly, shadows reduced complexity because they helped constrain the locations of objects. Finally, the other 5 types of junctions were permitted again. Although the number of combinatorically possible combinations gets high, the physically possible combinations become progressively more constrained. The most striking example was

the KA junction. Out of 312 million combinatorically possible configurations only 30 physically sensible labelings result. Allowing all nine types of junction yields a task easily solvable by computer using constraints.

Taking advantage of the physical reality of objects reduces the search space dramatically. Propagation of constraints is a demonstrably good method for solving this problem. To solve this problem directly with a constraint language, one would wire up a network of units that modeled verticies. For each number of lines joined at a vertex there would be a different type of constraint unit. Then the values BOUNDARY, CONVEX, CONCAVE, SHADOW, and CRACK would be asserted until there were no inconsistencies in the network. If one assigned all the BOUNDARY values to the perimeter of the network picture, and if each vertex unit only asserted a new value on an un-constrained line when there was only one possible output value, the network would converge to complete specification in one pass.

The second part of the chapter covered a microscopic portion of language understanding called sentence analysis. Just recognizing whether a word describes an action or the object of an action is hard. Applying constraints to the determination of tiny pieces of meaning from a subset of english was not as striking a demonstration as was the discussion of scene analysis. Answers did not converge as quickly because the ambiguities could not be eliminated as effectively. The presentation was not a bottom up rigorous sequence. Even so, the paper introduced many good ideas, and gave many good examples.

I draw three conclusions from the second part. Sentence analysis is too hard to be trivially solvable in the way that scene analysis turned out to be. But secondly, by recognizing the way certain words constrain the uses of other words, the search space of possible meanings is reduced. Finally, I believe that when constraint languages become more widely available, someone will use constraints a part of a method for consistently parsing meanings out of sentences.

3. Constraints and Circuits

This section covers four M.I.T. Artificial Intelligence Laboratory papers relating constraints and electrical circuits. All four of the papers I cite were written in part by Gerald Sussman. I was first exposed to constraint systems in a course of his I took in 1979. I find it an interesting coincidence that my understanding of constraint systems proceeded along the same direction that these papers follow if they are taken in chronological order.

The first paper [Sussman 75] is called *Heuristic Techniques in Computer-Aided Circuit Analysis*. It deals with a LISP program for DC electrical circuit analysis called EL. Unlike other circuit analysis systems, EL did not use formal methods like *The Node Method* or *The Loop Method* to solve for unknowns. Instead various inspection techniques were used. These techniques were developed by observing the ways skilled engineers analyzed circuits.

The paper is of primary interest as a good introduction. It defines into existence a whole new way of approaching circuit analysis by computer. Not only does the system get the right answer to the unknowns as would a more classical analysis program, it exposes its reasoning clearly to explain how an answer was determined. With this additional functionality, the complexity of the circuits that could be solved was surprising. I had the opportunity to use a similar system called EESYS in the course I took with Sussman. For those who cannot experience the thrill of having a computer analyze a complex circuit and then tell why it did what it did, this paper provides the next best thing.

Forward Reasoning and Dependency-Directed Backtracking In a System for Computer-Aided Circuit Analysis [Stallman 76] is the name of the second paper. The paper focuses on a new version of EL written in a new "rule language" called ARS (for Antecedent Reasoning System). This paper includes many refinements on the philosophy embodied in the EL system as well as information about ARS, and examples of analyzed circuits.

"Propagation of Constraints" is the name coined for the EL methodology. Propagation of

Constraints is presented as a generalization of Guillemin's technique for solving an electrical ladder network by inspection. Subsequent thought shows propagation of constraints to be general to many more domains than just that of electrical circuits. It names the problem solving method people have been using for years: filling in the knowns and taking short steps to satisfy unknowns until all are satisfied, using new knowledge as it becomes available to simplify the steps. Following the examples that show how EL works, one quickly sees how Propagation of Constraints works in practice.

Some of the internals of the new implementation of EL are described. The primitive parts of a circuit are devices and nodes. Devices can only be connected to other devices through nodes. The system deals with running laws when parts of a device get new values. The laws determine if enough information is present to run an equation to compute a new value. Most of the work of monitoring values and running laws is done not by EL, but by ARS. EL is a collection of cleverly expressed laws that ARS operates on by default.

The ARS system is a general constraint modeling language. Although the discussion of ARS is not very deep, some important parts are described: ARS works by running rules over a database. Rules are queued up to be run when some datum receives a new value. When two rules assert a value on a common place in the database, either a coincidence or a contradiction occurs. If a coincidence occurs, it is remembered that the value came from two places. If a contradiction occurs, the backtracking mechanisms runs.

ARS contains an interesting attempt at a backtracking mechanism through the implementation of NOGOOD's. When a chain of reasoning has been found to produce a contradiction, it is classified as NOGOOD. Through the mechanism of *out'ing* all the assumptions that gathered together to make the contradiction, are rendered untrue. This process can quickly be reversed by *unout'ing* the set of assumptions. The ideal stand alone constraint language would have some backtracking mechanism to automate the recovery from contradictions. Despite the amount of thought that has gone into the various implementations of NOGOOD's, no implementation of

NOGOOD's has yet been shown that works for all patterns of contradictions. I regarded the implementation of NOGOOD's as too hard for my first version implementation. Unfortunately then, mine just complains to the user when it finds a contradiction and as yet has no automated backtracking facility.

In the third paper [de Kleer 78], *Propagation of Constraints Applied to Circuit Synthesis, the work of* EL has been turned completely around. Instead of asserting circuit parameters and solving for unknown values, a new system SYN makes several models of a circuit and solve for circuit parameters. SYN does a lot of algebraic manipulation. The most interesting aspect of SYN is its ability to simplify its calculations each time the user specifies a model covering another viewpoint. The major problem with SYN is that the algebra describing even simple circuits gets too complicated too quickly. Even the course of simplifying can create intermediate steps that overrun the computer's address space. Even so, SYN represents a novel use for constraints, and demonstrates the utility of parallel viewpoints.

The fourth paper [Sussman 81], *Constraints -- A Language for Expressing Almost-Hierarchical Descriptions*, covers three topics in great depth. It demonstrates the use of a constraint modeling language in extremely broad terms. It shows explicitly how a general constraint language can model electrical circuits in the same way EL would. Finally it presents SLICES, the method of constructing several models of the same circuit to show multiple viewpoints. SLICES was used in SYN but not named until this paper.

As I read the first three papers, I began to fear that the constraint language I chose to develop would prove too low level to be interesting to anyone. Without NOGOOD's to automate backtracking, and with no clear way of propagating information about voltage and current along single wires only "wide" enough for integer values, I began to think I had thrown everything away. The modeling techniques demonstrated in this paper reassured me that there indeed was a way to make advanced things work with only the simple system I had conceived.

The paper begins with an excellent introduction to basic modeling with constraint units

wired together. It then demonstrates how one would create a resistor as a five pinned constraint unit. The problem of wiring a resistor is solved by creating two pin terminal units, and two terminal node units. Then the resistor is changed to have two terminal units hanging off of it. Circuits are not wired along simple integer wires, they are connected with abstract units called nodes and terminals. Given this methodology, a new user could easily understand how to take primitive constraint units like add and multiply, and create complete models of circuits just like those in EL.

By themselves the first two topics are sufficient for significant new insights on applying constraints, but the paper goes on. SLICES is described. Multiple viewpoint multi-models are demonstrated. This paper spells out all that is needed to introduce computer modeling with constraint languages to new people. However, the topics covered are more easily understood if the previous three papers are read first. Guy Steele, in his Ph. D. thesis, [Steele 80] shows in meticulous detail the constructs demonstrated here.

4. Thinglab

This section covers a paper by Alan Borning *The Programming Language Aspects of Thinglab a Constraint-Oriented Simulation Laboratory. Thinglab is a complete stand alone constraint* modeling language written entirely in Smalltalk. It is unfortunate that the Smalltalk language is not more widely available. The Thinglab simulation system comes close to being the ideal constraint language. It is not only able to assert constraint relations between objects, it permits propagation of many different types of information along a single "wire" as one message. Additionally, Thinglab contains a powerful graphical user interface. Thinglab looks like a more general and more refined instance of its ancestor SKETCHPAD.

If the goal is to sit down and attempt constraint modeling, Thinglab is, at this time, the system of choice. From the description given in the paper the system is ready for use in many applications. The existing primitives look sufficient for a lot of modeling, and the method for adding new parts to the modeling repitoire looks easy enough. There are graphical relations, mathematical relations, and there are methods for exposing objects in a variety of forms onto the screen. With all this going for Thinglab, there seem to be two prominent problems.

It is difficult enough to get an implementation of Smalltalk. The language is not being made available widely enough yet. Once one gets a Smalltalk implementation, one then has to get the Thinglab graphics running on one's favorite host machine. I strongly suspect that the reason Thinglab turned out so refined was that it took advantage of many of the advanced features available to Smalltalk systems running on Alto computers. Moving such a perfectly tuned product into the wider world of many machines may prove harder.

Secondly, in moving from the explicitly typed environment of C to the typeless environment of Smalltalk clouds the issue of how difficult it is to propagate certain flavors of data through a constraint network. Smalltalk's generalized message passing system simplifies the wiring up of constraint relations, but at a cost. Although cases in Smalltalk might be considered types, the efficiency issues are nebulous and ill-defined. This is an area which will require much

study in the future.

Thinglab contains three ways for satisfying constraints: The first two are the one-pass, and relaxation methods from SKETCHPAD. The third method is the one my implementation copied from Steele's, which Borning describes as "Propagation of Known States." Steele's method is to assert all the knowns and then assert new values if they can be computed in one step from values already asserted. I suspect that the method of propagation of known states together with multiple viewpoints would prove adequate for all constraint modeling, since the one-pass method is the same thing in reverse, and the relaxation method could be done with a constraint unit designed for iteration.

5. Constraints in System Omega

Phyllis Koton experimented with constraint systems in the Omega description language [Attardi 81]. Her work was published in an internal M.I.T. AI Lab document, *Representing Constraint Systems with Omega,* and is not widely available. Even so, I felt it should be mentioned for completeness. She uses special properties of Omega to avoid certain problems she found in Thinglab and the system by Sussman and Steele later to be called ARS.

Omega is a "reasoning system". It runs inference rules over a description network in an attempt to draw consistent conclusions about the description. The system is documented in *Semantics of Inheritance and Attributions in the Description System Omega.* Briefly, a user asserts portions of a description, relates the portions to each other, and lets the system attempt to draw conclusions. For instances when certain relations are conditionally true, Omega provides a mechanism for establishing multiple viewpoints.

Implementing a constraint system in Omega addresses the problem of what to do when local propagation of known states does not solve the network. Not content with Thinglab's relaxation methods, Koton relied on Omega's multiple viewpoint mechanism, and on its ability to draw more global conclusions. Omega never forgets an assertion. Defining a conflicting assertion creates a second viewpoint. This resembles the *out/unout* mechanism in ARS. Establishing new viewpoints works well except when values change frequently. The methods later added to ARS, kept alternate viewpoints to reasoning and away from values were more pragmatic.

Because it was updated to include some of the mechanisms Koton called for, it is possible that ARS was already able to model multiple redundant viewpoints without resorting to Omega's total inability to forget assertions. Koton's work is another attempt to implement a system that will both solve networks which fail to propagate values in one step, and will efficiently backtrack faulty reasoning.

6. Guy Steele's Thesis

Guy Steele's thesis *The Definition and Implementation of a Computer Programming Language Based on Constraints* is a one volume sourcebook on use and implementation of a constraint modeling language. Its 372 pages contain a little bit of everything. It will be difficult for others to match the completeness and depth of coverage that Steele has achieved here. My inspiration and direction come largely from this document, but my work is only a small step in one of the many directions for study suggested by Steele's presentation.

Steele first gives an example of constraint language modeling like those given in *Constraints -- A Language for Expressing Almost-Hierarchical Descriptions.* He describes the constraint method of modeling more completely than any previously cited work. He mentions several side issues such as local behavior of propagation, hierarchies for abstraction, and patterns of backtracking. In review, I had forgotten these were tradeoffs not axioms because his treatment convinced me so completely that his design choices were correct.

Implementation is the central focus of his thesis. He writes two complete systems, a "trivial" one and an advanced one. Both are coded before the reader's eyes and explained almost a line at a time. The trivial implementation contained only the essential features to express a constraint network and took no account of efficiency. The advanced implementation added automated backtracking, nested constraint macro-units and efficiency. Steele demonstrated the use, design, and implementation of a constraint language from the inside.

By the time I finished studying the advanced system, I felt I knew enough to implement something similar in C. My original goal of implementing Steele's advanced system in C changed, however. I found out that there are more ways to use low level constraint system primitives than just the creation of a stand alone language. I also saw that there were design issues that had to be re-thought when moving from LISP to C. I will discuss these two topics in much greater depth in the next chapter.

7. TK Solver

TK Solver is the second constraint language to achieve popularity. The first one was Visi-Calc and its various clones. TK Solver is by the same people that created Visi-Calc, and it is just as revolutionary. Visi-Calc expressed constraint relations between the various cells on a spreadsheet. The constraint relations were simple mathematical expressions, and propagation was by simple substitution as soon as a value changed. TK Solver much more closely resembles such systems as ARS and Omega in that there is a table of variables and a table of rules. One defines rules in algebra, sets some of the variables, and then lets the system solve.

Although it is not as comprehensive or as abstract as Thinglab, TK Solver has a friendly user interface that enables a user to enter equations with decently long mnemonic variable names. The package is designed to display text in a convenient to read form including comments. It also can tabulate and graph outputs of iterated rules.

Interacting with TK Solver with equations instead of the ways Thinglab and ARS require affects the way people think about expressing their ideas to the computer. The learning step of converting problems to a network is unnecessary. Since equations are normally nested in hierarchies, creating abstract macro constraint relations is automatic but does not add much abstraction to help shrink subproblems. The net effect is that many problems can be expressed directly and simply with no additional learning. Harder problems requiring more abstract approaches are still too hard to express.

TK Solver is the obvious second step after Visi-Calc to providing simple, abstract, and direct modeling to computer users.

Implementation

Now I describe of my implementation. My discussion is broken into five parts. Each part progresses to a successively higher level of functionality starting with low level data structures and ending with a high level evaluator.

Part one is about the data structures I created. I will describe how several months of refinement are stuffed into these seemingly simple structures.

Part two talks about real-time storage allocation. I show routines that will dynamically create and destroy the data structures discussed in part one.

Using constraint data-structures from a C program is covered in part 3. A demonstration program will run a simple Farenheit to Celsius converter constraint network.

Part four modifies the code in part three to do a logic simulation problem, which is a more carefully time ordered constraint problem.

Part five points at other code that I wrote that helped me understand the more global issues involved with a complete constraint language.

When I defined the data structures, I attempted to gather a lot of thought into a simple elegant package. I took special care with names, organization, efficiency and completeness. Fully half of my unabridged notes (see Appendix B) are devoted to changes, improvements, and fine tuning of the data structures.

I wanted names for things that would be both technically correct to those who had used constraint systems before, and intuitively satisfying to those seeing constraint systems for the first time. When I finally defined my data structure, I also defined a nomenclature for constraint systems designed to be more approachable than that used by previous researchers.

It takes five data types to wire a constraint network: Node, Cell, Unit, Rule and Core. There are also a couple of housekeeping data types that I will mention as they are needed. The definition of all the data types is in the module constraints.h in Appendix C. I have mentioned units, cells, and rules in previous sections. The reader may already have some intuitive notion of what I mean when I use these terms. I intentionally used my own nomenclature up to this point to produce this intuition.

I used the data structures documented in Steele's Ph.D. thesis as a foundation. My first step was to copy the rudiments of his data structures into C from LISP. Anything I did not understand, I labeled as obscure and copied verbatim. As I converted more of his structures into C I began to see numerous design tradeoffs that were sensible in LISP, but inappropriate in C. The essential difference is that Steele relied on LISP for anything that seemed too hard to implement by hand. I did not have that luxury. For example, a rule was a simple lisp procedure that was called when enough variables had data to satisfy its inputs. A macro constraint was a lisp procedure that pretended it was a human operator wiring up a collection of primitive constraints.

I had to re-think the entire evaluation system from the standpoint of compiled C code. I decided that macro constraints should conserve space and not duplicate anything that should be

sharable from instance to instance. Without LISP to handle storage allocation, value passing, and symbol management, I had to do all these things myself. I had to think of a way that C procedures could easily grab hold of values even deeply nested in macro constraint structures. I had to devise a method to enable rules to find the C procedures they would run. I had to reserve space in the data structure for the names of things. Where once LISP did all these things automatically, the data structure was changed to permit them explicitly in C with elegance and efficiency.

The resulting data structure bears only a slight resemblance to that used by Steele. To those who have extensively studied constraint systems, it is obvious that both Steele's and my implementations contain the crucial things to make constraint propagation work. I call things by different names for simplicity. I use things in different ways for greater generality. There is no longer the need to rely so heavily on the host language. The data structure will hold up in any implementation from a small constraint unit hanging off a big C program to a complete stand alone constraint language. As will be shown, little is wasted in moving to the more complicated implementations.

In addition to the foundation provided by Steele, I worked with an evaluator in mind. As I constructed the data structures, I also wrote the evaluator I would expect to use on them. I was not satisfied with the data structures until I had an evaluator in hand that would sensibly propagate values through them. The evaluator, called eval.c is in Appendix E, and is more fully discussed in part four of this chapter.

A constraint network, in this system, is a collecition of Cells wired at Nodes. Some Cells are constants. Some Cells are associated with constraint Units and are called the pins of the constraint Unit. The evaluation program makes sure that the Values of the Cells around the Node are consistent, and runs the Rules of the constraint Unit associated with the Cells

When the value of a Cell changes, it is awakened. When a Cell is awakened the evaluator goes to the constraint Unit that is the Cell's owner. (If a Cell is a constant it has no owner.) The

Cell is identified as a particular pin of the Unit. The evaluator goes to the Core of the constraint unit for the Rules that are associated with that pin. All Rules associated with that pin are examined. If the Cell forgot its value, any Cells that got their values from a Rule associated with this pin, forget theirs as well. If the Cell got a new value, Rules with enough information from other pins to satisfy its input conditions are run and they assert values on other Cells.

The evaluator then renders the Nodes of each of these cells consistent. Mine does so by putting the other Cells of each Node in a queue to be awakened after all the other Cells in the queue have been awakened. A first in first out queue seems to do the right thing. As the values of Cells change, their consequences are put on the queue to be evaluated when all the previous consequences have been evaluated. There may be pathological networks for which queuing up Cells is the wrong idea. Most other enqueue rules. I will cover this tradeoff in the evaluation section.

A Node is simply a linked list of Cells and a **supplier**. The code that defines a Node is as follows:

```
Struct node
{
        List *cells;      /* Cells joined at this node */
        Cell *supplier; /* The cell supplying this node's value */
};
```

A Node can only have one value and be consistent. The Cell of the Node that gives that value is called the **supplier**. The Cells of the Node contain internal state that determines which Cells take their values from the **supplier** and which Cell is the **supplier.**

Since C has no primitive data type of linked list I created one for use here. Each list element contains a pointer to the next element or a pointer NULL (which is #defined to be 0) if there is no next element. As shown by the code below, the abstract data type List is a list of Cells.

```
typedef struct liszt
{
```

```
        Cell *l_cell;
        struct liszt *l_next;
}List;
```

The purpose of a Cell is to hold a value, and enough additional state information about the value to let the constraint system use it and change it consistently. The C code to define a Cell is:

```
struct cell
{
        char *name;     /* Pointer to cell's name. */
        Value value;    /* The numerical value of the cell */
        State state;    /* state of the value */
        Node *repository;       /* Points back to repository */
        Unit *owner;    /* Points to unit or NULL if a constant */
        int index;      /* Array index if cell of a constraint. */
        Rule *rule;     /* Pointer to the rule that caused value */
        int mark;       /* Marks for tree walking */
};
```

The first component is the symbolic **name** of the Cell. Rather than define some arbitrary length for all names, I used the convention favored by C programmers: to use a pointer to the symbol. In this way, I leave it to the programs using the data structure to make space for symbols. I permit symbolic names to be shared, to be precisely sized, and to be of arbitrary length. These all depend on the amount of intelligence in the programs using the data structure. Regardless of what fanciness is added, names are, by convention, strings of characters terminated by an 8 bit byte of zeros. This too is standard C style.

The second component is the data **value**. I define an abstract data type through the construction:

```
typedef int Value;
```

If, at some later time Values are made more complicated, changing this one line of text in constraints.h will make all the general purpose routines work correctly. A scan of the uses made of Value will determine if the routine depends on the representation of this abstract data type. Values are the actual data passed throughout the constraint network.

The third component is the **state** of the Cell which is used to determine the significance the data stored in value. The Cell's **state** governs its interaction with the other Cells of the same Node. I defined the five states as an abstract data type:

```
typedef enum state
{
        PUPPET,
        KING,
        FRIEND,
        SLAVE,
        REBEL
} State;
```

These five states have the same symbolic names and basically the same meanings as the ones Steele used in his advanced implementation. I did some re-thinking of how contradictions should be handled (as I mention in my notes), and I decided to get rid of the DUPE state.

When a Cell has no value it's state is PUPPET. It is a ghastly error to use the value of a Cell when its state is PUPPET. The KING Cell is the Node's **supplier**, which means that the KING Cell gives its value to all PUPPET Cells of the Node converting them to SLAVE Cells. A SLAVE Cell has no value of its own. A FRIEND Cell is one whose value independently agrees with the KING. REBEL Cells have values that disagree with the KING. A node can have only one KING, but any number of SLAVE, FRIEND, or REBEL Cells. If, for some reason, the KING Cell loses its value, a FRIEND Cell can become the **supplier** of the node and is made KING. If the KING has no FRIENDs, a REBEL is made KING. If a node has no FRIEND or REBEL Cells, the **supplier** is made a PUPPET and all the SLAVE cells around the node are awakened and told that their value has been forgotten.

The **repository** component points back at the Node that this Cell is associated with. It points at an object of type Node as defined above. It is important to be able to find a Node given one of it's Cells. Otherwise it would be impossible to grab hold of a Node and make it consistent when one Cell changes.

Next is the Cell **owner**. It points to the constraint Unit that owns this Cell as a pin. As

mentioned above, if the Cell is a constant, it has an **owner** of NULL.

The **index** tells the pin number of the Cell if it is owned by a constraint Unit. The Cells that are pins of a constraint Unit and the Rules of a constraint Unit's Core, are indexed. The index number of a Cell tells where to find it in the Constraint Unit's array of variables, and where in the array of Rules to find those Rules associated with that Cell.

The **rule** component points at the Rule that gave this Cell its Value. This is useful when one wants to know *why* a Cell got a particular Value. It is necessary to know the Rule when determining if a Value should be forgotten. If the input of a Rule is forgotten, and if it is shown to be the Rule that caused this Value, the Value is no longer valid and must be forgotten.

The **mark** component is a throwback from Steele's implementation which I decided to keep in case some implementation needed to walk the network of Cells and perform operations only once. An **int** is about the smallest data type C will allocate space for. Although a **char** is smaller, it will generally be aligned on an **int** boundary. Therefore I decided to leave a whole **int** of marks, and leave it up to the various implementations how that **int** was to be divided. There are conventions for dividing **int**'s into enumerations that should be followed to ensure portability. When I knew what which enumerations I was going to use, I employed the **enum** data type to be both portable and abstract.

Conceptually it is the constraint Unit that does all the work because that is what the user wires together and sends values through. In my implementation, the Unit gathers together pointers to the two types of information embedded in a constraint unit: the instance specific information unique to every constraint unit such as the Cells that hold variable Values, and the invariant information among Units of the same type such as the Rules that run over the variables. Here is the definition of a constraint Unit:

```
struct unit     /* An actual instance of a constraint */
{
        char *pname;  /* Printed name of unit */
        Cell **vars;    /* Points to array of variables (use calloc) */
```

```
    Core *ctype;   /* Points back to core */
    int q_rules;   /* number of queued rules */
};
```

The **pname** is the printed name of the Unit. Each constraint unit is uniquely named so that Units that perform the same function but appear in different parts of the network may be uniquely identified. Like the Cell, the Unit holds only a pointer to the name string.

The **vars** component points at an array of Cells. By looking at the Core of the Unit, one knows the number of variables in the Unit. Therefore there is no special termination for the array of Cells. To get to a specific Cell, one takes the index number and goes down that many Cells. The C function **sizeof** tells how big a Cell is. To the evaluator, the pins of a constraint and its variables are the same. However, a user wiring up macro constraints will only see some of the variables as pins. When I talk about propagating values between constraints, I will say pins. When I talk about internal state that cannot be wired to by the user, and about the insides of a constraint unit, I will say variables.

The **q_rules** component is for implementations that want to keep track of which rules are enqueued for this constraint unit. The evaluator I wrote does not use the component, but it may become necessary if too much time is spent redundantly evaluating rules, or if networks will not evaluate stably because of rules becoming enqueued in pathological patterns.

The **ctype** component points to the Core of the constraint Unit. The Core contains all the information that is shared by units of the same type. The name ctype comes from "constraint type". I chose the name Core because it captured the essential idea of a gathering place for the essentials of a constraint type. For example: If several ADDER units were wired together, there would be several Units, but only one Core. In C I defined the Core like this:

```
struct core     /* constraint core or type */
{
    char *symbol; /* name as given in top level expressions */
    int var_ct;   /* number of vars */
    char **names; /* Indexed array of pointers to pin names */
    Rlist **rules;  /* rules triggered when pin gets or loses a value */
};
```

The purposes of the **symbol** and **var_ct** components should be obvious from previous descriptions. Each type of constraint has a unique name like **adder** and **multiplier**. The pointer to the name is stored in **symbol**. The **var_ct** component is used to prevent searches through the arrays from exceeding the bounds of the arrays and falling off the edges.

The **names** array points at the names each pin of the constraint has. For example: a **divider** constraint might have pins labeled **dividend**, **divisor**, **quotient**, and **remainder**. It is not clear whether Cells should be forced to have the same names as the pins they represent. There is no ambiguity in the phrase **the divisor of divider ten**, but one might want that same cell to have a global name as well. Therefore both Cells and Cores have components that point to names.

The **rules** array is also indexed by pin number. Each element in the array is a pointer to a linked list of Rules. An array element then, is really a bucket of Rules indexed by pin number. Appendix C shows the Rlist is almost identical to the List except that one holds Cells and the other Rules.

Of all the data types, that of Rule is the most complex. I wanted it to work correctly for both compiled C function primitive operations and constraint network macro operations. My notes detail the considerable thought I gave to creating macro-constraint Units that would not require duplication of entire networks of Units for every instance of the macro-constraint. I realized that the only instance-specific information was the data in the Cells. Everything else could be stored in a core in the same way as with primitive constraint Units. Since Cores were already cleanly defined as the dispatcher to Rules, the only logical place for macro-constraint implementation was in the Rules.

The two types of Rules are defined as an **enum** data type as follows:

```
typedef enum rule_type
{
        PRIMOP,
        CONSTR,
} Rule_type;
```

It is up to the evaluator to recognize the two types of rules and use them correctly. My implementation uses translation tables to snake through an arbitrarily deep nest of macro-constraints. I will explain the use of translation tables after I am through with my description of Rules. The re-thinking of Rules to use translation tables is unique to my implementation. I was not satisfied with my Rule data type until I had proven that it was general enough by writing the code to do translation tables.

In addition to being complicated, the definition of Rule is ill-structured and ugly. Rules do not have symbolic names, and the components of a Rule do not follow a clean logical sequence. Even so, the components of a Rule will do the job in a general and reasonably elegant way. Here is the definition of a Rule:

```
struct rhule            /* Rule does its work by side effect UGH */
{
        Rule_type ty;   /* Tells what kind of rule this is */
        int *ins;       /* Array tells cell numbers of inputs (-1 flags end) */
        int out;        /* Cell number of Unit that this rule outputs to */
        int delay;      /* Time delay for this rule */
        R_flag flags;   /* flag bits */
        char *text;     /* Pointer to magical text that helps do this rule */
        union {
                Value (*primop)() ;     /* points to C funct. for this rule */
                Core *ragmop;           /* points to constraint for this rule */
        }op_ptr;
};
```

The **rule_type** names whether the rule is a primitive operation (PRIMOP) or a macro-constraint relation (CONSTR).

The use put to the **ins** and **out** components depends on the type of Rule. If the Rule is a PRIMOP, the **ins** component is an array of integers terminated by -1 that names the inputs that must have valid Values for this Rule to run. The evaluator goes down the array checking the pins listed until it hits -1. This means that the **ins** array must be one larger than the total number of variables in the Unit, in case all the pins need to have values. For a PRIMOP, the **out** component names the pin number of the Cell that will get the new Value if this Rule asserts one. Here **ins** and **out** map C procedures which have inputs and outputs into constraints which have

bi-directional pins. As will be shown below, a **plus** constraint requires six Rules among its three pins to convert the directed + function of C to the bi-directional **plus** constraint.

If the Rule is a CONSTR, **out** and **ins** serve the purpose of translating between two levels of abstraction. A CONSTR Rule maps a high level constraint into a collection of lower level ones. Since the pins are still bi-directional at this point, there need be no distinction between ins and outs. However, there are other distinctions that must be made between the higher and lower level constraints.

When running a CONSTR Rule, all that one does is identify the lower level Rule to run, push the table of pin translations onto the stack, and wake up the pin of the lower level Core that corresponds to this one. The **out** component gives the pin number of the lower level Core that this pin corresponds to. The **ins** component is the translation table. When the lower level constraint runs, it attempts to access Cells. Looking upward through the stack of translation tables, the evaluator uses the pin number of the lower level pin in hand as an index into the **ins** array. The element it finds is the pin number of the higher level pin. More will be said about how translation tables work later in this section.

The **delay** component is for use by logic simulation, and can be ignored for ordinary constraint system use. If the evaluator wishes to take advantage of the delay, it can be so customized. The section on logic simulation presents an evaluator that uses delay.

The **flags** component is an abstract data type:

```
typedef int R_flag;
```

Like the q_rules component of Core, it is intended for use in tree walking algorithms and in preventing redundant running of Rules. Nothing that I have implemented uses it. However, this component may turn out to be crucial at some later time.

I mentioned earlier that Rules do not have printed names. They do have a **text** component that holds a pointer to whatever string of text the user of the Rule would like to store.

The last component is the hairiest. The **op_ptr** is either a pointer to a C function if the Rule is a PRIMOP, or it contains a pointer to the Core of the next lower level constraint Unit if this Rule is a CONSTR. By following this pointer, and by pushing a translation table onto the stack of translation tables, the macro-constraint Rule is exposed to be a complete constraint Unit without variables. Only the outermost macro-constraint nest has variables. Its variables are all the cells for all the connected lower level constraint Cores. Several Cells wired together in a network are compressed into one variable of the macro-constraint, and all the mappings of the Cells is handled by translation tables in the ins component of the various Rules.

Now it is time to explain how macro-constraints and the system of translation tables work. Appendix F contains the module frames.c, the code that maintains the translation tables. I create a global data object called world. It is a linked list of translation tables. Each list element consists of two components, the pointer to the translation table and the link to the next translation table. The translation table comes from the ins component of a CONSTR Rule. It is the responsibility of the evaluator to take the ins component and push it onto the stack with the routine add_tab. When one level of translation goes away, the routine clr_top is called to pop the topmost translation table off and throw it away. All accesses to cells must be done through the routine t_access which knows how to thread through the maze of abstract Cores until it finds the Cells of the constraint Unit.

To make it clear how the data structure is used to wire constraint Units into networks and how networks can be compiled into macro-constraints I will build some. Appendix G contains the code necessary to create a complete primitive constraint Core and associated Rules for the **plus** function. It has three pins 0, 1, and 2 with names a, b, and c respectively. The constraint relationship is that a + b = c. Given any two values the third will be computed. If all three are asserted the evaluator has to determine if a contradiction exists and deal with it properly. The code is carefully commented to show exactly how the data structure is filled in with constants, pointers, and functions for a working plus relation.

Given the primitive operator **plus** already exists and there is a Core for **plus** with the necessary rules implemented as C functions. An instance of the **plus** Unit is drawn in figure 1. There is the **plus** archetype and three Cells. The Cells would be in an array associated with Unit **plus1**.

Figure two shows the resulting network if two **plus** Units were wired together with a constant Cell whose Value is 3. A Node serves the important function of joining the Cells in the **vars** array of the two **plus** Units. The three "wires" would be connected to Nodes that would bring in other constant Cells or Units.

To compile this network into a macro-constraint called **thing** one would merge all Cells connected to Nodes. Figure 3 shows the resulting object. A **thing** is a macro-constraint with five Cells, and CONSTR Rules associated with each of them. For example, Cell 0 has one Rule which talks to a Core extracted by the compiler from the old **plus1** Unit. This Core is a PRIMOP, the archetypical **plus**. This **plus** function expects to talk to three Cells with index numbers 0, 1, and 2. The compiler built each of the CONSTR Rules to know in its **out** component the index into the array of Rules within the lower level Core and it generated a translation table, and stuck it into the **ins** component of the Rule. I assume that no lower level constraint Core will have more pins than the upper level one will have variables. This should always be true!

The evaluator transfers control to the lower level PRIMOP Core after pushing the translation table onto the stack. It uses the **out** component of the Rule as the index into the Core's array of Rules to run. This process recurses until one finally hits a PRIMOP core to run. Then the evaluator uses the stacked translation tables to get hold of the Cells with the values to modify. This system, although only demonstrated for a two level constraint function, is general to an arbitrary depth.

To use the translation table, the PRIMOP C procedure calls **t_access** with the pin number of the current Cell. A typical C procedure using **t_access** looks like this:

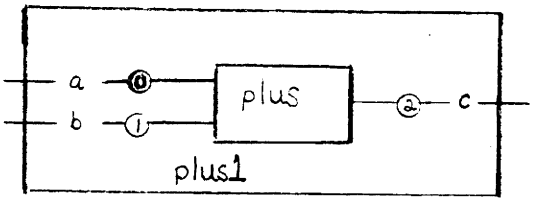
```
Value p_f_2 (u)                    /* c = a + b */
Unit *u;
{
        Cell *p, *q;
        p = t_access (0, u);    /* p become a */
        q = t_access (1, u);    /* q becomes b */
        return ((Value)((int)p-> value + (int)q-> value));
}
```

t_access uses this as an offset into the array of integers for the number of the upper level variable that corresponds to this Cell. It threads its way through the stack of tables until there are none to look at. The variable number in hand is the one in the Unit's array that we want. It grabs the pointer to that Cell and returns it as the Cell to access.

From all the previous discussion, it is clear that data structures are difficult to get right. Much thought must be given to the application of the data through the evaluator. Mutable and immutable data types must be carefully separated. And much abstraction must be used to keep strait all the little bits of data so important for the proper flow of control.

figure 1

An instance of a constraint Unit



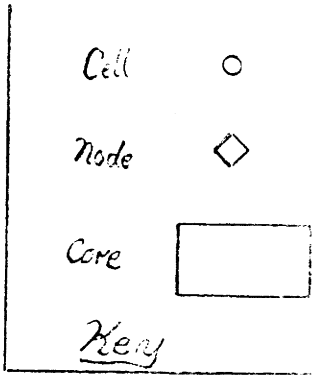


figure 2

A Network

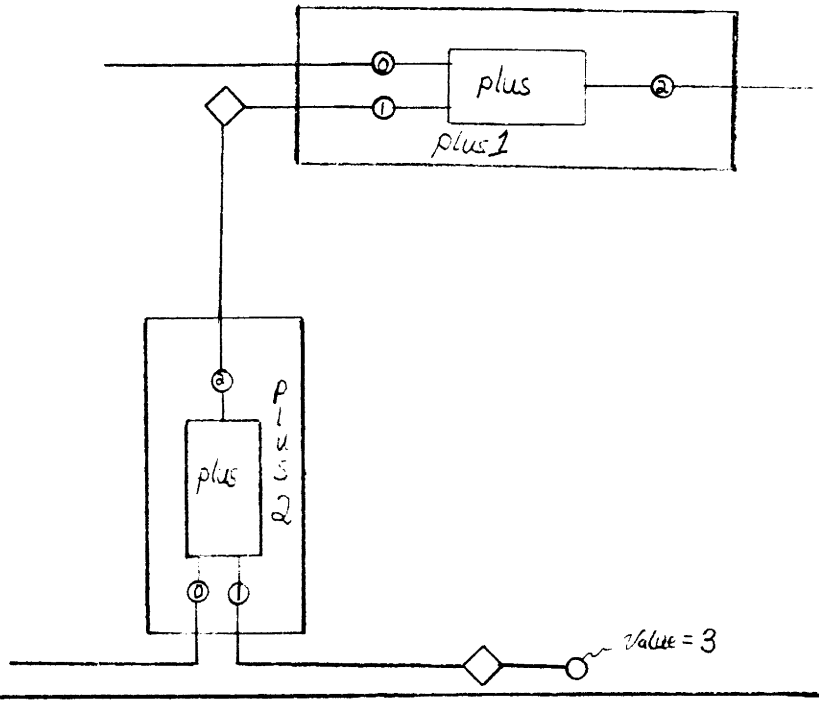
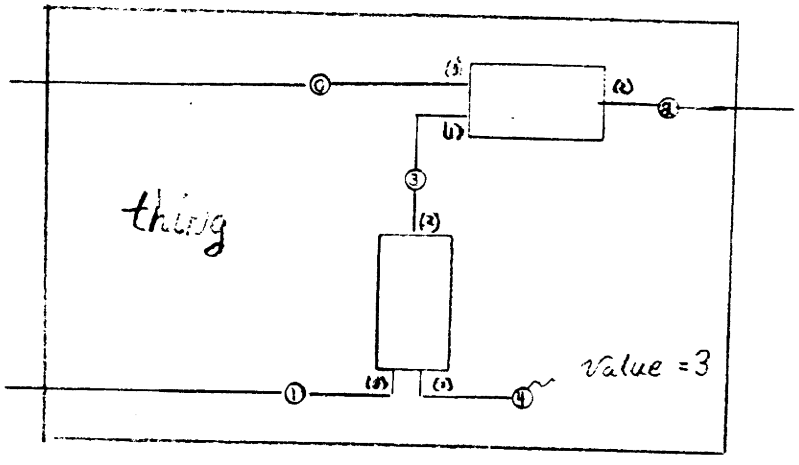


figure 3

An instance of a macro-constraint Unit



Throughout my design and implementation, I have avoided imposing arbitrary size limits on data objects. In the previous section I made special mention of my use of pointers to character strings rather than fixed size strings. I wanted my implementation to remain free of arbitrary limits to both the size of objects and the number of objects of a particular type.

C, unlike LISP or Clu does not have implicit real time storage allocation for data objects. Under ordinary conditions, the size and quantity of each type of data object must be defined at compile time. I decided that, for most uses, this limitation was unacceptable. Therefore I decided to make provision for real-time storage allocation. All the data objects are designed to work in both static, and dynamic environments.

The standard C library contains procedures for dynamic allocating and freeing of storage. The three that I use are **malloc**, **calloc**, and **free**. The **malloc** function takes an unsigned integer naming the number of bytes to allocate. It returns a pointer to the space it allocates or NULL if it cannot allocate the space. For portability the **sizeof** operator will correctly give the size in bytes of any data object known at compile time. The **calloc** function does much the same thing as **malloc**, and is used for allocating several objects of the same size all at the same time. The statement

calloc (1, sizeof(int));
does the same things as
malloc (sizeof (int));
The **calloc** function returns a pointer to the big block of storage exactly the same way that **malloc** does. The only advantage to using **calloc** is that it is more abstract.

The **free** function takes a pointer to storage allocated by **calloc** or **malloc** and makes the space available for other use. It is important to avoid using the contents of space that has been **free**d. Also, as Dennis Ritchie and Brian Kernighan write in *The C Programming Language* "There are no restrictions on the order in which space is freed, but it is a ghastly error to free

something not obtained by calling calloc." Therefore don't free pointers to things unless they are dynamic objects that dynamically allocated. If used carefully, the library functions to allocate and free space work well. However, tiny errors in usage can cause gross errors in operation. I have gathered the creation and deletion routines for the data structures described in the previous section into the module data.c. There are a couple other housekeeping data structures included as well. The file is included as Appendix D.

Creation and deletion of dynamic objects in the C environment has three parts: syntax, initialization, and housekeeping. In LISP, if one gets the syntax right, the housekeeping and initialization happen automatically. But in C, one can have a program that is syntactically correct that blows up because something was used after it was freed, or some other housekeeping violation.

The syntax of storage management is fairly simple. I have already mentioned **malloc**, **calloc**, and **free**. I have also mentioned that the arguments to **malloc**, and **calloc** are **unsigned**. The argument to **free** is **char ***, a pointer to an array of characters. To ensure portability, I cast all the abstract data pointers into **char ***. Pointers, by convention, are really the same data object, and the same size, but the explicit casting guards against surprises in C implementations that do not follow this convention, and permit type checking at all times. The type checking helped locate inconsistent use of my various abstract data types.

Initialization is the setting of initial values as the data types are created. Although a list element is created empty, a Unit initially contains pointers to its name and its Core. Were it not for initialization, all the creation routines would be essentially the same line of code differing by the size of the object to be created.

Housekeeping is the hardest. Here languages like LISP and Clu with implicit storage allocation shine. Since all objects are explicitly created and destroyed, extreme care must be exercised to free only those objects that are no longer in use. For example, if a Node were destroyed, but the Cells joined at the Node were not, they would refer back to the space occupied by the freed Node. There is no telling what would happen. This housekeeping task gets to be a great

headache. I *believe I have followed deletion sequences guaranteed to be robust.* But with the myriad of creations and deletions and the connectivity involved, it is easy to use something inconsistently and have it blow up after some hours of use.

The alternative to explicit management of storage is implicit management of storage. The latter is easier to keep clean, but there is additional machine overhead. For example, C assumes that the programmer uses data types correctly. If the compiler generates the code, it runs; even if it calls for taking the square root of a character. Clu, with its run-time type checking prevents inconsistent use of data types, but at the expense of checking for consistency at every use. Run-time type checking can be made arbitrarily efficient, and can take only small amounts of time on powerful machines, but the expense remains. The LISP storage allocator is written to guarantee that only unused objects are returned to the pool of free space. Yet garbage collection is the ugliest thing to get right in a LISP implementation. It too can be made arbitrarily cheap but will never reach a cost of zero.

Stacks, queues, linked lists, and complicated structures, are all created and destroyed in the run-time environment. By implementing them all by hand I learned a great deal about the tradeoffs involved with various implementation strategies. For example, I worked for a long time to create a network structure of Cells, Cores, Nodes and Units that would not change much over time, so that allocation and de-allocation costs would be minimized. I defined abstract data types, and then checked them with **lint**, the Emily Post of C style. I refused to do much of any run-time type checking, but I did check for simple inconsistencies of usage like trying to wake a Cell that was unconnected to a Node. I re-used objects wherever reasonable, but throw away and re-created when that took less overhead than trying to see if an object was re-usable.

I did all the checking for consistent creation and deletion at the time I wrote the program. In retrospect, I would have liked to have some kind of storage allocator like that which underlies LISP to handle my garbage collection for me. I wish I had an underlying allocation system that would have been able to watch over my creations, accesses, connections, disconnections, and dele-

tions and automated the process of consistency checking. I know enough about the issues involved so that I would not create and destroy objects with wild abandon. One of the major reasons I initially chose C over LISP was to avoid unwanted storage allocator overhead. Having done it the hard way, I now believe I see ways to use the LISP storage allocator without letting it run too often.

Using **malloc** and **calloc** for such things as stacks and queues turns out to be fairly inefficient. I wish that a stack like the one C has for passing runtime arguments was available. Then I would not have all the procedure call overhead for something that grows and shrinks as fast as a queue or a list as things are being evaluated. Clearly, there are many linked lists of things that have to exist simultaneously and that change quickly. Careful coding of allocation routines is necessary, and using malloc here is probably a simple start, but not good enough. In the code, I mention that the queue is initially written to allocate and de-allocate space for every enqueue and dequeue. I believe that this is too wasteful. A new version should be written that allocates space for ten or so queue entries at a time and uses that space before allocating more.

The conclusion I draw is that C does not help enough for housekeeping of complicated dynamic structures, but careful coding can create working efficient code that has little machine overhead.

There are three basic types of data being created: simple strings of characters, simple structures like the parts of a queue or the connective tissue in a linked list, and hairy structures like Cells and Cores. I will now show the code for one of each kind.

The simplest thing to create and destroy is a string of characters. All that is necessary is to know how long the string is that you want to create. I designed the **mk_sym** routine assuming some procedure with a very large buffer would completely assemble the string and then call **mk_sym** to inter the string permanently. I did sacrifice some efficiency by making a procedure call to **umk_sym** instead of calling **free** directly. I decided it was more important to centralize the deallocation of each abstract data type in data.c. If the overhead is too great, the calls to the un-

make functions can be replaced by direct calls to **free** Here is the code to make and unmake a character string:

```
/*
 * Allocate space for a symbolic name len chars long.
 */

char *mk_sym (len)
int len;
{
        char *c;
        len++;
        if ((c = calloc (len, sizeof (char))) == NULL) mem_err ("mk_sym");
        return (c);
}

/*
 * Free a symbol.  One procedure call less efficient, but more consolidated.
 * Beware of GHASTLY errors!  Use care when calling this.
 */

umk_sym(s)
char *s;
{
        free (s);
}
```

Once the space for the string has been allocated the library routine **strcpy** copies the text into the space allocated.

A linked list is a structure containing two components, the list element and a pointer to the next piece of the list. A queue is a special linked list that differently manages where new elements are added. Here is the code for list creation and deletion.

```
/*
 * Return an empty list element with both pointers set to NULL
 */

List *mk_list ()
{
        List *l;
        if ((l = (List *)malloc (sizeof (List))) == NULL)
                mem_err("mk_list");
        l-> l_cell = NULL;
        l-> l_next = NULL;
        return (l);
```

```
}

/*
 * Destroy a list element.
 */

umk_list (l)
List l;
{
        free ((char *)l);
}
```

Notice the cast to **List** in **mk_list** and the cast to **char** in **umk_list**. These are to keep the abstract data types pure and thereby to help eliminate inconsistent housekeeping.

Finally there are complicated structures like Nodes and Cores. The code to create and destroy a node is shown below:

```
/*
 * Return an empty node all ready to have cells connected to it.
 */

Node *mk_node () {
        Node *c;
        if ((c = (Node *)malloc (sizeof (Node))) == NULL)
                mem_err("mk_node");
        c-> cells = NULL;               /* Initialize everything */
        c-> supplier = NULL;                    /* to zero for an */
}

/*
 * Destroy a Node.  Please remember to strip off all the cells before
 * you destroy it!
 */

umk_node (n)
Node *n;
{
        free ((char *)n);
}
```

Nodes add initialization to allocation. More complicated structures are in data.c but they all act about the same way a Node does. They differ in the specific data going to their respective components, and in their complexity. The Node is the simplest of the class of complex structures but is still representative. From these examples, reading the rest of data.c in Appendix D should be no problem.

This section presents a working constraint system. It is a farenheit to celsius conversion constraint system like that described in Steele's Ph.D. thesis [Steele 80]. I use the data structures, some real-time storage allocation, and an evaluator. The demonstration system takes the form of a C program. It allocates the Cells, Nodes, and Units it will need. Then it connects them together with an **attach** function much simpler than one that would be used for a network that is alive and running. It then asks the user which end of the constraint network to put the initial value. It takes in an initial value, and puts it into the network. The network is then run, and the opposite end from that specified for input is interrogated for the output value.

Appendix J contains the program fccf.c, and two support modules cell.c and primop.h. The primop.h module is intended to name all the primitive operators that the system will know about. Currently it is three lines long:

```
extern Core *plus;
extern Core *mult;
extern Rule *IS_CONST;
```

The file is included in the evaluator module, and in all the modules that define primitive operators. The three primitive operators defined are the addition operation with the Core **plus**, the multiplication operation with the Core **mult**, and the assertion of constant values through Cells that have the Rule **IS_CONST** as the reason for their having a value.

The module cell.c contains the code to implement the **IS_CONST** Rule. The code segment prints an error message because one should never run the constant rule. The important part of the Rule is the **ins** component. (Remember that **ins** is the list of pins that must have values for this rule to run.) The **ins** component is the null set! When the evaluator attempts to assert a value on a pin, it looks to see if the value is the same as one already present. If so, it assigns the Rule depending on the smallest proper subset of inputs as the reason why the value was set. The null set of inputs is always a proper subset of any Rule. Therefore constant Cells will not get new reasons for existance when Rules run redundantly.

The primitive operator **plus** is defined in the module plus.c which is in Appendix G. It does not use any real-time storage allocation. At the time I wrote it, I was afraid to rely on the storage allocator. The module defines many variables of global scope. The advantage of defining a primitive operator statically is there is no program that has to be run to allocate space and fill it. The disadvantage is the proliferation variables which *must* have names that are unused anywhere else in the system. Using **mk_core** from data.c to create a primitive operator is left to the reader as an exercise. One important kludge must be mentioned: In the module constraints.h the Rule data type defines an **op_ptr** as a **union** of a pointer to a Core or a pointer to a C procedure. It is impossible to initialize a **union** at compile time. Instead of creating a runable initialization procedure, I created an Irule data type (for Initializable rule). At the end of plus.c I name **plus** as a pointer to something that is coerced into a pointer to a Rule from being a pointer to an Irule.

Here is the Irule data type:

```
struct irhule           /* Rule does its work by side effect UGH */
{
        Rule_type ty;  /* Tells what kind of rule this is */
        int *ins;       /* Array tells cell numbers of inputs (-1 flags end) */
        int out;/* Cell number of Unit that this rule outputs to */
        R_flag flags;   /* flag bits */
        char *text;     /* Pointer to magical text that helps do this rule */
        Value (*prymop)() ;   /* points to C funct. for this rule */
};
```

The only difference is that instead of the **op_ptr** component there is a **prymop** component that is always a pointer to a C function that returns an object of type Value.

The module mult.c is in Appendix H. It was constructed by copying plus.c and changing the operations to be appropriate to multiplication. As my notes show, I had to conceive of what to do about run-time errors like attempts to divide by zero. The solution I finally used was to let the Rule itself deal with such contradictions, since they were inherently specific to the running of the rule.

There are two important aspects of primitive operators that I would like to mention: Because constraint relations are bi-directional, special care must be taken to correctly apply uni-

directional C procedures. Adding bi-directionality requires some redundancy and inefficient use of space. Perhaps there is a way to automating and compressing the information naming which Cells are listened to for assertion of each value. Applications that don't require the added functionality of bi-directional functions may turn out to be too expensive to implement right now. Secondly, it is important to remember that input values come from the Node **supplier**. This means that input values must be acquired by calling **node_val** on the Cell returned by **t_access**.

Although there is an **attach** function in the module wire.c, (see Appendix I) I decided I would simplify the work since I would wire the network all at once. I also wrote routines to replace those in frames.c because I would have not macro-constraints, but I did not want to confuse myself by editing **t_access** out of the evaluator. The versions of **add_tab** and **clr_top** provide the useful error diagnostic of complaining *loudly* if they are ever run. If they did run it would be because the evaluator had somehow got the idea that it was looking at a macro-constraint, and in this implementation, there aren't any. As another simplification, I stuffed constant values directly into the Cells of the constraint Units rather than wire Cells and Nodes for the constant values. The two global Cells **Farenheit** and **Celsius** are connected to Nodes to prove that the conventional strategy works. These simplifications helped to significantly reduce debugging time.

In my discussion of data structures, I said that I wrote an evaluator to satisfy myself that the data structures were sound. After removing a few tiny inconsistencies having to do with determining if Cells and Nodes had values, I used the evaluator shown in Appendix E. (The file eval.c reflects all the bug fixes.) It was with much fear that I began to write the farenheit-celsius converter. I knew that I would need to make the evaluator work, and I was not sure that it would. I was pleasantly surprised to find that it did what it was supposed to when it was supposed to, with only a few easily correctable bugs. I am particularly grateful to Pace Willison for showing me how to use the symbolic debugger **sdb**. Since the big job of getting the evaluator to work is done, more and more of the code can be played with and added. The evaluator was too big to expect to work all at once, but it did anyway.

The real-time storage allocation worked nicely. I encountered a few problems when I forgot to initialize vital pieces of the Unit data structure. Although the code of data.c is hairy and difficult to house-keep, I kept to a subset of the functions, and used them because, they already existed and did what I would have had to do manually several times. There are several incremental improvements that could be made in fccf.c that require using more of data.c such as the creation of the Cores for **plus** and **mult**. Also higher level functions to create and wire several Cells to a Node might be handy.

One thing the reader will notice as my code is examined: I have very verbose error messages. Through the miracle of **print_cell** I know the most complete and unambiguous identity of any Cell that is being watched. When I ran the system for the first time, I had forgotten that I had even written **print_cell**. When the system blew up but printed a perfectly valid Cell name to look at and wonder about, I realized that I not only had good error diagnostics, but that they did not have to be debugged. I left in the monitoring of **enqueue** and **dequeue** operations so that the reader could get a clearer picture of what was happening. They are internal to **enqueue** and **dequeue** in data.c if anyone wants to remove them.

I have done enough talking. I now let the system speak for itself. (Aided by the UNIX scripting program which keeps a running file of what I do at the terminal.)

```
Script started on Mon May  2 03:12:22 1983
Warning: no access to tty; thus no job control in this shell...
% fccf
Is the input farenheit or celsius?  (f or c) : f
What is the input value? 5
Enqueueing cell c of plus adder1.
Dequeueing cell c of plus adder1.
The cell a of plus adder1 now has value -27.
Enqueueing cell a of mult multiplier1.
Dequeueing cell a of mult multiplier1.
The cell c of mult multiplier1 now has value -135.
Enqueueing cell c of mult multiplier2.
Dequeueing cell c of mult multiplier2.
The cell b of mult multiplier2 now has value -15.
The Celsius value is -15.
% fccf
Is the input farenheit or celsius?  (f or c) : c
What is the input value? 5
```

```
Enqueueing cell b of mult multiplier2.
Dequeueing cell b of mult multiplier2.
The cell c of mult multiplier2 now has value 45.
Enqueueing cell c of mult multiplier1.
Dequeueing cell c of mult multiplier1.
The cell a of mult multiplier1 now has value 9.
Enqueueing cell a of plus adder1.
Dequeueing cell a of plus adder1.
The cell c of plus adder1 now has value 41.
The Farenheit value is 41.
% fccf
Is the input farenheit or celsius? (f or c) : c
What is the input value? -15
Enqueueing cell b of mult multiplier2.
Dequeueing cell b of mult multiplier2.
The cell c of mult multiplier2 now has value -135.
Enqueueing cell c of mult multiplier1.
Dequeueing cell c of mult multiplier1.
The cell a of mult multiplier1 now has value -27.
Enqueueing cell a of plus adder1.
Dequeueing cell a of plus adder1.
The cell c of plus adder1 now has value 5.
The Farenheit value is 5.
% fccf
Is the input farenheit or celsius? (f or c) : f
What is the input value? 41
Enqueueing cell c of plus adder1.
Dequeueing cell c of plus adder1.
The cell a of plus adder1 now has value 9.
Enqueueing cell a of mult multiplier1.
Dequeueing cell a of mult multiplier1.
The cell c of mult multiplier1 now has value 45.
Enqueueing cell c of mult multiplier2.
Dequeueing cell c of mult multiplier2.
The cell b of mult multiplier2 now has value 5.
The Celsius value is 5.
% fccf
Is the input farenheit or celsius? (f or c) : 100
f or c, try again c
What is the input value? 100
Enqueueing cell b of mult multiplier2.
Dequeueing cell b of mult multiplier2.
The cell c of mult multiplier2 now has value 900.
Enqueueing cell c of mult multiplier1.
Dequeueing cell c of mult multiplier1.
The cell a of mult multiplier1 now has value 180.
Enqueueing cell a of plus adder1.
Dequeueing cell a of plus adder1.
The cell c of plus adder1 now has value 212.
The Farenheit value is 212.
% fccf
Is the input farenheit or celsius? (f or c) : c
What is the input value? 0
Enqueueing cell b of mult multiplier2.
Dequeueing cell b of mult multiplier2.
```

```
The cell c of mult multiplier2 now has value 0.
Enqueueing cell c of mult multiplier1.
Attempt to divide by zero, value unchanged.
Enqueueing cell b of mult multiplier2.
Dequeueing cell c of mult multiplier1.
The cell a of mult multiplier1 now has value 0.
Enqueueing cell a of plus adder1.
Attempt to divide by zero, value unchanged.
Enqueueing cell a of mult multiplier1.
Dequeueing cell a of plus adder1.
The cell c of plus adder1 now has value 32.
Dequeueing cell b of mult multiplier2.
The cell b of mult multiplier2 has value: 0 which contradicts.
 with node supplier  the global cell Celsius which has value: 0.
Dequeueing cell a of mult multiplier1.
The cell a of mult multiplier1 has value: 0 which contradicts.
 with node supplier  the cell a of mult multiplier1 which has value: 0.
The Farenheit value is 32.
%
script done on Mon May  2 03:13:59 1983
```

I run six test cases. The first four prove that the system will convert an arbitrary number to Celsius and then back again. I chose the input value 5 because it seemed to be a nice number to test. It was close enough to 32 to make the system work near a boundary condition. The fifth test case was a popular test number, the boiling point of water. It proved that the system would work at values away from 32. The last test case did two things, it tested the system at the boundary condition 32, and it made the contradiction system work.

Buried in **mult** is a test for division by zero. The reason why an attempt was made to divide by zero was a Rule was run redundantly, and when it looked at the valid inputs 0 * 0 = 0 and tried to re-derive 0 = 0 / 0 it got confused. The two times this occurred, the Cells acting as the divisors were put on the contradiction queue. After all the forward running evaluations were made, the contradicting Cells were dequeued and sent to the function **fix_contra** which right now prints a pretty message to the user naming the Cell and the supplier as contradicting. I thought of keeping the routine silent if the Cell and the Node supplier were equal, but it might turn out later that a buggy primop enqueued Cells that don't really contradict, and right now the noise is harmless, and serves a debugging function. This points out that primitive operators have to be careful in their error testing to take into consideration the possibility of redundant queueing. Either the evaluator has to be made smarter to prevent redundant queueing, or the operators have

to be made smarter.

The final test is to feed **fccf** 32 degrees farenheit and make sure that it correctly produces 0 celsius as an output.

```
Script started on Mon May  2 03:36:51 1983
Warning: no access to tty; thus no job control in this shell...
% fccf
Is the input farenheit or celsius?  (f or c) : f
What is the input value? 32
Enqueueing cell c of plus adder1.
Dequeueing cell c of plus adder1.
The cell a of plus adder1 now has value 0.
Enqueueing cell a of mult multiplier1.
Dequeueing cell a of mult multiplier1.
The cell c of mult multiplier1 now has value 0.
Enqueueing cell c of mult multiplier2.
Dequeueing cell c of mult multiplier2.
The cell b of mult multiplier2 now has value 0.
The Celsius value is 0.
%
script done on Mon May  2 03:37:06 1983
```

No problem.

By producing working output, I have proven that my data structures actually do propagate constraint information. For explicit details on the construction of the network, refer to Appendix J. My commenting in fccf.c was particularly verbose. I have indeed created a basis for constraint language modeling in C.

This section describes the modifications made to the evaluator to do logic simulation. There are two differences between constraint propagation and logic simulation. Logic simulation is almost always modeling objects with uni-directional inputs and outputs, as opposed to bi-directional input/output pins. The only applications where bi-directionality would be useful would be to model loading on lines, and to recognize when an output is trying to force another output into an inconsistent value. By using care in modifying the constraint evaluator, these two applications might still be possible to implement. The other difference is that the units being modeled have an explicit propagation delay. Special care must be taken to ensure that everything happens at the right time. The new evaluator is in Appendix N.

This section also demonstrates a working logic simulation, of an oscillator using three inverters. Actually, the implementation uses two inverters and a NAND gate wired to look like an inverter. This is a little harder and therefore more interesting.

One low level change that I made was to exercise my option to modify the kind of passed as a Value. I made a compile time option to constraints.h so that if **LOGICAL** was defined, Values changed from **int** to and **enum**. When I did this, I had to change the nature of cell.c, and a few other files to return an element from the **enum** instead of a zero **int** when they failed. These lines were inserted in the file to keep **lint** quiet. They are commented as such in the source files.

The section on data structures briefly mentioned the delay field of rules. The constraint evaluator made no use of this component, but the logic evaluator will rely on it. The central difference between the logic evaluator and the constraint evaluator is that the logic evaluator has replaced the first in first out queue with a priority queue.

A priority queue is a linked list of first in first out queues, each with an associated time offset. Here is the C code that defines a priority queue data structure. It is taken from

constraints.h and also appears in Appendix C.

```
typedef struct pqueue          /* The insides of a prio queue */
{
        Queue *pq_queue;
        int pq_offset;         /* Relative time offset from previous */
        struct pqueue *pq_next;
}Pqueue;
```

The **pq_next** component points at the next element in the list of queues. The **pq_offset** component is an integer that tells how many time units the current element is later than the current one. I do not use an absolute time scale because that might overflow the range of integers, and the range of integers is not the same for all implementations of C. I did not want to implement a time base using modular arithmetic, because I did not want to think about making it work without errors. When the **pq_queue** component of the head of the list is empty, the next attempt to dequeue a cell will move the next element of the list to the head. Since it is now the zero offset, the only time scale that would make sense is one that relates elements of the list to their predecessors. This is a simple and robust scheme.

The first in first out queue is holds the cells that want to be awakened. Although I mentioned the use of a first in first out queue earlier, I did not show what it was made of.

```
typedef struct q_link                  /* The insides of a queue */
{
        Cell *q_cell;                  /* The Cell enqueued */
        struct q_link *q_next;         /* The link to next */
}Q_link;

typedef struct queue                   /* An actual queue */
{
        Q_link *q_root;
        Q_link *q_last;
}Queue;
```

The only difference between a queue and a linked list is that both the first and last elements are remembered in easy to find places. The module data.c contains the code that performs enqueue, and dequeue operations for both the first in first out and the priority queues.

I needed only a few changes to the evaluator to convert its operation from a first in first

out to a priority queue. Any time a cell was to be enqueued, I added code to specify a time offset which comes from the rule. Since the rule was running, I knew the time offset would be from zero, and the priority queue would do the rest. The UNIX **diff** program provides the following helpful list of differences:

```
148c159
< wake_slaves (node)
---
> wake_slaves (node, o)
149a161
> int o;
158c170
<                       enqueue (wake, me);
---
>                       penqueue (agenda, me, o);
225c237
< fix_node (c)
---
> fix_node (c, o)
226a239
> int o;
239c252
<                       wake_slaves (n);
---
>                       wake_slaves (n, o);
242c255
<               else if(other == c) wake_slaves (n);
---
>               else if(other == c) wake_slaves (n, o);
247c260
<                       wake_slaves (n);
---
>                       wake_slaves (n, o);
279c304
<                                       wake_slaves(n);
---
>                                       wake_slaves(n, o);
286c311
<                       wake_slaves(n);
---
>                       wake_slaves(n, o);
299c324
< forget (c)
---
> forget (c, o)
300a326
> int o;
310c336
<                       if (c-> repository != NULL) fix_node (c);
---
>                       if (c-> repository != NULL) fix_node (c, o);
```

```
354c380,381
<                       int ivar, *inpt;
---
>                       int o, ivar, *inpt;
>                       o = r-> delay;
358c385
<                               if (ans-> rule == r) forget (ans);
---
>                               if (ans-> rule == r) forget (ans, o);
380c407
<                       fix_node (ans);
---
>                       fix_node (ans, o);
```

Lines beginning with a less than sign are from eval.c. Those beginning with a greater than sign come from lseval.c. The numbers with letters are editing instructions from diff telling how to convert from eval.c to lseval.c: c for change, a for append, d for delete. The numbers correspond to line numbers within the two files, eval.c on the left and lseval.c on the right.

To make things look pretty, I modified the **print_cell** function to print out TRUE and FALSE instead of integers. Here is the list of differences:

```
87c87,98
<       fprintf (stderr, "%d",(int)v);
---
>       switch (v)
>       {
>       case T:
>               fprintf (stderr, "TRUE");
>               return;
>       case F:
>               fprintf (stderr, "FALSE");
>               return;
>       default:
>               fprintf (stderr, "*BAD*");
>       }
>
```

The *BAD* message is a diagnostic telling that somebody tried to print the value of a Cell that was not a valid Cell Value. In practice, the system will blow up with a core dump for most accesses that are not pure and correct.

Another change I made for personal taste was to eliminate contradictions and replace them with the message that a node value was being superceded. Here is the list of differences im-

plementing that change:

```
254,255c269,280
<                       c-> state = REBEL;
<                       enqueue (contra, c);
---
>                       other-> state = REBEL;
>                       n-> supplier = c;
>                       fprintf (stderr, "The ");
>                       print_cell (c);
>                       fprintf (stderr, " with value ");
>                       print_val (c-> value);
>                       fprintf (stderr, " supersedes the\n");
>                       print_cell (other);
>                       fprintf (stderr, " with value ");
>                       print_val(other-> value);
>                       fprintf (stderr, ".\n");
>                       wake_slaves (n,o);
```

This means that the contradiction queue goes away, and that the slaves of the node must be awakened with the superceding value.

The remaining differences were to those lines of code I installed to keep **lint** quiet. (I mentioned the existence of lines such as these at the beginning of this section.) Here is that list of differences:

```
103c114
<               return (0);     /* keeps lint quiet */
---
>               return (F);     /* keeps lint quiet */
127c138
<               return (0);
---
>               return (F);
131c142
<               return (0);     /* keeps lint quiet */
---
>               return (F);     /* keeps lint quiet */
```

With about thirty lines of changes to a 454 line file, the evaluator knows about time ordering, and logical values. The last set of changes are optional, and will do the right thing if they are not made, since the binary code for the F component of a boolean value is the same as an integer zero, in virtually every implementation of **enum**. If only the modifications to install priority queues are made, the system will run as before, with no change. I think the next version of the

constraint evaluator I write will contain the priority queue and delays, just to permit a wider range of uses.

Before we can simulate logic, new primitive operators need to be defined. Appendix L contains the two logical functions nand.c and not.c. They are simpler than the **plus** primitive operator because they do not implement bi-directionality. (And because an inverter only has two pins.) An improvement on these two functions would be to add some consistency checking rules at the output pins to see if two outputs were trying to force conflicting values. Perhaps such an implementation would re-introduce the contradiction queue. I decided not to add these things, because my primary purpose was to make the evaluator work with a minimum of fanciness.

The program that wires the network, sets the initial values, and runs is called lsim.c. It is in Appendix K along with lcell.c and lprimop.h. The lcell and lprimop modules are not strictly necessary to a working system, but they exist to make explicit the differing Value objects and primitive operators of the logic simulation. They perform analogous functions to the cell.c and primop.h modules from the constraint system.

The only other notable aspect of not.c and nand.c is the **delay** components of the rules. To imply that NAND gates run a bit slower (because they are more complex) than inverters, I set the **delay** component of the NAND gate to 4 and that of the inverter to 3.

The oscillator is a circular structure, and that means that it never gets a stable value to stop with. I aborted the sample run to avoid having to produce a thesis with an embedded document of infinite length. Once again I use the UNIX script program. Here is the sample run:

```
Script started on Wed May  4 02:06:58 1983
Warning: no access to tty; thus no job control in this shell...
% lsim
Network wired, beginning run.
For time offset 0 : Enqueueing cell in of not NOT1.
For time offset 0 : Enqueueing cell out of not NOT2.
Dequeueing cell in of not NOT1.
The cell out of not NOT1 now has value FALSE.
For time offset 3 : Enqueueing cell a of nand NAND1.
Dequeueing cell out of not NOT2.
Dequeueing cell a of nand NAND1.
The cell q of nand NAND1 now has value TRUE.
For time offset 4 : Enqueueing cell in of not NOT2.
Dequeueing cell in of not NOT2.
```

```
The cell out of not NOT2 now has value FALSE.
The cell out of not NOT2 with value FALSE supersedes the
global cell kicker with value TRUE.
For time offset 3 : Enqueueing cell in of not NOT1.
Dequeueing cell in of not NOT1.
The cell out of not NOT1 now has value TRUE.
For time offset 3 : Enqueueing cell a of nand NAND1.
Dequeueing cell a of nand NAND1.
The cell q of nand NAND1 now has value FALSE.
For time offset 4 : Enqueueing cell in of not NOT2.
Dequeueing cell in of not NOT2.
The cell out of not NOT2 now has value TRUE.
For time offset 3 : Enqueueing cell in of not NOT1.
Dequeueing cell in of not NOT1.
The cell out of not NOT1 now has value FALSE.
For time offset 3 : Enqueueing cell a of nand NAND1.
Dequeueing cell a of nand NAND1.
The cell q of nand NAND1 now has value TRUE.
For time offset 4 : Enqueueing cell in of not NOT2.
Dequeueing cell in of not NOT2.
The cell out of not NOT2 now has value FALSE.
For time offset 3 : Enqueueing cell in of not NOT1.
Dequeueing cell in of not NOT1.
The cell out of not NOT1 now has value TRUE.
For time offset 3 : Enqueueing cell a of nand NAND1.
Dequeueing cell a of nand NAND1.
The cell q of nand NAND1 now has value FALSE.
For time offset 4 : Enqueueing cell in of not NOT2.
Dequeueing cell in of not NOT2.
The cell out of not NOT2 now has value TRUE.
For time offset 3 : Enqueueing cell in of not NOT1.
Dequeueing cell in of not NOT1.
The cell out of not NOT1 now has value FALSE.
For time offset 3 : Enqueueing cell a of nand NAND1.
Dequeueing cell a of nand NAND1.
The cell q of nand NAND1 now has value TRUE.
For time offset 4 : Enqueueing cell in of not NOT2.
Dequeueing cell in of not NOT2.
The cell out of not NOT2 now has value FALSE.
For time offset 3 : Enqueueing cell in of not NOT%
%
script done on Wed May  4 02:07:26 1983
```

As the source file lsim.c shows, I created a global Cell called **kicker** to force one wire into a true state. If I had done nothing more than wake up the input of an inverter, after setting its state to KING, the simulation would have worked just fine. The point where the evaluator says that the value of **kicker** is superceded is the first time a value has propagated all the way around the network.

The logic simulator works. To prove that the priority queue correctly orders events in

time I made a few modifications to lsim.c to create a network that test the case of enqueueing something and then dequeueing something that was enqueued earlier by the system, but later in offset time. The program race.c is in Appendix O. (I am almost out of letters of the alphabet, I better finish soon.) Here is the sample run:

```
Script started on Thu May  5 05:59:22 1983
Warning: no access to tty; thus no job control in this shell...
% race
Network wired, beginning run.
For time offset 0 : Enqueueing cell a of nand NAND1.
For time offset 0 : Enqueueing cell in of not NOT1.
Dequeueing cell a of nand NAND1.
The cell q of nand NAND1 now has value FALSE.
For time offset 4 : Enqueueing cell in of not NOT2.
Dequeueing cell in of not NOT1.
The cell out of not NOT1 now has value FALSE.
For time offset 3 : Enqueueing cell out of not NOT2.
Dequeueing cell out of not NOT2.
Dequeueing cell in of not NOT2.
The cell out of not NOT2 now has value TRUE.
The cell out of not NOT2 with value TRUE supersedes the
cell out of not NOT1 with value FALSE.
%
script done on Thu May  5 05:59:33 1983
```

The cell **in** of **NOT2** is enqueued, then cell **out** of **NOT2** is enqueued. Then the crucial thing happens: the cell **out** is dequeued *before* cell **in**. A careful inspection of the previous sample run shows that in every other case, cells are enqueued on a first in first out basis. To make the network in race.c behave as it did, the priority queues had to work.

A Stand Alone System

This section is intentionally short. It describes the contents of Appendix I (if you will, the imaginary appendix.) It is beyond the scope of my thesis to write a stand alone constraint language in C. It is a difficult thing to debug the real-time code to implement it. I did however, wish to fully understand the issues involved with the study of a constraint language in C. Therefore I produced enough code to *describe* a complete stand-alone constraint language modeling system. The code is not meant to work. It is intended to describe my ideas about a stand-alone constraint language as unambiguously as possible. After having written it in C, I see that such a description in LISP would not only be unambiguous, but would also be able to run in limited ways to teach me things about my ideas of implementation. But as the sage says: "Oh well..."

Appendix I consists of five modules main.c, commands.c, wire.c, avl.c, and symtab.c. Perhaps someday either I or some gallant soul with many hours to spend in adb will debug the code, resolve the final inconsistencies and have a constraint language in C. I decided to include it, so that it would be available to other developers.

The module main.c is the once-only start-up code for the constraint language. It creates the queues for ordinary evaluation and for contradiction resolution. It contains a simple parser for symbolic names that the user can type to access the commands.

Half of all the commands needed are contained in commands.c. The primary reason the command implementation is incomplete is that the remaining commands are just more complicated variations on the commands already included. The **define** command is not included because I considered compilation to be a lot of hairy tree walking and compressing into data structures I already understood. I decided to leave compilation in the same category as canonicalization: Beyond the scope of the thesis and the background behind the thesis, but a module for which hooks could easily be left.

At the next lower level is wire.c which contains code for connecting and disconnecting the

network while it is running. I considered the ability to make incremental changes of vital importance and therefore I wanted to define how it would occur.

Symbol table management is hard. The file symtab.c contains the code that describes how I envisioned the ideal method of having symbols: Although Cells, Cores, and Units would have unique names, there was no reason a Cell could not have the same name as a Core, since the parser could determine by context which was referred to. Also the Cells of a constraint Core would have names generic to the type of constraint unit. Every Adder would have an **a**, **b**, and **c** pin. Symtab does all this, and calls a lower level symbol table manager to store the symbolic names.

The module avl.c represents an interesting diversion. I wanted to design the most elegant method of symbol table management for a constraint language in C. I thought about hash tables, but they were of fixed arbitrary size. Extensible hashing is not space efficient. Balanced binary trees are only as big as the data stored, and can search in order log (n) time. I wanted to do insertion and deletion of symbols. After studying a couple of books on algorithms, I wrote C code to do complete avl tree manipulation. I had fun generating C code I never saw anyone else crazy enough to write. I may decide to do something with it later.

Together these routines use the C syntax to describe the entirety of a constraint language as I would implement it in C. They represent one step beyond the code I debugged and ran for this thesis. I actually touched all the issues of a stand alone constraint language despite the difficulty of writing, and debugging one in C.

The sample runs in parts three and four of the implementation chapter showed that the data structures did indeed provide a basis for a constraint and logic modeling system in C. The routines performing real-time storage allocation worked. The development time for programs using the data structures was surprisingly small after I wrote the first primitive operator. With the aid of the UNIX **sdb** debugger, debugging time was fairly short.

From here there are many avenues open for continued exploration. If the contradiction handling scheme is re-thought, the evaluators for logic and constraints can be merged into one evaluator with one compile time flag that takes care of values. Time ordering and propagation delay may turn out to be important in constraint modeling. Since it turns out to be easy to add, it should be made available.

The routines in The Imaginary Appendix should be made to work. It is probably only a few months work to polish everything down to avl.c into working order. Naturally there will have to be some changes to take advantage of the time ordering addition to eval.

The section on real-time storage allocation suggested that LISP might be the most ideal environment for studying constraint languages. With the availablity of **sdb** that may not be completely true. The crucial debugging aids turned out to be the ability to single step one source line at a time, and the option of climbing into procedure calls for single stepping or executing them as one line.

The UNIX **make** program, with its ability to perform long sequences of compilations made re-compiling the system much easier. Once I defined the modules to be compiled in the **makefile**, two word commands did the rest. The makefile I used to generate **fccf, lsim, race** and this thesis is in Appendix M. Once I had one C program that did logic simulation, I could use the existing contents of the makefile to simplify the compilation of the new program.

Now that I have a feel for writing and debugging major systems in C, I might go on to

not only make the code in The Imaginary Appendix work, but hang a graphical front end off of it. I expect that such a front end would look very much like the one Borning built for Thinglab. One crucial difference I would make would be to use the UNIX device-independent plot routines to permit portability.

As a testimonial to the portability of the system, I ran the Farenheit to Celsius conversion program on both a VAX 11/780 and a PDP 11/45. Doing so, I found two tiny syntax errors that the Portable C compiler did not complain about, but that the UNIX Version 7 C compiler did not like. The mistake was mine. I violated the syntax, but **lint** did not catch it. The fix was made and the system has been run on the previously two systems, and a VAX 11/750 as well.

The final conclusion I make is that the hard work of defining bottom level data objects and making them work is done. Now begins the exciting work of modeling constraint systems and logic with the low level constructs. Ever since I got the first code to work, I have gotten progressively more curious about what next I could do with the system.

Appendix A
References

[Attardi 81]

Attardi, Giuseppi, and Simi, Maria. *Semantics and Inheritence and Attributions in the Descripiton System Omega,* AI Memo 642. M.I.T. Artificial Intelligence Laboratory (Cambridge, Massachusetts, August 1981).

[Borning 81]

Borning, Alan. *The Programming Language Aspects of Thinglab, A Constraint-Oriented* Simulation Laboratory ACM TOPLAS Vol. 3 No. 4 (Palo Alto, California, October 1981). Based on Ph. D. Thesis. Stanford University (March 1979). Also in paper of same name Xerox Palo Alto Research Center (Palo Alto, California, July 1979).

[de Kleer 78]

de Kleer, Johan, and Sussman, Gerald Jay. *Propagation of Constraints Applied to Circuit Synthesis,* AI Memo 485. M.I.T. Artificial Intelligence Laboratory (Cambridge, Massachusetts, September 1978). Also in *International Journal of Circuit Theory,* Vol. 8, No. 2, pp. 127-144, (April 1980).

[Koton 81]

Koton, Phyllis E. *Representing Constraint Systems with Omega* AI Working Paper 222. M.I.T. Artificial Intelligence Laboratory (Cambridge, Massachusetts, November 1981).

[Stallman 76]

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, Massachusetts, September 1976). Also in *Artificial Intelligence Journal,* Vol. 9, No. 2, pp. 135-196 (October 1977).

[Steele 80]

Steele, Guy Lewis. *The Definition and Implementation of a Computer Programming*

Language Based on Constraints. Ph. D. Thesis. Massachusetts Institute of Technology as AI Technical Report 595. M.I.T. Artificial Intelligence Laboratory (Cambridge, Massachusetts, August 1980). Also as VLSI Memo #80-32 (Cambridge, Massachusetts, 1980).

[Sussman 81]

Sussman, Gerald Jay, and Steele, Guy Lewis. *Constraints -- A Language for Expressing Alnost-Hierarchical Descriptions.* AI Memo 502A (Replaces AI Memos 433 and 502). M.I.T. Artificial Intelligence Laboratory (Cambridge, Massachusetts, August 1981). Also in *Artificial Intelligence Journal,* Vol 14, pp. 1-39, (1980)

[Sussman 75]

Sussman, Gerald Jay, and Stallman, Richard M. *Heuristic Techniques in Computer-Aided Circuit Analysis.* AI Memo 328. M.I.T. Artificial Intelligence Laboratory (Cambridge, Massachusetts, March 1975. Also in *IEEE Transactions on Circuits and Systems* Vol. CAS-22 (November 1975).

[Sutherland 63]

Sutherland, Ivan E. *SKETCHPAD: A Man-Machine Graphical Communication System.* Ph.D. Thesis, Massachusetts Institute of Technology as M.I.T. Lincoln Laboratory Technical Report 296 (Lexington, Massachusetts, January 1963).

[Williams 82]

Williams, Gregg. "Software Arts' TK Solver" *Byte* Vol. 7, No. 10 (Peterborough, New Hampshire, October 1982).

[Winston 79]

Winston, Patrick Henry. *Artificial Intelligence* (Reading Massachusetts, April 1979)

Appendix B

Implementation notes
This is a file of little things I notice as
I implement Constraints in C

9/29/82

I see no need for a difference between the definition of repository and node. Since the repository will contain a pointer to all the cells that it knows about, there is no real difference.

A point that seems important to remember in making the top level language is that it will have its biggest trouble talking about rules. Steele seperated primitive constraints from macros in that a primitive constraint would have rules written in lisp which would diddle the insides of a constraint. Perhaps it will be neccessary to define a constraint type RULE to make the neccessary unification.

Initially the data structures are being copied verbatim from the improved implementation given in the PhD thesis. A few minor changes are made on the fly to make things a little nicer for C.

Differences between the data structures used in this implementation and in Steele's. should be noted in this file!

Anything having to to with RULES will probably need major re-thinking.

I don't understand what the rule component in a Cell is for.

Names are integers which can be associated with printnames (character strings) when and where convenient.

NB If a cell is a pin of a constraint, the cell's owner is the constraint and the cell's name is the pin number.

In a constraint instance, the info field is, for lack of anything better now, a string of characters. This may change when the nature of the types of info stored becomes clearer.

The definition of rules is totally different. In the first trial implementation, there is a pointer to a C function that does the operation of the rule. If the rule is a macro, the function is one that is canned and the linkage is crocked by the top level language. (this should be made as efficient as possible since it will end up bing the rule rather than the exception.

In a rule, I don't understand how the id-bit works. It looks as if it can just bee a yes/no variable instead of an integer

It also occurs to me that values are currently though of as being integers exclusively. I should do what I can to make it possible for values to be expanded to more general things. Might int float and char[] be sufficient?

I now understand the rule portion of cell, it will name the rule responsible for the mess.

The rule and constraint mechanism was not general enough. I changed the deffinition of Constraint and rule so that rules could either be other constraints or primitive operators. I have also seperated internal and external state.

There are hooks installed in this mess so that Value is now of type int. To change the whole system to speak of other values, change the typedef and modify things.

New IDEA! instead of < generating> names for these objects, I will store as thier name, the address that they live in (from the storage allocator) instead. This way, instead of associating a name with a pointer to

the object I will use the pointer (guaranteed to be unique) as the name. (Yow! I've just invented gensym)

9/30/82

I now have insight into how to bring all the objects into existance. Each of the defind objects can be sucked out of malloc. Human readable names should be parsed and then allocated for.

I must remember to design routines particularly carefully because it is a ghastly error to use something that has been destroyed (freed). Perhaps linkages to things across other things should be eliminated if they're not used a lot and if they can be re-derived.

Perhaps nogood sets should be eliminated.

10/4/82

I see a way to make constraint instances more efficient: Instead of allocating cells and randomly accessing them, perhaps it would be better to CALLOC an array of cells and index into that array for the various variables in a constraint. The same might be true of rules. The question is: Would I ever wish to edit a constraint or would I rathe re-define it? If I wished to edit it, it would be best to alloc cells and rules one at a time.

BILL: IF YOU CAN SOMEHOW ENCAPSULATE THE POINTER TO THE ARRAY OF CELLS, THERE IS A STANDARD FUNCTION REALLOC WHICH WILL INCREASE THE SIZE OF A MALLOC'D OBJECT POSSIBLY BY MOVING IT (RETURNING A POINTER TO ITS NEW HOME AFTER COPY). -jfw

Until I figure out why it is neccessary to CREATE a new node for each constant, I will leave the supplier and repository fields of a newly created constant null, and not generate repositories for cells upon creation.

UH OH! I have a bit of a problem. When replacing variable indicies with pointers I threw out the baby with the bathwater. I need a representation that can VERY quickly find a rule given the "name" of a variable that just changed state so tha the necessary changes can be propagated. Sussman's very simple constraint propagator is helpful here. It has just connectors (nodes), constants, and constraints (which have no internal state)

I have just re-thought the deffinition of a constraint to access its cells in an array. Now I can calloc the whole mess of variables all at once, and I can send a message back to the constraint core telling which variable got bashed.

10/5/82

I have simplified the core data type: it does not seem to need a list of variables (pins) so I got rid of it.

I am also simplifying rules: Buckets of rules is an array of pointers to rules. Therefore Lists no longer need to point to Rules. Also it seems that rules are called under the assumption that they know what input variables they were called with. It is not clear that the output variable or variables need to be explicitly named.

Having a cell id that is a pointer to itself now appears to be a dumb idea! In fact all the id's that point to themselves are stupid! They have been removed. Certain of the cells and constraints have human readable names. That should be sufficient for the top level, and anything else should be pointed at by something that is pointed to by top level.

10/6/82

I simplified the nomenclature a bit: I call an instance of a constraint a Unit. I may decide later to call it a Part. Simple descriptive names seem preferable.

I have changed the definition of rules so that they will point to another rule. This enables rules to link to more elements in a bucket of rules. This simplifies the implementation of Lists since they no longer point at rules. It also means fewer calls to malloc to create another list element.

AH HA! I now see why a Core must record the number of variables in a constraint: When you create an instance of a constraint, Core tells you how many variables it has.

10/7/82

I now have a reason why a single cell needs a node, it was used when cells had a repository for a group of values, now they have their own values. I have chosen to change the procedures for finding values to know about cells without repositories rather than bring into existance repositories and then throw them away when the cells are merged

The mechanism for giving a value to a DUPE was kludgy in Steele's implementation! The contents of a cell was a pointer to the supplier rebel king. Instead I will just take its value directly. In the case of a slave cell I will indirect to the node supplier as did Steele. THis whole hierarchy of values might go away. I have chosen to keep it this long because it looks like a nice way to handle contradictions.

10/8/82

I have just noticed that things assume that cells, freshly created are puppets. I have added code to mk_cell to take care of this. (PUPPET defines the cell to have NO value). I have also added code to mk_unit to initialize all the cells to a puppet state.

I have defined data types, the routines to create empty objects from heap storage, and routines to determine if cells are bound, and to return their values. These routines should work for most any constraints implementation.

I have found out why a rule wants to know all the cells that trigger it: In hairy implementations, as a consequence of resolving contradictions, cells in a constraint unit that got their value from a contradiction need to be kicked. Any cell whose rule was triggered by this cell must have its value forgotten. YOW! that is kind of important. It seems that this mechanism should be made implicit in every constraint. I would have thought that that was the purpose of the forget rules. If forget consequences are implicit in the implementation at some level, I see no reason at all for forget rules. Steele never uses them either, and the definition mechanism does not accomodate them as far as I could see. I suspect that I should revise the data structure to eliminate forget rules and build an implicit consequence crusher into every constraint.

I have re-thought the way rules should work. Each rule will have an array called ins. This array of integers

names the indicies of pins in a Unit that the rule takes its input values from. There will also be an out integer which will name the index of the output cell. When a cell gets a new value it lands in a rule which will then check the cells on the ins array to see if they are bound to values. If not the rule stops. If so, a procedure will be called which gathers the values and produces an answer which will go into the cell indexed by out. When a value is forgotten, the rule will be run to check if the ins were bound. If so, the cell named in out is forgotten too. THIS IS HOW CONSEQUENCES SHOULD WORK! To this end, the forget rules array has been deleted.

I have decided to eliminate the no-good set system entirely. Instead I will build a more consistent assumption mechanism on later.

10/9/82

I made the storage allocator more robust by makeing every call to calloc and alloc signal mem_err if they return the NULL pointer. I should probably replace every instance of that mess with a C_calloc and C_malloc which would signal the error conditions.

I added the queue data type to unblock my mind on how to queue up a bunch of cells to update when one set of updates is done.

Perhaps I am being inordinately naughty, but I am removing the mechanism for forcing cells to be equal (the = = construct in Steele's implementation. A node will take care of things adequately and more simply and more generally. Something like the = = mechanism may have to be re-installed to deal with macro constraints which I have been doing MUCH thinking about lately.

It seems (while I am on the subject) that the primary difficulty in Macro constraints is in figuring out how to wire them together so that pimitive constraints can eventually get to the variables. The idea of creating and wiring up the macro constraint every time a new one is needed seems unneccessarily ugly, slow and big. The creation of a pure segment per deffinition, and then letting the constraint creation mechanism work normally seems best.

It seems that as a consequence of eliminating = = I also eliminate the need for dupes. So be it!

Convention: Changing a cell in a node causes all values to be rendered consistant by calling fix_node. Then all the cells whose values were changed are queued for awakening. If a node is unbound, a rule that fires on the corresponding cell is retracted. To do that we see if the rule made some cell KING, REBEL, or FRIEND. If so, and if this was the rule that ran, then we RETRACT! Otherwise we keep looking at rules. This means that it is too hard to keep the contra field of a node consistent. Therefore < POOF!> off it goes.

10/15/82

(It has been a long time since my last entry in the notes. Most of my time has been spent trying to implement the ideas I thought of. At last I have some notable thoughts)

I had previously thiught to zero out the rule of a PUPPET cell but I realize that when a cell loses a value it does so for a reason. I just cant take and forget the values of all the cells that depend on random sets of input variables. Henceforth, forget does not erase the rule field!

10/16/82

After thinking for several days and then talking to Pace for an hour and a half I have decided on a strategy to implement macro constraints. There will be a set of procedures that will know how deep into a macro one is while trying to run a rule. When presented with an index number, It will go through the necessary stack operations to eventually locate the actual cell. Macro constraints contain NO INTERNAL NODES!. I will need to create a doubly linked dynamically allocated list with pointers to top and bottom so that I can stack translation tables onto it. The ins field of a rule looks like a likely place to put a translation table. Perhaps I will add an entry for translation table and if it is not null, I am dealing with a primitive constraint. By using the magic procedures, I think it may be possible to make the differences between code that interprets primitive rules and macro rules, very small.

10/18/82

Here is the way a translation table works: A rule's translation table contains three things: A count of the number of pins in the lower level constraint, A list of low to high translations, and A list of high to low translations.

When a constraint rule is run is has to find out what pin of the lower level constraint to run. It calls t_run which offsets into the translation table at ttab+ttab[0]+pin+1. This mysterious offset means that element 0 of the translation table tells how man pins there are in the lower level constraint. Elements 1 through ttab[0] are translation entries for the lower level constraint. Elements ttab[ttab[0]+1] to the end are translation entries for the higher level constraint.

When a running constraint needs to access a pin it calls t_access to index into the list of low level pins to find out what top level cell it should access.

10/19/82

I have just about finished writing the recursive apply that will correctly evaluate nested constraints. Unfortunately there is one large problem in my implementation: Each cell records the rule that set its value. It does this for two reasons: 1) To make sure that if it is necessary to forget a value it will do so only for the right reason. 2) When the user asks WHY the system did something. The bug is that the only rules recorded are the lowest level rules. (They are the ones that actually set values.) In an implementation with a full-function why, the implementation will have to list lower and lower level rules in a list so that a hierarchy of why's can be expressed.

11/2/82

After a bit of a vacation and a lot of thought I have come up with some ideas on the parser. Firstly, at the suggestion of Evan and Pace, I will implement a nice scheme for storing symbolic names: Binary tree with supressed duplicates. All cells will have a name field that will point to a symbol in the tree. Additionally they will have an integer index which will be the pin number of the constraint they are attached to. (If they are, in fact, attached to one.) This will replace the idea that cells had either a name or an index. This will simplify matters for the parser since we can talk efficiently about the b of adder foo.

While we are talking about constraints, it is now obvious that each constraint unit should have a name and that each core should have a name. In the best implementation there should be no problem with having a global cell, and any number of different constraint types all having the same name. (The name string is

shared at a trade of space for more time spent doing lookups of several different things.) At this rate maybe an extensible hash table might end up being the right thing.

Concept: A network should be wired together incrementally and interactively by the user. Then when the constraint is wired together to the user's satisfaction he should issue a special command to wire a copy of his network with superfluous stuff squished out. This macro constraint will then be avaliable by type. Perhaps the user will have to be prompted for unknown things like the names of pins and the name of the new constraint type.

The parser should be simple: it should be able to CREATE instances of old data types like cells and primitive constraints (as units). It should be able to CONNECT things together. It should be able to DESTROY created objects and clean up after the disconnections it makes while doing so. It should DISCONNECT connected objects. It should DEFINE new macro constraints from the network. It should be able to EXPUNGE the entire class of UNITS and its associated macro constraint type. It should be able to SET and CLEAR the value of a cell. (It might be nice to warn the user if he is setting to a local cell rather than a global cell.)

The parser should help the user interogate the network by getting the VALUE of something. Perhaps some way to print the entire network would be useful. It should be able to FOLLOW a value back to its source so that the user knows how it came into existance. Perhaps a trace or wallpaper facility should be added so that long chains of constraint propagation can be recorded.

That is about all the parser should do! The rest comes from applying the network.

The primitive constraints that the intial implementation will have should be tiny but easily added to. There should be some thought given to a set of standard library routines that each newly written primop in C would rely upon. The first such routines that come to mind are the routines in frames.c that permit a cell to be found no matter how deeply it is buried in the net. There will have to be standard diciplines for asserting and evaluating cell values.

11/9/82

Changed constraints.h to account for printed names of cells and units.

Changed data.c in same way.

Same in eval.c.. (only change was in routine awaken).

11/16/82

Check on following bits of C syntax: Did I implement the table of dispatch routines correctly? Did I implement the array of keywords correctly? Did I apply sizeof correctly (all these things in the command routine.

After checking with John Woods, these things are NOW implemented correctly.

11/18/82

Three vital bits of housekeeping: 1) The wakeup queue and the contra queue must be created from main.

(This is now done.) 2) No function is yet defined that will take responsibility for dequing things from the wakeup queue and actually sending them to wake! 3) There is as yet nothing at all to deal with the contra queue! Although there is one routine that enqueues, that is all. For this implementation, it is probably best to have just a routine that names the offending cell and then starts up a command loop. The starting up of a command loop will be complicated by the fact that there may be a file open already. Probably a check to see if file is stdin and if not save the file pointer and temporarily set the file to stdio until the command loop ends.

The Ident data type has gone away because cells now contain both a symbolic name and an optional index number.

At last I think I understand how the symbol table should work: Global cells and constraint units should be accessible directly. There will be a unique name for every global cell. There may be many constraint units of different type of the same name. There will be more named things than either global symbols or constraint units. All names will go into the symbol table and increment the reference count. When at last every possible instance of a symbol is deleted, the reference count will go to zero and the symbol wil be chopped. This means that a node of the symbol table will have ONE pointer to a cell and a list of pointers to Units, and a reference count. This all means that there are basically two ways to access an object: directly from the symbol table if it is a global cell, or indirectly through a constraint unit if it is anything else. If we are looking for a cell local to a constraint unit, we first find if there is such a constraint unit. We then look at all the constraint units for one with a core whose name matches the type we want. Then we look at the array of cells for the one that matches the one we want.

11/23/82

I have implemented the search and insert routines for a symbol table employing AVL trees. I will probably have to throw it all away and use a linear search table when debugging time comes. I believe that I have screwed up the test for end condition in the balancing act within insert. Although AVL trees are a pretty data structure, I found that implementing them was quite hard. I greatly simplified my work by including back pointers. Now that I have actually written routines to insert, single rotate and double rotate, I see that there might be a simpler way that does not involve back pointers. But I believe I will need the back pointers for delete so... This all took a week to do, and it probably was not strictly worthwhile for the thesis as such, but it did provide an interesting side road to travel, and I had an opportunity to understand AVL trees from the inside.

11/24/82

I have implemented delete for avl trees. WOW was that all hard! THis mess will undoubtedly be impossible to debug. It seems that as I write the commands, I will have to write some nice interfaces for insertion and deletion of symbols that hide the details of my avl implementations. I know of some strangenesses that exist in the avl routines that require an external routine to handle symbol table stuff while it confines itself to the absolutely easiest problems in the avl stuff (which are hard!).

There should be routines for insertion of Global cells, units, other symbols, and there should be similar routines for deletion, and find.

11/25/82

THere is no reason for a find for other symbols. Insert symbol will do the right thing: If the symbol exists,

re-use the text. If it does not exist create it. The times when we want a symbolic name for anything that is not either a global cell or a Unit, we already have a pointer to the text from when we used insert to create it!

Just cleaned up a naming incinsistency: now lists, queues, alists (which should probably go away), and ulists have the convention that their structure elements are prefixed by []_ where [] is the first character of the name of the data type (in lower case).

Another naming inconsistency that still exist is that to make the next element in a list or a Ulist, mk_< thing> is called, but to make the next element in a queue gen_queue is called. This might want to be resolved

The interfaces between symbol table operations and the AVL data structure are now done. In the process I see an inconsistency in my use of pointer references within structures. I see I don't really know when to use fix this as soon as I learn when to use which.

11/26/82

UH OH! I just discovered that in addition to a list of units, I will need a list of cores because when a constraint unit is created it creates a unit of type < core> and damn it you have to find the core. Furthermore you have to be able to delete it from the table because the Expunge command relies on that.

No! Wait, there is no more than one of every type of thing. Therefore each core must have a unique name, just like every symbol must. I don't need a list, I need another entry like the one for global. I will just copy the code and make a few minor changes...

NOTE: remember to, at some time write a routine tolower (string) that will drop all the chars to lower case by side effect.

I have just noticed two other inconcistencies in my implementation: minor: I do not correctly use the syntax of structure references. I will have to go through the whole system and clean that up. major: I do not really know when to use a SLAVE state and a PUPPET state for a cell. Currently mk_cell will set the initial state to PUPPET, and mk_unit will set the current states to SLAVE. Perhaps something in wire.c should decide who is a puppet an who is a slave. My current rule of thumb is: If it is a global cell, it should be a PUPPET. If it is the cell of a constraint unit, it should be a SLAVE, until after its value gets set by the constraint rule or rules.

Yet another inconsistency: get_token assumes the string it is dumping into is allocated. Nearly every call to get_token assumes that get_token will allocate a string and return it filled. Somebody is wrong. It looks like MAXTOKEN should be defined in constraints.h and then everyone should define a char array of size MAXTOKEN when calling get_token.

11/27/82

Oh yes, I almost forgot one more command for the parser: the SHOW command which should give the value of any cell in the system. (the identity of the cell will be parsed with the same routine that connect uses.)

As an important addition, I will install some verbosity in the eval routines so that values will be printed as they change. This will require some kind of coercion from the Value data type to a printable string, but the SHOW command will need it too... oh well.

It seems a particularly nasty thing to try and enter the symbolic name of every symbol in a constraint unit. Therefore the appropriate thing to do is to have an array in the CORE which names the variables. This array is copied into the cell names in mk_unit but the reference count is NOT incremented. This will simplify the destruction process. The names of cells in a constraint unit are interned in the symbol table when the core is created. Welcome names as a new member of the structure Core.

11/28/82

I have built all that I want to build for now! There are hooks in the system for all kinds of things if I want to add them later. The low level routines if ever they get debugged will handle not only the creation commands but also the deletion commands if I add them. The FOLLOW command was not implemented because it seemed too hard to do all the right tree walking, and besides the rule information is not full enough here. The DEFINE command was also not implemented because it too would have to do too much tree walking, but the system should support it with few addtions.

Now all that remains are the definition of the bodies. For the constraint system I will implement ADDER, and MULTIPLIER For the logic system I will implement INVERTER NAND and DELAY. After these are implemented, debuggin begins.

For the distant future, an algebraic evaluator and canonicalizer, and loop eliminator should go before the main command loop. That is probably a big enough project for a whole thesis.

4/8/83

Now begins debugging

I noticed a conceptual bug in the way I was using get_token. I used the value it returned as the string size but once in a dozen calls to it. I did no bounds checking and there was MUCH potential for the user to type a token too long and have it blow up. I modified my calls to get_token and get_token itself so that it would take a bound as a parameter and it would not exceed that bound. I got rid of the returned value and changed the one instance it was used to need it no longer.

4/11/83

I decided that having bot an id bit and a flag word in a rule was redundand, silly and wasteful. I deleted the id bit and retained the more general flag bit.

4/13/83

In my attempt to explain how translation tables work, I have discovered a way to simplify them. Since each CONSTR Rule to be run will have its own translation table, the best thing to do is to have one number in the table that will name which pin of the lower level constraint CORE should have its Rules run, and an array which tells which Cells of the Unit correspond to the pins of the lower level constraint Core. To implement this I should change the data structure use as follows: For a PRIMOP: use ins for the list of input variables, and out for the output variable. (As before.) For a CONSTR: use ins for the list of translations, and out to tell which one of the lower level pins THIS Rule should transfer control to. This simplifies frames.c, and apply within eval.

4/15/83

I decided that Rules should be sharable. This means that the c_type field of the Rule should go away and that the rules field of the core should return to being a linked list. (it is a bit ugly, but I think it will save on space and complexity.) The compiler for these constraints will have to be REALLY smart, but that is another tub of fish.

4/20/83

I have decided to gather up the routines that free things into data.c and put them with the routines that allocate. Previously free was distributed thoughout the system, now there will be congute umk_thing to go with mk_thing procedures in data.c. This affects symtab, avl.c, and a few other places. I hope that the routines, gathered together will make the code more robust.

4/23/83

While writing the primitive operator for multiplication, I noticed the following problem: If a primitive operator needs to signal an internal failure such as division by zero, it is not clear what should happen. The evaluator was initial conceived to grab hold of the Cell that would take the output value, run the primop, stuff the value that the primop returned into the Cell, and then perform housekeeping like making the node consistent. I initially assumed that the evaluator could know in advance if a new value could be asserted. The way it would know was that the inputs to the primop would either all have values, or not. If not, the rule would not run and it would not assert a value. I just discovered that a t est for division by zero must occur while the rule is running. If there is a division by zero, the value should not be changed, but a contradiction should be signaled. The mechanism to do that is not well defined in view of the other assumptions I made while writing this system. The best thing seems to be to slightly modify eval.c to insert the following assumption: If the value returned by a primop is the same as the current value, and if the cell is KING, then let the existing housekeeping stuff decide if this rule should be the one to give the cell a value. If the cell is not KING, assume that the rule is signaling internal failure. If it is internal failure, the evaluator does no housekeeping, and leaves it up to the primop to decide on the action which would depend on the severity of the problem. The probable best action to take is to signal the problem to the standard error, and enqueue the cell for contradiction. This is what mult.c does.

This solution takes into consideration a lot of modularity considerations. Since a lot of system housekeeping is necessary, primops are still constrained to have one output pin and the system takes care of asserting the value on that pin. The possiblilty of letting the primop assert the value would only work if the evaluator wanted to remember the previous values of all the cells of the constraint, and then check to see which ones changed. This would be an interesting approach, but seems to do a lot of needless testing in the majority of cases, and only provide a marginally more abstract method of value assertion. It is vitally important to remember that primops should not take it upon themselves to modify the contents of any cells unless it is DESIRED that the evaluator not notice the change of state, and the primop actually intends to do the housekeeping itself. It is possible although difficult for the primop to get all the state information on the constraint's cells, but it is a duplication of function for it do any housekeeping. That is what the evaluator is for!

Appendix C
constraints.h

```
/* EMACS_MODES: c, !fill */

/* data structure definitions for Constraints */

/*
 * We know how to talk about Big objects:
 *      Unit    An instance of a constraint
 *      Core    The immutable part of a constraint
 *      Rule    Either a C function or another, lower level, Unit.
 *      Node    A collection of cells all hooked together
 *      Cell    The fundamental unit of constraint data
 * Little objects:
 *      List    An element in a linked list of pointers to cells
 *      Ulist   Linked list of pointers to units
 *      Rlist   Linked list of pointers to rules
 * Housekeeping Objects:
 *      Alist   Association list of symbolic names and addresses
 *      Queue   A queue of cells to wake up
 *
 * Pointers have been used in place of names to make things run fast, and
 * those things that want a human readable name have a character array
 * that contains the human readable name.
 */

/*
 * First we have to define true and false.  I thought they were in
 * stdio.h, but je me trompe...
 */

#define TRUE 1
#define FALSE 0
/*
 * Now a few defines to make meta-circular defintions of data types
 * to work in C.
 */

#define Unit struct unit
#define Core struct core
#define Rule struct rhule
#define Irule struct irhule          /* Initializable rule *ugh* */
#define Node struct node
#define Cell struct cell

/*
 * Now some cleanups to make lint happy and to keep our types virginal.
 */



/*
 * Now the actual defintions
 */
```

```
typedef struct liszt
{
        Cell *l_cell;
        struct liszt *l_next;
}List;

typedef struct ulist
{
        Unit *u_unit;
        struct ulist *u_next;                   /* The link to next */
}Ulist;

typedef struct rlist
{
        Rule *r_rule;
        struct rlist *r_next;                   /* The link to next */
}Rlist;


/*
 * A node is collection of cells one of whom supplies THE value for
 * the node.
 */

struct node
{
        List *cells;        /* Cells joined at this node */
        Cell *supplier;     /* The name of the cell supplying this node's value */
};

#ifdef LOGIC
typedef enum value
{
        F,
        T
} Value;
#else
typedef int Value;
#endif

/*
 * states for a cell
 * this lets us find out the state of a cell as fast as we can
 */

typedef enum state
{
        PUPPET,
        KING,
        FRIEND,
        SLAVE,
        REBEL
} State;
```

```
struct cell
{
        char *name;     /* Pointer to cell's name. */
        Value value;    /* The numerical value of the cell */
        State state;    /* state of the value */
        Node *repository;       /* Points back to repository */
        Unit *owner;    /* Points to unit or NULL if a constant */
        int index;      /* Array index if cell of a constraint. */
        Rule *rule;     /* Pointer to the rule that caused value */
        int mark;       /* Marks for tree walking */
};

struct core     /* constraint core or type */
{
        char *symbol;   /* name as given in top level expressions */
        int var_ct;     /* number of vars */
        char **names;   /* Indexed array of pointers to pin names */
        Rlist **rules;  /* Array of rule buckets */
};

struct unit     /* An actual instance of a constraint */
{
        char *pname;    /* Printed name of unit */
        Cell *vars;     /* Points to array of variables (use calloc) */
        Core *ctype;    /* Points back to core */
        int q_rules;    /* number of queued rules */
};

typedef enum rule_type
{
        PRIMOP,
        CONSTR,
} Rule_type;

typedef int R_flag;

struct rhule            /* Rule does its work by side effect UGH */
{
        Rule_type ty;   /* Tells what kind of rule this is */
        int *ins;/* Array tells cell numbers of inputs (-1 flags end) */
        int out; /* Cell number of Unit that this rule outputs to */
        int delay;      /* Time delay for this rule */
        R_flag flags;   /* flag bits */
        char *text;     /* Pointer to magical text that helps do this rule */
        union {
                Value (*primop)() ;     /* points to C funct. for this rule */
                Core *ragmop;  /* points to constraint for this rule */
        }op_ptr;
};

/*
 * Because it is impossible to initialize a Union Data type,
 * We create a bogus initializable rule type for primitive operators
 * that we can statically initialize.
 * Worse still, we have to cast an Irule to a Rule after we are
```

```
 * finished creating it.
 * ("That's no tit! Its a bogus cream tit with a cherry on top!"
 *                                  -- Silly Buggers)
 */

struct irhule               /* Rule does its work by side effect UGH */
{
        Rule_type ty;   /* Tells what kind of rule this is */
        int *ins;/* Array tells cell numbers of inputs (-1 flags end) */
        int out; /* Cell number of Unit that this rule outputs to */
        int delay;      /* Time delay for this rule */
        R_flag flags;   /* flag bits */
        char *text;     /* Pointer to magical text that helps do this rule */
        Value (*prymop)() ;     /* points to C funct. for this rule */
};

typedef enum type
{
        CELL,
        RULE,
        UNIT
}Thing;

typedef struct alist        /* Association list */
{
        Thing type;
        union {
                Cell *cptr;
                Rule *rptr;
                Unit *uptr;
        };
        struct alist *a_next;
}Alist;

/*
 * A simple dynamically allocated queue. A gift from John Woods
 * If this turns out to run too slowly, a better implementation
 * would alloc a bunch of links at once
 */

typedef struct q_link                   /* The insides of a queue */
{
        Cell *q_cell;                   /* The Cell enqueued */
        struct q_link *q_next;          /* The link to next */
}Q_link;

typedef struct queue                    /* An actual queue */
{
        Q_link *q_root;
        Q_link *q_last;
}Queue;

/*
 * A priority queue for time ordered events.  This is a linked list of
 * queues, and it uses the Queue data type extensively.
```

```
*/

typedef struct pqueue           /* The insides of a prio queue */
{
        Queue *pq_queue;
        int pq_offset;          /* Relative time offset from previous */
        struct pqueue *pq_next;
}Pqueue;

/*
 * Here are the declarations of our Runtime queues  Remember to create
 * them in main() before they are actually used
 */

extern Queue *wake;                     /* Central Queue of cells to awaken */
extern Queue *contra;
extern Pqueue *agenda;                  /* For logic simulation */
```

Appendix D
data.c

```
#include < stdio.h>
#include "constraints.h"

/*
 * EMACS_MODES: c, !fill
 * Constraint Language baby version  William D. Cattey
 *
 * Primitives to create and destroy instances of our datatypes
 * This code makes EXTENSIVE use of malloc.  Beware of ghastly errors!
 * fatal, mem_err, mk_sym, umk_sym,  mk_cell, mk_glob, is_const,
 * mk_list, umk_list, mk_ulist, mk_rlist, mk_node, umk_node, gen_vars,
 * gen_array, gen_names, mk_core, mk_unit, umk_unit, mk_rule, q_create,
 * enqueue, dequeue, pq_create, pq_add, pq_del, pq_top.
 */

char *malloc();
char *calloc();

/* This code will expand to clean up after a fatal error */

fatal () {
        abort(); /* dump core for sdb*/
}

/*
 * This code runs when the allocator can't.  Maybe something nicer
 * Will eventually replace it.
 */

mem_err (where)
char *where;
{
        fprintf(stderr, "Allocate operation failed in %s.0,where);
        fatal ();
}

/*
 * Allocate space for a symbolic name len chars long.
 */

char *mk_sym (len)
int len;
{
        char *c;
        len++;
        if ((c = calloc ((unsigned)len, sizeof (char))) == NULL) mem_err ("mk_sym");
        return (c);
}

/*
 * Free a symbol.  One procedure call less efficient, but more consolidated.
```

```
 * Beware of GHASTLY errors!  Use care when calling this.
 */

umk_sym(s)
char *s;
{
        free (s);
}

/*
 * Return an empty Cell. Initialize to have a state of PUPPET, no repository,
 * no rule and leave the value undefined.
 */

Cell *mk_cell () {
        Cell *c;
        if ((c = (Cell *)malloc (sizeof (Cell))) == NULL)
                mem_err ("mk_cell");
        c-> name = NULL;
        c-> state = PUPPET;
        c-> repository = NULL;
        c-> owner = NULL;
        c-> rule = NULL;
        c-> mark = 0;
        return (c);
}

/*
 * Return an empty list element with both pointers set to NULL
 */

List *mk_list ()
{
        List *l;
        if ((l = (List *)malloc (sizeof (List))) == NULL)
                mem_err("mk_list");
        l-> l_cell = NULL;
        l-> l_next = NULL;
        return (l);
}

/*
 * Destroy a list element.
 */

umk_list (l)
List *l;
{
        free ((char *)l);
}

/*
 * Create an empty element in the list of pointers to units.
 */
```

```
Ulist *mk_ulist () {
        Ulist *u;
        if ((u = (Ulist *)malloc (sizeof(Ulist))) == NULL)
                mem_err("mk_ulist");
        u-> u_unit = NULL;
        u-> u_next = NULL;
        return (u);
}

/*
 * Create an empty element in the list of pointers to rules.
 */

Rlist *mk_rlist () {
        Rlist *r;
        if ((r = (Rlist *)malloc (sizeof(Rlist))) == NULL)
                mem_err("mk_rlist");
        r-> r_rule = NULL;
        r-> r_next = NULL;
        return (r);
}

/*
 * Return an empty node all ready to have cells connected to it.
 */

Node *mk_node () {
        Node *c;
        if ((c = (Node *)malloc (sizeof (Node))) == NULL)
                mem_err("mk_node");
        c-> cells = NULL;               /* Initialize everything */
        c-> supplier = NULL;            /* to zero for an */
        return (c);
}

/*
 * Destroy a Node.  Please remember to strip off all the cells before
 * you destroy it!
 */

umk_node (n)
Node *n;
{
        free ((char *)n);
}

/*
 * Creating a constraint is hard.  There are arrays of cells to generate,
 * a core to create, and a unit to bind it all together for each instance.
 */

#define gen_vars(N) (Cell *)calloc((unsigned)N,sizeof(Cell))
#define gen_names(N) (char **)calloc((unsigned)N,sizeof(char *))
#define gen_array(N) (Rlist **)calloc((unsigned)N,sizeof(Rlist *))
```

```
/*
 * Make an empty core for n variables
 */

Core *mk_core (n)                       /* Make an empty core of n vars */
int n;                                  /* number of variables */
{
        Core *c;                        /* The core under construction */
        if ((c = (Core *)malloc(sizeof(Core))) == NULL)
                mem_err ("mk_core");
        c-> symbol = NULL;
        c-> var_ct = n;
        if ((c-> rules = gen_array(n)) == NULL) mem_err ("mk_core0");
        if ((c-> names = gen_names(n)) == NULL) mem_err ("mk_core1");
        return (c);
}

/*
 * Return an empty unit associated with a specific core.
 */

Unit *mk_unit (c)
Core *c;
{
        int i;
        Unit *u;
        Cell *t_access ();
        if ((u = (Unit *)malloc(sizeof(Unit))) == NULL) mem_err ("mk_unit");
        u-> ctype = c;          /* Point back */
        if ((u-> vars = gen_vars (c-> var_ct)) == NULL) mem_err("vars");
        for (i = 0; i < c-> var_ct; i++)
        {
                (u-> vars+i)-> state = SLAVE; /* all start as slave */
                (u-> vars+i)-> name = *(c-> names+i);
                (u-> vars+i)-> owner = u;               /* Tell them we own them */
                (u-> vars+i)-> index = i;               /* Tell them their index */
        }
        u-> pname = NULL;
        u-> q_rules = 0;
        return (u);
}

/*
 * Routine to unmake a created unit.  PLEASE do not call this unless
 * all references to the unit in question have been destroyed.
 */

umk_unit (u)
Unit * u;
{
        free ((char *)u);
}

/*
 * Rules are complicated.  Either they are primop rules which call a
```

```
 * C function, or they are a Constraint type rule which has a translation
 * table to tell which variables of the abstract constraint map into which
 * variables in the lower level operation.
 * This routine simply creates an empty rule with enough space to handle
 * inct variables.  Inct should equal the number of variables if this is to
 * be a PRIMOP rule, or the sum of the number of variables in the upper and
 * lower level functions if this is a constraint type rule.
 */

Rule *mk_rule (type, inct)
int inct;
Rule_type type;
{
        Rule *r;
        int *i;
        if ((r = (Rule *)malloc(sizeof(Rule))) == NULL) mem_err ("mk_rule");
        if ((i = (int *)calloc((unsigned)inct+1, (sizeof(int)))) == NULL)
                mem_err ("int");
        r-> ty = type;
        r-> ins = i;
        r-> out = -1;
        r-> delay = 0;
        r-> flags = 0;
        r-> text = NULL;
        r-> op_ptr.primop = NULL;
        r-> op_ptr.ragmop = NULL;
        *(r-> ins) = -1;          /* No input pins initially */
        return (r);
}

/*
 * Functions for Cell queues, to create enqueue and dequeue.
 */

Queue *q_create ()
{
        Queue *q;
        if ((q = (Queue *)calloc (1, sizeof(Queue))) == NULL) mem_err("q_create");
        return (q);
}

enqueue (q,cell)                  /* Put a cell on the queue */
Queue *q;
Cell *cell;
{
        Q_link *temp;
        fprintf (stderr, "Enqueueing ");
        print_cell (cell);
        fprintf (stderr, ".0);
        if ((temp = (Q_link *)calloc(1, sizeof (Q_link))) == NULL)
                mem_err ("enqueue");
        if (q-> q_root == NULL)
        {
                q-> q_root = q-> q_last = temp;
                q-> q_last-> q_cell = cell;
```

```
        }
        else
        {
                q-> q_last-> q_next = temp;
                q-> q_last = q-> q_last-> q_next;
                q-> q_last-> q_cell = cell;
        }
}

Cell *dequeue (q)                       /* Pull an item out of the queue */
Queue *q;
{
        Q_link *temp;
        Cell *c;
        if ((temp = q-> q_root) == NULL ) return (NULL);
        c = temp-> q_cell;
        fprintf (stderr, "Dequeueing ");
        print_cell (c);
        fprintf (stderr, ".0);
        q-> q_root = temp-> q_next;
        free ((char *)temp);
        return (c);
}

Pqueue *pq_create ()
{
        Pqueue *q;
        if ((q = (Pqueue *)malloc (sizeof(Pqueue))) == NULL) mem_err("q_create");
        q-> pq_offset = 0;
        q-> pq_queue = q_create ();
        q-> pq_next = NULL;
        return (q);
}


penqueue (q, c, o)
Pqueue *q;
Cell *c;
int o;
{
        Pqueue *this, *prev, *temp;
        if (q == NULL)
        {
                fprintf (stderr, "Queue dissappeared!0);
                fatal ();
        }
        if (o <  0)
        {
                fprintf (stderr, "Negative time???0);
                fatal();
        }
        fprintf (stderr, "For time offset %d : ", o);
        prev = q;
        this = q-> pq_next;
        while (this != NULL)
```

```
        {
                if (prev-> pq_offset == o || this-> pq_offset > o)
                        break;
                o -= prev-> pq_offset;
                prev = this;
                this = prev-> pq_next;
        }
        if (prev-> pq_offset == o)
        {
                enqueue (prev-> pq_queue, c);
                return;
        }
        if ((temp = (Pqueue *)malloc(sizeof (Pqueue))) == NULL)
                mem_err ("penqueue");
        temp-> pq_offset = o;
        temp-> pq_next = this;          /* might be NULL */
        temp-> pq_queue = q_create ();
        prev-> pq_next = temp;
        enqueue (temp-> pq_queue, c);
        if (this != NULL) this-> pq_offset -= o;
}

Cell *pdequeue (q)
Pqueue *q;
{
        Cell *c;
        Pqueue *temp;
        if (q == NULL)
        {
                fprintf (stderr, "Queue dissappeared!0);
                fatal ();
                return (NULL); /* keeps lint quiet */
        }
        if ((c = dequeue (q-> pq_queue)) == NULL)
        {
                temp = q-> pq_next;
                if (temp == NULL) return (NULL);
                q-> pq_queue = temp-> pq_queue;
                q-> pq_next = temp-> pq_next;
                free ((char *)temp);
                c = pdequeue (q);       /* in case middle queue empty */
        }
        return (c);
}
```

Appendix E
eval.c

```
#include < stdio.h>
#include "constraints.h"
#define MAX_NUM 14          /* Maximum chars in a number */

/*
 * This file Contains the functions for evaluating a constraint network.
 * We can diddle values here, awaken cells, render nodes consistent, run
 * rules, and do whatever housekeeping is neccessary.
 * getsup, boundp, print_val, node_val, cell_value,
 *  wake_slaves, sel_rule, duel,
 * fix_node, forget, apply, awaken, fix_contra.
 */

/*
 *  Remember to get t_access from frames.c or write one by hand.
 */

Cell *t_access();

/*
 * Return the cell who is the supplier of this cell's node.  If this
 * cell has no node, return this cell.
 */

Cell *getsup (c)
Cell *c;
{
        Node *n;
        if ((n = c-> repository) == NULL)
                return (c);
        else if (n-> supplier == NULL)
                return (c);
        else
                return (n-> supplier);
}

/*
 * Determine if this cell's node is bound.
 * First get the supplier of the node.  The funciton getsup
 * returns the cell itself if the node had no supplier.
 * If the supplier is king or puppet, the job is easy:
 * Return TRUE if KING, FALSE if PUPPET.  Fix_node will deal with
 * the case of this cell wanting to become the supplier of a supplier-
 * less node.  If the cell is a slave and there is no supplier,
 * we return FALSE.
 */

boundp (c)
Cell *c;
{
        State s;
```

```
        char *str;
        c = getsup (c);
        s = c-> state;
        switch (s) {
        case KING:
                return (TRUE);
        case PUPPET:
                return (FALSE);
        case FRIEND:
                str = "Friend";
                break;
        case SLAVE:
                if (c-> repository-> supplier == NULL)
                        return (FALSE);
        case REBEL:
                str = "Rebel";
                break;
        default:
                str = "Impossible";
        }
        fprintf (stderr, "The ");
        print_cell (c);
        fprintf(stderr, " is unbound in state %s.0, str);
        fatal ();
        return (FALSE);                 /* keeps lint quiet */
}

/*
 * Print out a value on the standard error.  This routine can get
 * arbitrarily complex as values get more complex.
 */

print_val (v)
Value v;
{
        fprintf (stderr, "%d",(int)v);
}

/*
 * Given a cell return the value of its node.
 */

Value node_val (cell)
Cell *cell;
{
        cell = getsup (cell);
        if (cell-> state == KING)
                return (cell-> value);
        else {
                fprintf(stderr,"Supplier for %s is not king!0,cell-> name);
                fatal ();
                return (0);        /* keeps lint quiet */
        }
}
```

```
/*
 * Return an appropriate cell value for this cell., Get a node value
 * if necessary.
 */

Value cell_val (cell)                   /* find a cell's value */
Cell *cell;
{
        State s;
        s = cell-> state;
        switch (s) {
        case KING:
        case FRIEND:
        case REBEL:
                return (cell-> value);
        case SLAVE:                     /* NEVER look at the value of a slave */
                return (node_val (cell)); /* Because I never clear it */
        case PUPPET:
                fprintf (stderr,"You tried to take the value of a PUPPET0);
                return (0);
        default:
                fprintf (stderr, "Cell in impossible state!0);
                fatal ();
                return (0);       /* keeps lint quiet */
        }
}

/*
 * When a node's king changes, all the slaves must be awakened to deal
 * with the new node value. This routine goes through a node and enqueues
 * all the slave cells onto the wake queue where awaken will eventually see
 * them. Changed values propagate through a network from the awakened cell as
 * follows: An awakened cell looks at the constraint that owns it and tries
 * to run all the rules it knows about. The rules look at nearby cells and
 * assert or retract values of nodes they output to. Each of these nodes
 * is cleaned up in turn, by setting an unambiguous king and queueing all
 * slave cells to be awakened. This order of propagation seems to be
 * the most robust.
 */

wake_slaves (node)
Node *node;
{
        List *l;
        Cell *me;
        l = node-> cells;
        while (l != NULL)
        {
                me = l-> l_cell;
                if (me-> state == SLAVE)
                        enqueue (wake, me);
                l = l-> l_next;
                me = l-> l_cell;
        }
```

```
}

/*
 * Given two rules r1 and r2 which are primops, return r1 if both lists
 * of ins are same length.  If r1's list is longer return r2 if its
 * list of ins  is a proper subset of r1's.
 */

Rule *sel_rule (r1, r2)
Rule *r1, *r2;
{
        int k, f, *kins, *fins;
        kins = r1-> ins;
        fins = r2-> ins;
        while (((f = *fins++) != -1) && ((k = *kins++) != -1))
                ;                               /* fly to the end of one list */
        if (k != -1) return (r1);  /* Implies fins > = kins */
        fins = r2-> ins;
        while ((f = *fins++) != -1)
        {
                kins = r1-> ins;
                while ((k = *kins++) != -1)
                {
                        if (f == k) break;
                }
                if (f != k) return (r1);
        }
        if (*fins == -1) return (r2);
        else return (r1);
}

/*
 * Given a friend and a king install the friend as king if he has a
 * value depending on a subset of the variables that gave the king his value.
 */

duel (kng, fnd)
Cell *kng, *fnd;
{
        Rule *k, *f, *a;
        if (kng-> repository-> supplier != kng)
                fprintf (stderr, "King not first in duel0);
        k = kng-> rule;
        f = fnd-> rule;
        a = sel_rule (k, f);
        if (a == k)
                fnd-> state = FRIEND;
        else {
                kng-> state = FRIEND;
                kng-> repository-> supplier = fnd;
        }
}

/*
 * Function to take a cell and make its node consistant.
```

```
* Each cell in the node is examined.  Its value and state are
* updated, then it is enqueued to have its owner awakened
* When called, cell is either a PUPPET if the value was just
* forgotten or a KING if a new value was just set.  This routine
* finds a new king if possible when the cell is a puppet, or deals
* with instaling the new KING.
*/

fix_node (c)
Cell *c;
{
        Node *n;
        State s;
        Cell *other;
        s = c-> state;
        n = c-> repository;
        other = n-> supplier;
        if (s == KING)          /* trying to set node value */
        {       /* If there is no supplier, make us it. */
                if (other == 0)
                {
                        n-> supplier = c;
                        wake_slaves (n);
                }
                /* If we are already supplier wake slaves with new value */
                else if(other == c) wake_slaves (n);
                /* If c supplier is PUPPET, make us supplier */
                else if (other-> state == PUPPET) {
                        other-> state = SLAVE;
                        n-> supplier = c;
                        wake_slaves (n);
                }
                /* If we are a friend of supplier select best KING */
                else if (other-> value == c-> value)
                        duel (other, c); /* Battle for Supplier hood */
                /* Otherwise we disagree and are among the REBELS */
                else {
                        c-> state = REBEL;
                        enqueue (contra, c);
                }
        }
        else {                          /* We must be a PUPPET */
                if (other == c) {       /* If we were KING */
                        List *l;
                        l = n-> cells;  /* Look for a FRIEND to KING */
                        while (l != NULL) {
                                other = l-> l_cell;
                                if (other-> state == FRIEND) {
                                        other-> state = KING;
                                        n-> supplier = other;
                                        return;
                                }
                                l = l-> l_next;
                                other = l-> l_cell;
                        }
```

```
                        l = n-> cells;   /* Look for a REBEL to KING */
                        while (l != NULL) {
                                other = l-> l_cell;
                                if (other-> state == REBEL)
                                {
                                        other-> state = KING;
                                        n-> supplier = other;
                                        wake_slaves(n);
                                        return;
                                }
                                l = l-> l_next;
                                other = l-> l_cell;
                        }
                        n-> supplier = c; /* No hope forget node value */
                        wake_slaves(n);
                }
                else
                fprintf(stderr, "You called fix_node to forget a slave0);
        }
}

/*
 * Forget the contents of a cell, taking into consideration
 * its old state.  This routine is called when a cell's value
 * was forgotten as a consequence a constraint being undone.
 */

forget (c)
Cell *c;
{
        State s;
        s = c-> state;
        switch (s) {
                case PUPPET:  /* No retraction needed */
                case SLAVE:   /* It CANT be our fault */
                        break;
                case KING:
                        c-> state = PUPPET;
                        if (c-> repository != NULL) fix_node (c);
                        break;
                case REBEL:
                        c-> state = SLAVE;
                        awaken (c);
                        break;
                case FRIEND:
                        c-> state = SLAVE;
                        break;
        }
}

/*
 * Recursive apply.  This routine relies on the magic contained in the
 * file frames.c to work.  It is imperative that, each time a new abstract
 * level is dug down into, the translataion table is pushed onto the translation
 * stack.
```

```
*/

apply (u, sup, rl)
Unit *u;
Cell *sup;
Rlist *rl;
{
        Rule *r;
        while (rl != NULL && (r = rl-> r_rule) != NULL)
        {
                if (r-> ty == CONSTR)
                {
                        Core *c;
                        Rlist *nr;              /* nr stands for new rules */
                        int nn, *table;         /* nn stands for new name */
                        table = r-> ins;        /* get ttab from Rule */
                        add_tab (table); /* Push onto stack of ttabs */
                        c = r-> op_ptr.ragmop; /* get core for lower level op */
                        nn = r-> out;           /* new name from Rule */
                        nr = *(c-> rules+nn); /* new rule */
                        apply (u, sup, nr);     /* Apply it all to lower level op */
                        clr_top();              /* Flush ttab when finished */
                }
                else
                {
                        Value v, (*p)();
                        Cell *ans;
                        int ivar, *inpt;
                        ans = t_access(r-> out, u);
                        if (sup-> state == PUPPET)
                        {
                                if (ans-> rule == r) forget (ans);
                        }
                        inpt = r-> ins;
                        /* First check to see if we can run rule */
                        while ((ivar = *inpt++) != -1)
                                if (!boundp (t_access(ivar, u))) goto next;
                        p = r-> op_ptr.primop;
                        v = (*p)(u);    /* NOTE p can look at ALL vars */
                        if ( boundp (ans) && node_val(ans) == v )
                         { /* avoid work */
                           if (ans-> state == KING)
                                ans-> rule = sel_rule (ans-> rule, r);
                           return;
                         }
                        ans-> value = v;
                        fprintf (stderr, "The ");
                        print_cell (ans);
                        fprintf (stderr, " now has value ");
                        print_val (v);
                        fprintf (stderr, ".0);
                        ans-> state = KING;
                        ans-> rule = r;
                        fix_node (ans);
                }
```

```
                next:           /* break out two levels */
                rl = rl-> r_next;
        }
}

/*
 * When a node's value has changed, each cell must be awakened to cause its
 * owner (a constraint unit) to run and generate (or retract) values
 * This routine assumes that the cell's value is already changed
 * and uses its current state to generate consequences
 */

awaken (cell)
Cell *cell;
{
        Cell *sup;
        Unit *u;
        Rlist *r;
        Core *c;
        int pin;
        if (cell-> repository == NULL) {
                fprintf(stderr,"Wake a lone cell?0);
                return;
        }
        if ((u = cell-> owner) == NULL) return; /* If a constant, no rule */
        c = u-> ctype;
        pin = cell-> index;
        r = *(c-> rules+pin);           /* The rules that fire on this cell */
        sup = getsup(cell);
        apply (u, sup, r);
}

/*
 * Print out a cell name as either a global cell or a constituent of
 * of a constraint box.
 */

print_cell (c)
Cell *c;
{
        if (c-> owner == NULL)
        {
                fprintf (stderr, "global cell %s",c-> name);
        }
        else
        {
                fprintf (stderr, "cell %s of %s %s",
                        c-> name, c-> owner-> ctype-> symbol, c-> owner-> pname);
        }
}

/*
 * This routine runs when there is a contradiction.  Unfortunately, it
 * doesn't do much of anything yet.  It should MAKE the user do something
 * about the contradiction, but alas it just warns of the existance of
```

```
 * a contradiction, and then forgets about it.
 */

fix_contra (c)
Cell *c;
{
        fprintf (stderr, "The ");
        print_cell (c);
        fprintf (stderr, " has value: ");
        print_val (c-> value);
        fprintf (stderr, " which contradicts.0);
        fprintf (stderr, " with node supplier  the ");
        print_cell (c-> repository-> supplier);
        fprintf (stderr, " which has value: ");
        print_val (c-> repository-> supplier-> value);
        fprintf (stderr, ".0);
}
```

Appendix F
frames.c

```
#include < stdio.h>
#include "constraints.h"

/*
 * This file attempts to gracefully implement nested constraints
 * by creating a stack of translation tables and  generating a pointer
 * to a cell from a variable index number.  These routines must be informed
 * where in the nested structure we are at all times.
 */

char *malloc();          /* To make lint happy we declare malloc */

typedef struct ttpt
{
        int *ttab;
        struct ttpt *next_t;
} Ttpt;

Ttpt *mk_ttab ()
{
        Ttpt *t;
        t = (Ttpt *)malloc (sizeof (Ttpt));
        if (t == NULL)
        {
                mem_err("mk_ttab");
                return ((Ttpt *) NULL);
        }
        else return (t);
}

static Ttpt *world = NULL;

add_tab (table)
int *table;
{
        Ttpt *env;
        env = mk_ttab ();
        env-> ttab = table;
        if (world == NULL)
        {
                world = env;
                env-> next_t = NULL;
        }
        else
        {
                env-> next_t = world;
                world = env;
        }
}

clr_top ()
```

```
{
        Ttpt *t;
        if ((t = world) == NULL){
                fprintf(stderr, 'Flush nonexistant translation table?0);
                return;
        }
        world = t-> next_t;
        free ((char*)t);
}

Cell *t_access (name, u)
int name;
Unit *u;
{
        Ttpt *ptrs;
        int *table;
        ptrs = world;
        while (ptrs != NULL)
        {
                table = ptrs-> ttab;
                name = *(table + name);
                ptrs = ptrs-> next_t;
        }
        return (u-> vars+name);
}
```

Appendix G
plus.c

```
#include < stdio.h>
#include "constraints.h"
#include "primop.h"

/*
 * Hardwired plus constraint
 */

Cell *t_access ();
Value node_val();

/*
 * ns is the indexed array of pointers to names of pins
 */

char *plus_ns [] = { "a", "b", "c" };

/*
 * These three functions do the actual work.  They call t_access to
 * grab Cells and then mung on their values
 * Three unidirectional functions are created and called when they
 * have data values to create bidirectional addition/subtraction
 * duality.
 */

Value p_f_0 (u)                 /* a = c - b */
Unit *u;
{
        Cell *p, *q;
        p = t_access (2, u);    /* p becomes c */
        q = t_access (1, u);    /* q becomes b */
        return ((Value)  ((int)node_val(p) - (int)node_val(q)));
}

Value p_f_1 (u)                 /* b = c - a */
Unit *u;
{
        Cell *p, *q;
        p = t_access (2, u);    /* p becomes c */
        q = t_access (0, u);    /* q becomes a */
        return ((Value)  ((int)node_val(p) - (int)node_val(q)));
}


Value p_f_2 (u)                 /* c = a + b */
Unit *u;
{
        Cell *p, *q;
        p = t_access (0, u);    /* p become a */
        q = t_access (1, u);    /* q becomes b */
        return ((Value)  ((int)node_val(p) + (int)node_val(q)));
```

```
}

/*
 * These are the three arrays of ins for the three rules.
 */

int p_i_0[] = { 2, 1, -1}; /* Pins 2, and 1 must have values. */
int p_i_1[] = { 2, 0, -1}; /* Pins 2 and 0 must have values. */
int p_i_2[] = { 0, 1, -1}; /* Pins 0 and 1 must have values. */

/*
 * Now the actual rules for pins 0, 1, and 2.
 */

Irule p_r_0 =
{
        PRIMOP,                 /* ty */
        p_i_0,                  /* ins */
        0,                      /* out */
        0,                      /* delay */
        0,                      /* flags */
        NULL,                   /* text */
        p_f_0                   /* primop */
};

Irule p_r_1 =
{
        PRIMOP,
        p_i_1,
        1,
        0,
        0,
        NULL,
        p_f_1
};

Irule p_r_2 =
{
        PRIMOP,
        p_i_2,
        2,
        0,
        0,
        NULL,
        p_f_2
};

/*
 * Now, by hand, without forward references we create the
 * lists of rules complete with their pointers to next for
 * pins 0, 1, and 2 respectively.
 * Each pin has two rules.  Depending on which if either
 * of the remaining two pins have values, each pin may run
 * and assert a value on one or the other remaining pins.
 */
```

```
Rlist p_l_0a = {(Rule *)&p_r_2, NULL };
Rlist p_l_0 = {(Rule *)&p_r_1, &p_l_0a };

Rlist p_l_1a = {(Rule  *)&p_r_0, NULL };
Rlist p_l_1 = {(Rule *)&p_r_2, &p_l_1a };

Rlist p_l_2a = {(Rule *)&p_r_1, NULL };
Rlist p_l_2  = {(Rule *)&p_r_0, &p_l_2a };

Rlist *plus_rs [] = {&p_l_0, &p_l_1, &p_l_2};

/*
 * Finally, the core!  The unit is created in real time
 * mk_unit.
 */

Core p_core =
{
        "plus",
        3,
        plus_ns,
        plus_rs
};

Core *plus = &p_core;
```

Appendix H
mult.c

```
#include < stdio.h>
#include "constraints.h"
#include "primop.h"

/*
 * Hardwired multiply constraint
 */

Cell *t_access();

/*
 * ns is the indexed array of pointers to names of pins
 */

char *mult_ns [] = { "a", "b", "c" };

/*
 * These three functions do the actual work.  They call t_access to
 * grab Cells and then mung on their values
 * Three unidirectional functions are created and called when they
 * have data values to create bidirectional multiplication/division
 * duality.
 */

Value m_f_0 (u)                 /* a = c / b */
Unit *u;
{
        Cell *a, *b, *c;
        a = t_access (0, u);
        b = t_access (1, u);
        c = t_access (2, u);
        if ((int)node_val(b) == 0)      /* zero divide test */
        {
                fprintf (stderr,
                        "Attempt to divide by zero, value unchanged.0);
                enqueue (contra, b);
                return (node_val(a));
        }
        return ((Value)  ((int)node_val(c) / (int)node_val(b)));
}

Value m_f_1 (u)                 /* b = c / a */
Unit *u;
{
        Cell *a, *b, *c;
        a = t_access (0, u);
        b = t_access (1, u);
        c = t_access (2, u);
        if ((int)node_val(a) == 0)      /* zero divide test */
        {
                fprintf (stderr,
```

```
                    "Attempt to divide by zero, value unchanged.0);
                enqueue (contra, a);
                return (node_val(b));
        }
        return ((Value)  ((int)node_val(c) / (int)node_val(a)));
}


Value m_f_2 (u)                 /* c = a * b */
Unit *u;
{
        Cell *p, *q;
        p = t_access (0, u);    /* p become a */
        q = t_access (1, u);    /* q becomes b */
        return ((Value)  ((int)node_val(p) * (int)node_val(q)));
}

/*
 * These are the three arrays of ins for the three rules.
 */

int m_i_0[] = { 2, 1, -1};/* Pins 2, and 1 must have values. */
int m_i_1[] = { 2, 0, -1};/* Pins 2 and 0 must have values. */
int m_i_2[] = { 0, 1, -1};/* Pins 0 and 1 must have values. */

/*
 * Now the actual rules for pins 0, 1, and 2.
 */

Irule m_r_0 =
{
        PRIMOP,                 /* ty */
        m_i_0,                  /* ins */
        0,                      /* out */
        0,                      /* delay */
        0,                      /* flags */
        NULL,                   /* text */
        m_f_0                   /* primop */
};

Irule m_r_1 =
{
        PRIMOP,
        m_i_1,
        1,
        0,
        0,
        NULL,
        m_f_1
};

Irule m_r_2 =
{
        PRIMOP,
        m_i_2,
```

```
        2,
        0,
        0,
        NULL,
        m_f_2
};

/*
 * Now, by hand, without forward references we create the
 * lists of rules complete with their pointers to next for
 * pins 0, 1, and 2 respectively.
 * Each pin has two rules.  Depending on which if either
 * of the remaining two pins have values, each pin may run
 * and assert a value on one or the other remaining pins.
 */

Rlist m_l_0a = {(Rule *)&m_r_2, NULL };
Rlist m_l_0 = {(Rule *)&m_r_1, &m_l_0a };

Rlist m_l_1a = {(Rule *)&m_r_0, NULL };
Rlist m_l_1 = {(Rule *)&m_r_2, &m_l_1a };

Rlist m_l_2a = {(Rule *)&m_r_1, NULL };
Rlist m_l_2  = {(Rule *)&m_r_0, &m_l_2a };

Rlist *mult_rs [] = {&m_l_0, &m_l_1, &m_l_2};

/*
 * Finally, the core!  The unit is created in real time
 * mk_unit.
 */

Core m_core =
{
        "mult",
        3,
        mult_ns,
        mult_rs
};

Core *mult = &m_core;
```

Appendix I
The Imaginary Appendix
main.c

```
#include < stdio.h>
#include "constraints.h"
#define BUFSIZE 512
#define MAX_COM 20

FILE *file;                          /* global definition of file */
Queue wake, contra;

/*
 * In case there is no other listing of what files are neccessary,
 * this comment should tell!  The constraint system consists of:
 * constraints.h  Definitions of abstract data types
 * data.c         Routines for creating objects
 * frames.c       Routines for maintaining translation tables
 * avl.c          Tree that symbol table lives in
 * symtab.c       Routines for maintaining symbol table
 * wire.c         Routines to wire together networks
 * main.c         Startup and command parsing
 * eval.c         Propagates values
 * commands.c     Actual commands
 * bodies.c       Definitions of primitive constraints
 */


/*
 * This is the actual MAIN!
 * If arguments are specified it is assumed that the input will
 * come from these files before the tty driver loop is started.
 * Command is called to do the actual work of reading input and
 * then parsing.  Command dispatches to actual operations.
 */

main (argc, argv)
int argc;
char *argv[];
{
        extern FILE *file;
        extern Queue wake, contra;
        wake = q_create ();
        contra = q_create ();
        int i;
        while (--argc > 0 && (*++argv) != NULL)
        {
                if ((file = fopen (*argv, "r")) == NULL)
                        fprintf (stderr,
                        "Cannot open file %s, continuing.0,
                        *argv);
                command ();
                fclose (file);
        }
```

```
        file = stdin;
        command ();
}


/*
 * Parser:  Gets handed a buffer full of characters to act on.  It tokenizes
 * them and then when it gets a whole token it sends it out to be munged upon.
 */

/*
 * Routine to get characters from the input file and assemble a token
 * string in s. If EOF is the first character read in a token an empty s
 * is returned.  (If EOF occurs in the middle of the token, we must rely on
 * the library to give a second EOF.
 * This routine relies on the caller to allocate space for the token string
 * and to specify the maximum length string to fit in the array.
 */

get_token (s, max)
int max;
char s[];
{
        extern FILE *file;
        int c, i;
        while (blank (c = getc(file) ))
                ;
        if (c == EOF) {
                s[0] = ' ';
                return (0);
        }
        if (special (c)) {
                s[0] = c;
                s[1] = 0;
                return (1);
        }
        i = 0;
        while (c != EOF && i != max)
        {
                s[i++] = c;
                if (blank(c)) break;
                else if (special (c)) break;
                c = getc(file);
        }
        if (c != EOF) ungetc (c, file);
        s[i] = ' ';
        return ();
}

blank (c)
char c;
{
        switch (c) {
        case ' ':
        case '':
```

```
        case '0: return (1);
        default: return (0);
}

special (c)
char c;
{
        switch (c) {
        case '(':
        case ')':
        case ';': return (1);
        default: return (0);
}

static char *keys[] = {
        "create",
        "destroy",
        "connect",
        "disconnect",
        "define",
        "expunge",
        "set",
        "clear",
        "show",
        "follow"
};

extern int com_create (),
        com_destroy(),
        com_connect(),
        com_disconnect(),
        com_define(),
        com_expunge(),
        com_set(),
        com_clear(),
        com_show(),
        com_follow();

static int (*code)()[] = {
        com_create,
        com_destroy,
        com_connect,
        com_disconnect,
        com_define,
        com_expunge,
        com_set,
        com_clear,
        com_show,
        com_follow
};

command ();
{
        char com[MAX_COM];
        int (*doit)() = NULL;
```

```
        get_token (com, MAX_COM);
        while (com[0] !=  )
        {
                tolower (com);
                for (i = 0; i <  sizeof (keys); i++)
                {
                        if (strcmp (com, keys[i]) == 0)
                                doit = code [i];
                }
                if (doit == NULL)
                        fprintf (stderr, "I don't know how to %s.0,com);
                else
                {
                        Cell *sleepy;
                        (*doit)();
                        /* Actually awaken queued cells! */
                        while ((sleepy = dequeue (wake)) != NULL)
                                awaken (sleepy);
                        while ((sleepy = dequeue (contra)) != NULL)
                                fix_contra (sleepy);
                }
        }
}
```

```
#include < stdio.h>
#include "constraints.h"
#define MAX_SYM 512

/*
 * Commands.  The file of commands that do things.  These routines
 * are dispatched to from the parser and do a lot of hairy calling in
 * symtab.c, data.c and wire.c
 * com_create, (com_destroy), parse_cell, com_connect, (com_disconnect),
 * com_set, com_clear, com_show, (com_define), (com_expunge), (com_follow).
 */


/*
 * Create either a cell or a unit of the specified type.
 * The syntax parsed here is:
 * "create cell [name]" to create a global cell and
 * "create < core>  [name]" to create an instance of a constraint.
 */

com_create ()
{
        char type[MAX_SYM];
        get_token (type, MAX_SYM);
        if (strcmp (type, "cell") = = 0)
        {
                Cell *c
                c = mk_cell ();
                c-> name = ins_cell (type, c);
        }
        else
        {
                Unit *u;
                Core *c;
                c = find_core (type);
                if (c = = NULL)
                {
                        fprintf(stderr, "Can't find core %s0,type);
                        return;
                }
                get_token (type, MAX_SYM);
                u = mk_unit (c);
                u-> pname = ins_unit (type, u);
        }
}

/*
 * Destroy a created element.  Remember to meticulously clean up all the
 * pointers.
 */

com_destroy();
{
```

```
        fprintf (stderr, "Destroy is not yet implemented, sorry.0);
}

/*
 * Parse an < ident> (see below) into an actual cell. The only hair comes when
 * a cell is affiliated with constraint unit. The only way to get hold of
 * id is to look at all the constraint units until one is found with the
 * right core. Then a check through all its cells are made to see if one
 * exists of the right name.
 */

Cell *parse_cell ();
{
        Cell *c;
        char nam[MAX_SYM];
        get_token (nam, MAX_SYM);
        if (strcmp (nam, "cell") == 0)
        {
                get_token (nam, MAX_SYM);
                c = find_cell (nam);
                if (c == NULL)
                        fprintf(stderr, "Can't find global %s.0,nam);
                return (c);
        }
        else if (strcmp (nam, "the") == 0)
        /* Takes advantage of re-use of symbols (they are eq and equal!) */
        {
                Unit *u;
                Core *co = NULL;
                char *core_sym, cell_sym;
                Ulist *ul;
                get_token (nam, MAX_SYM);               /* Name of Cell we want */
                cell_sym = find_sym (nam);
                if (cell_sym == NULL)
                {
                        fprintf(stderr, "No such Cell as %s.0,nam);
                        return (NULL);
                }
                get_token (nam, MAX_SYM);    /* Keyword 'of' */
                if (strcmp (nam, "of") != 0)
                {
                        fprintf(stderr, "I expected an 'of' not '%s'.0,nam);
                        return (NULL);
                }
                get_token (nam, MAX_SYM);    /* Name of Core we want */
                core_sym = find_sym (nam);
                if (core_sym == NULL)
                {
                        fprintf(stderr, "No such type of unit as %s.0,nam);
                        return (NULL);
                }
                get_token (nam, MAX_SYM);    /* Name of the unit we want */
                ul = find_unit (nam);
                while (ul != NULL)
                {
```

```
                        if (ul-> u_unit-> ctype-> symbol == core_sym)
                        {
                                u = ul-> u_unit;
                                co = u-> ctype;
                        ul = ul-> u_next;
                }
                if (co == NULL)
                {
                        fprintf(stderr, "Can't find core %s.0,core_sym);
                        return (NULL);
                }
                int i;
                for (i = 0; i < = co-> var_ct; i++)
                {
                        if (co-> names+i == cell_sym)
                        {
                                c = u-> vars+i
                                return (c);
                        }
                }
                fprintf (stderr, "No cell %s in unit %s.0, cell_sym, nam);
        }
        else
        {
                fprintf (stderr, "I expected 'the' not %s.0,nam);
                return (NULL);
        }
}


/*
 * Connect two cells at a node.
 * The syntax parsed here is:
 * "connect < ident> to < ident> " where < ident> is:
 * "cell [name]" or "the [name] of < core> < unit> ".
 * in the latter case, because of the data structures involved,
 * one must find the list of units and find one whose core has the
 * right name. All units point at their cores, but no core points
 * at all its units. The core is a VERY static structure.
 */

com_connect ()
{
        Cell *c1, *c2;
        char gubble[MAX_SYM];
        c1 = parse_cell ();
        if (c1 == NULL)
        {
                fprintf (stderr, "Failed to parse first cell. Aborting.0);
                return;
        }
        get_token(gubble, MAX_SYM);
        if (strcmp (gubble, "to") != 0)
        {
                fprintf (stderr, "I expected 'to' not %s.0,gubble);
```

```
                return;
        }
        c2 = parse_cell ();
        if (c2 == NULL)
        {
                fprintf (stderr, "Failed to parse second cell. Aborting.0);
                return;
        }
        if (c1-> repository == NULL && c2-> repository == NULL)
        {
                Node *n;
                State s;
                n = mk_node ();
                /* Kludge cell c1 on as a supplier of the node and then */
                /* attach cell c2 using attach */
                s = c1-> state;
                case (s) {
                case SLAVE:
                        c1-> state = PUPPET;
                        break;
                case FRIEND:
                        c1-> state = KING;
                        break;
                case REBEL:
                        c1-> state = KING;
                        break;
                }
                n-> supplier = c1;
                n-> cells = mk_list ();
                n-> cells-> l_cell = c1;
                c1-> repository = n;
                attach (c2, n);
        }
        else if (c1-> repository == NULL && c2-> repository != NULL)
        {
                attach (c1, c2-> repository);
        }
        else if (c1-> repository != NULL && c2-> repository == NULL)
        {
                attach (c2, c1-> repository);
        }
        else
        {
                merge (c1-> repository, c2-> repository);
        }
}

/*
 * Disconnect a cell from a node.  Be careful to clean up any mess made.
 */

com_disconnect ()
{
        fprintf (stderr, "Disconnect is not yet implemented, sorry.0);
}
```

```
/*
 * Set the value of a global cell.  There is no use dealing with
 * all the possible contradictions of setting local cells.
 */

com_set ()
{
        Value v;
        Cell *c;
        char glob[MAX_SYM];
        get_token(glob, MAX_SYM);
        v = parse_val ();               /* Arbitrarily complex parser */
        c = find_cell (glob);
        if (c == NULL)
        {
                fprintf (stderr, "Can't set nonexistant cell %s.0,glob);
                return;
        }
        c-> state = KING;
        c-> val = v;
        if (c-> repository != NULL)
                fix_node (c);
        fprintf (stderr, "Set value of cell %s to ",glob);
        print_val (v);
        fprintf (stderr, ".0);
}


/*
 * Clear the value of a global cell.  This makes the cell in question
 * a puppet and then lets the node decide what to do next.
 */

com_clear ()
{
        char glob[MAX_SYM];
        cell *c;
        get_token (glob, MAX_SYM);
        c = find_cell (glob);
        if (c == NULL)
        {
                fprintf (stderr, "Can't clear nonexistent cell %s.0,glob);
                return;
        }
        forget (c);                     /* call routine in eval! */
        fprintf (stderr, "Cleared value of cell %s.0,glob);
}

/*
 * Show the value of a cell.  This routine may get mor complicated as
 * the Value data type gets more complicated.
 * The syntax parsed is "show < ident> " where < ident>  is the same as
 * for connect.
 */
```

```
com_show ()
{
        Cell *c;
        c = parse_cell ();
        fprintf (stderr, "The value of cell %s is: ",c-> name);
        print_val (c-> value);
        fprintf (stderr, ".0);
}

/*
 * Parse a value from the input stream.  Currently things are
 * mindlessly simple since the only kinds of values are integers!
 * This routine can easily be replaced with something hairy in the
 * future
 */

Value parse_val ()
{
        char num[MAX_NUM];
        get_token(num, MAX_NUM);
        return ((Value)atoi (num));
}

/*
 * Follow back a line of reasoning that gave this cell its value.
 */

com_follow()
{
        fprintf (stderr, "Follow not yet implemented, sorry.0);
}
```

```
#include < stdio.h>
#include "constraints.h"

/*
 * This file contains routines for wiring together networks
 * from nodes and cells and constraint units.
 * It also knows how to convert a network into a macro constraint.
 * attach, merge
 */

/*
 * Attach a cell to a node making sure that the node is consistent
 * after the attach is complete.
 */

attach (c,n)
Cell *c;
Node *n;
{
        State s;
        List *l;
        if (c-> repository != NULL) {
                fprintf(stderr, "Tried to attach instead of merge. Merging.0);
                merge (c-> repository, n);
        }
        s = c-> state;
        c-> repository = n;
        l = mk_list ();
        l-> l_cell = c;
        l-> l_next = n-> cells;
        n-> cells = l;
        case (s) {
        case SLAVE:
                if (n-> supplier-> state == KING) awaken(c);
                break;
        case REBEL:
        case FRIEND:
                c-> state = KING;
                fix_node (c);
                break;
        case KING:
                fix_node (c);
                break;
        case PUPPET:
                c-> state = SLAVE;
                if (n-> supplier-> state == KING)       awaken(c);
                break;
        }
}

/*
 * Routine to merge two nodes.  Calls attach on cells that it pulls
 * one at a time off of a node.
```

```
*/

merge (n1, n2)
Node *n1, n2;
{
        List l;
        /* Clear repositories to prevent wakeups if we have any circularities */
        l = n2-> cells;
        while (l != NULL) {
                l-> cptr-> repository = NULL;
                l = l-> next;
        }
        l = n2-> cells;
        while (l != NULL) {
                attach (l-> cptr, n1);
                n2-> cells = l-> next;
                umk_list (l);           /* BEWARE l is FREED */
                l = n2-> cells;
        }
        umk_node (n2);                  /* BEWARE n2 is FREED */
}
```

```
#include < stdio.h>
#include "constraints.h"

/*
 * Routines to do symbol table management. Requires some underlying
 * symbol table structure like a linear list or an avl tree.
 * (Designed with program avl.c in mind.)
 * Contains procedures ins_sym, ins_cell, ins_core, ins_unit, cell_find,
 * core_find, find_unit, del_sym, del_cell, del_core, del_unit.
 * Find mk_sym and umk_sym in data.c.


/*
 * Install a symbol in table for a random symbol and increment the
 * reference count. Return a pointer to the symbol string.
 */

char *ins_sym (key)
char *key;
{
        extern Avl *symtab;
        Avl *this;
        this = find (key);
        if (this == NULL)
        {
                symtab = mk_avl();
                this = symtab;
                this-> sym = mk_sym (strlen (key));
                strcpy (this-> sym, key);
        }
        if (strcmp (this-> sym, key) != 0)
        {
                insert (this, key);
                this = (this-> sym < key) ? this-> left : this-> right;
        }
        this-> ref_ct++;
        return (this-> sym);
}

/*
 * Insert a global cell in the table whose symbolic name is key.
 * Annoy the user if the cell's name is being redefined. Annoy the
 * user if the same cell is being interned twice. Return a pointer
 * to the symbol string.
 */

char *ins_cell (key, c)
char* key;
Cell *c;
{
        extern Avl *symtab;
        Avl *this;
        this = find (key);
```

```
        if (this == NULL)
        {
                symtab = mk_avl();
                this = symtab;
                this-> sym = mk_sym (strlen (key));
                strcpy (this-> sym, key);
        }
        if (strcmp (this-> sym, key) == 0)
        {
                if (this-> global == c)
                        fprintf (stderr, "Redundantly defining %s0, key);
                else if (this-> glob != NULL)
                        fprintf (stderr, "RE-defining active cell %s0, c);
        }
        else
        {
                insert (this, key);
                this = (this-> sym < key) ? this-> left : this-> right;
        }
        this-> global = c;
        this-> ref_ct++;
        return (this-> sym);
}

/*
 * Insert a core pointer in the table whose symbolic name is key.
 * Annoy the user if the core's name is being redefined.  Annoy the
 * user if the same core is being interned twice.  Return a pointer
 * to the symbol string.
 */

char *ins_core (key, c)
char* key;
Core *c;
{
        extern Avl *symtab;
        Avl *this;
        this = find (key);
        if (this == NULL)
        {
                symtab = mk_avl();
                this = symtab;
                this-> sym = mk_sym (strlen (key));
                strcpy (this-> sym, key);
        }
        if (strcmp (this-> sym, key) == 0)
        {
                if (this-> core == c)
                        fprintf (stderr, "Redundantly defining %s0, key);
                else if (this-> core != NULL)
                        fprintf (stderr, "RE-defining active core %s0, c);
        }
        else
        {
                insert (this, key);
```

```
                this = (this-> sym < key) ? this-> left : this-> right;
        }
        this-> core = c;
        this-> ref_ct++;
        return (this-> sym);
}

/*
 * Routine to Insert a symbol that names a unit. Return pointer to symbol.
 */

char *ins_unit (key, u)
char* key;
Unit *u
{
        extern Avl *symtab;
        Avl *this;
        Ulist *me;
        this = find (key);
        if (this == NULL)
        {
                symtab = mk_avl();
                this = symtab;
                this-> sym = mk_sym (strlen (key));
                strcpy (this-> sym, key);
        }
        if (strcmp (this-> sym, key) == 0)
        {
                me = this-> units;
                while (me != NULL)
                {
                        if (me-> u_unit = u)
                                fprintf(stderr,
                                        "Defining unit %s again.0, key);
                        me = me-> u_next;
                }
        }
        else
        {
                insert (this, key);
                this = (this-> sym < key) ? this-> left : this-> right;
                me = this-> units;
        }
        me = mk_ulist ();
        me-> u_unit = u;
        this-> ref_ct++;
        return (this-> sym);
}

/*
 * Routine to find a Global cell of name key, and return pointer to it.
 * Return the null pointer if no such cell exists.
 */

Cell *cell_find (key)
```

```
char *key;
{
        Avl *this;
        this = find (key);
        if (strcmp (this-> sym, key) != 0)
        {
                return (NULL);
        }
        else
        {
                return (this-> global);
        }
}

/*
 * Routine to find the core whose name is key, and a return pointer to it.
 * Return the null pointer if no such core exists.
 */

Core *core_find (key)
char *key;
{
        Avl *this;
        this = find (key);
        if (strcmp (this-> sym, key) != 0)
        {
                return (NULL);
        }
        else
        {
                return (this-> core);
        }
}

/*
 * Routine to find a Unit named by key and return the Ulist of pointers
 * to it.  Return the NULL pointer if no such symbol exists.
 */

Ulist *find_unit (key)
char *key;
{
        Avl *this;
        this = find (key);
        if (strcmp (this-> sym, key) != 0)
        {
                return (NULL);
        }
        else
        {
                return (this-> units);
        }
}

/*
```

```
 * Routine to decrement the reference count of a symbol and remove
 * it if the count hits zero.
 */

del_sym (key)
char *key;
{
        Avl *this;
        this = find (key);
        if (strcmp (this-> sym, key) != 0)
        {
                fprintf(stderr,"Tried to delete nonexistent symbol %s0,key);
                return;
        }
        if (this-> ref_ct == 0)
        {
                fprintf(stderr,
                        "Ref_ct on symbol %s is already zero!0, key);
                return;
        }
        this-> ref_ct--;
        if (this-> ref_ct == 0)
        {
                if (this-> global != NULL
                        || this-> core != NULL || this-> units != NULL)
                {
                        fprintf(stderr,
                                "Can't delete symbol %s0, key);
                }
                umk_sym (this-> sym);
                this-> sym = = NULL;
                delete (this);
        }

}

/*
 * Routine to Delete a Cell from the symbol table, and then decrement the
 * reference count and delete the symbol and the table node if ref_ct hits
 * zero.  (Remember it is a ghastly error if the count hits zero too soon.)
 */

del_cell (key, c);
char *key;
Cell *c;
{
        Avl *this;
        this = find(key);
        if (strcmp (this-> sym, key) != 0)
        {
                fprintf (stderr, "Tried to delete nonexistant cell %s0,key);
                return;
        }
        if (c != this-> global)
        {
```

```
                fprintf (stderr, "Cell %s is not a global0, key);
                return;
        }
        this-> global = NULL;
        if (this-> ref_ct == 0)
        {
                fprintf(stderr, "Ref_ct alread zero on Cell %s!0,key);
                return;
        }
        this-> ref_ct--
        if (this-> ref_ct == 0)
        {
                if (this-> global != NULL
                        || this-> core != NULL || this-> units != NULL)
                {
                        fprintf(stderr,
                                "Can't delete symbol %s0, key);
                }
                umk_sym (this-> sym);
                this-> sym == NULL;
                delete (this);
        }
}

/*
 * Routine to Delete a Core from the symbol table, and then decrement the
 * reference count and delete the symbol and the table node if ref_ct hits
 * zero.  (Remember it is a ghastly error if the count hits zero too soon.)
 */

del_core (key, c);
char *key;
Core *c;
{
        Avl *this;
        this = find(key);
        if (strcmp (this-> sym, key) != 0)
        {
                fprintf (stderr, "Tried to delete nonexistant core %s0,key);
                return;
        }
        if (c != this-> core)
        {
                fprintf (stderr, "Core %s is not a core0, key);
                return;
        }
        this-> core = NULL;
        if (this-> ref_ct == 0)
        {
                fprintf(stderr, "Ref_ct alread zero on symbol %s!0,key);
                return;
        }
        this-> ref_ct--
        if (this-> ref_ct == 0)
        {
```

```
                if (this-> global != NULL
                        || this-> core != NULL || this-> units != NULL)
                {
                        fprintf(stderr,
                                "Can't delete symbol %s0, key);
                }
                umk_sym (this-> sym);
                this-> sym = = NULL;
                delete (this);
        }
}

/*
 * Routine to Delete a Unit from the symbol table and decrement the reference
 * count. Remove the symbol and the table element if the count hits zero.
 */

del_unit (key, u)
char *key;
Unit *u;
{
        Avl *this;
        Plist *me, temp;
        this = find(key);
        if (strcmp (this-> sym, key) != 0)
        {
                fprintf (stderr, "Tried to delete nonexistant unit %s0,key);
                return;
        }
        me = this-> units;
        while (me != NULL)
        {
                if (me-> p_unit = = u) break;
                me = me-> p_next;
        }
        if (me = = NULL)
        {
                fprintf (stderr, "No such Unit as %s.0, key);
                return;
        }
        /* Hack! Copy next into this and delete next */
        temp = me-> p_next;
        me-> p_unit = temp-> p_unit;
        me-> p_next = temp-> p_next;
        umk_unit (temp);
        if (this-> ref_ct = = 0)
        {
                fprintf(stderr, "Ref_ct already zero on symbol %s!0,key);
                return;
        }
        this-> ref_ct--
        if (this-> ref_ct = = 0)
        {
                if (this-> global != NULL
                        this-> core != NULL || this-> units != NULL)
```

```
                {
                        fprintf(stderr,
                                "Can't delete symbol %s0, key);
                }
                umk_sym (this-> sym);
                this-> sym = = NULL;
                delete (this);
        }
}
```

```
#include < stdio.h>
#include "constraints.h"
Avl symtab = NULL;

/*
 * Symbol table.  The symbol table is a balanced (AVL) binary tree of
 * symbols, and pointers to the various objects named.
 * Contains the procedures mk_avl, umk_avl, find, branch, avl_ptr,
 * s_rotate, d_rotate, insert, swap, delete.
 * Find mk_sym and umk_sym in data.c.
 * This routine is of lasting value therefore:
 * April 23 (c) 1983 William D. Cattey  All rights reserved.
 * If anyone uses this routine, I'd like to know about it and take part
 * in whatever developement takes place.
 */

typedef enum balance {
        LEFT,
        RIGHT,
        EQUAL
} Balance;

typedef struct avl {
        struct avl *father;             /* Father to this node */
        struct avl *left;           /* Left branch */
        struct avl *right;          /* Right branch */
        Balance bal;                    /* Balance status */
        char *sym;                      /* Actual symbol string */
        int ref_ct;                     /* Reference count */
        Cell *global;                   /* Pointer to global cell if any */
        Core *core;                     /* Pointer to archetypical constr */
        Ulist units;                    /* List of pointers to units */
} Avl;

/*
 * Create an empty element of the avl tree for the symbol table.
 */

Avl *mk_avl ()
{
        Avl *a;
        if ((a = malloc (sizeof (Avl))) = = NULL) mem_err ("mk_avl");
        a-> father = NULL;
        a-> left = NULL;
        a-> right = NUll;
        a-> bal = EQUAL;
        a-> sym = NULL;
        a-> ref_ct = 0;
        a-> global = NULL;
        a-> core = NULL;
        a-> units = NULL;
        return (a);
}
```

```
/*
 * Destroy an avl tree element.
 */

umk_avl (a)
Avl *a;
{
        free ((char *)a);
}

/*
 * Find:  Locates a symbol in the symbol table, and returns a pointer to the
 * Avl_elt that contains it, or a pointer to the place where the symbol
 * should go if it does not yet exist.  (A quick compare of the symbol in
 * the elt that is returned will make sure -- its redundant but more robust,
 * and gets rid of the need to return control information as well.
 * If find returns null, the symbol table is empty.
 */

Avl *find (name)
char *name;
{
        extern Avl *symtab;
        Avl this;
        this = symtab;
        if (this == NULL) return (NULL);
        int b;
        while (this != NULL)
        {
                if ((b = strcmp (this-> sym, name)) < 0)
                        this = this-> left;
                else if (b > 0)
                        this = this-> right;
                else
                        return (this);
        }
        if (this == NULL)
                return (this-> father);
}

/*
 * Routine to figure out if we are the left or right subtree.
 * Returns EQUAL if we are the root.
 */

Balance branch (this)
Avl this;
{
        if (this-> father == NULL) return (EQUAL)
        else if (this-> father-> left = this)
                return (LEFT);
        else if (this-> father-> right != this)
                fprintf (stderr, "ERROR! Inconsistent tree.0);
        else
```

```
                return (RIGHT);
}
/*
 * Return pointer to element within AVL element that points to this.
 */

Avl **avl_ptr (this)
Avl *this;
{
        Balance dad;
        dad = branch (this);
        switch (dad) {
        case EQUAL:
                return (&symtab);
        case LEFT:
                return (&(this-> father-> left));
        case RIGHT:
                return (&(this-> father-> right));
        }
}

/*
 * Perform a single rotation.  (does not alter any balances, that is
 * the responsibility of the calling routine)
 */

s_rotate (this, child)
Avl *this;
Balance dad, child;
{
        Avl **above, **below, **middle;
        /* Identify pointer to fill with new root */
        above = avl_ptr(this);
        /* Identify pointer that points to new root */
        middle = (child == LEFT) ? &(this-> left) : &(this-> right);
        /* Identify pointer to take subtree from */
        below = (child == LEFT) ? &((*middle)-> right) : &((*middle)-> left);
        *above = *middle;
        (*above)-> father = this-> father;
        *middle = *below;
        (*middle)-> father = this;
        *below = this;
        (*below)-> father = *above;
}

/*
 * Double rotate (Balance must be modified by caller upon our return.)
 */

d_rotate (this, child)
Avl *this;
Balance dad, child;
{
        Avl **above, **below, **middle, **l_sub, **r_sub;
```

```
        /* Identify pointer to fill with new root */
        above = avl_ptr (this);
        middle = (child == LEFT) ? &(this-> left) : &(this-> right);
        below = (child == RIGHT) ? &((*middle)-> left) : &((*middle)-> right);
        l_sub = &((*below)-> left);
        r_sub = &((*below)-> right);
        *above = *below;
        (*above)-> father = this-> father;
        (*l_sub)-> father = *above;
        (*r_sub)-> father = *above;
        if (child == LEFT)
        {
                *below = *l_sub;
                *l_sub = *middle;
                *middle = *r_sub;
                *r_sub = this;
                (*below)-> father = *l_sub;
                (*middle)-> father = *r_sub;
        }
        else
        {
                *below = r_sub;
                *r_sub = *middle;
                *middle = *l_sub;
                *l_sub = this;
                (*below)-> father = *r_sub;
                (*middle)-> father = *l_sub;
        }
}

/*
 * Insert:  Takes a pointer to a symbol and a pointer to an
 * element in the symbol table. It is assumed that the symbol is not
 * already present.  The symbol is added and the avl tree is balanced.
 */

insert (elt, key)
Avl *elt;
char *key;
{
        Avl *this;
        int d;
        Balance b, b_old = EQUAL;
        if ((d = strcmp (elt-> sym, key)) == 0)
                fprintf (stderr, "Attempt to redefine symbol %s.0, key);
        this = mk_avl();
        if (d > 0)
        {
                if (elt-> right != NULL) {
                        fprintf (stderr, "ERROR! Killing valid tree branch.0);
                }
                b = RIGHT;
                elt-> right = this;
        }
        else
```

```
{
        if (elt-> left != NULL) {
                fprintf (stderr,"ERROR! Killing valid tree branch.0);
        }
        b = LEFT;
        elt-> left = this;
}
this-> sym = mk_sym (strlen (key));
strcpy (this-> sym, key);
this-> father = elt;
this = elt;
while (this != NULL)
{
        if (this-> bal == EQUAL)
                this-> bal = b;
        else if (this-> bal != b)
        {
                this-> bal = EQUAL;
                break;
        }
        else
        {
                /* Rotations during inserts require knowledge */
                /* of where we came from */
                if (b == b_old)         /* Perform single rotation */
                {
                        s_rotate (this, b);
                        this-> bal = EQUAL;
                        this-> father-> bal = EQUAL;
                        break;
                }
                else            /* Perform double rotation */
                {
                        Avl **above;
                        above = avl_ptr(this);
                        d_rotate (this, b);
                        if ((*above)-> bal = LEFT)
                        {
                                (*above)-> left-> bal = EQUAL;
                                (*above)-> right-> bal = RIGHT;
                        }
                        else
                        {
                                (*above)-> left-> bal = LEFT;
                                (*above)-> right-> bal = EQUAL;
                        }
                        (*above)-> bal = EQUAL;
                        break;
                }
        }
        /* set b to name the longer subtree which is always the one */
        /* we came from while inserting */
        b_old = b;              /* Remember two back for rotate */
        b = branch (this);
        this = this-> father;
```

```
        }
}

/*
 * Routine to take an element in the symbol table marked for deletion
 * that has two children, and find the one which is previous to it in normal
 * order which has only one child.  I arbitrarily choose to explore
 * all right children of the first left child.  (I could have searched for
 * the one immediately following instead.)  Then the contents of the
 * newly found node are copied up into elt.
 */

Avl *swap (elt)
Avl *elt
{
        Avl *other;
        for (other = elt-> left-> right;other-> right != NULL;other = other-> right)
                ;
        elt-> ref_ct = other-> ref_ct;
        elt-> sym = other-> sym;
        elt-> global = other-> global;
        elt-> core = other-> core;
        elt-> units = other-> units;
        return (other);
}

/*
 * Routine to delete an element from the Symbol table.  Checks to see if
 * it really is ok to delete the element, then chops the element,
 * patches the other elements back around the hole, frees the allocated
 * space taken up by the element, and then balances the tree.  Assumes
 * that the symbol key has already been removed from the tree.
 */

delete (elt)
Avl *elt;
{
        Avl **dad, *temp;
        balance b;
        if (elt-> ref_ct >  0 || elt-> global != NULL
                || elt-> core != NULL || elt-> plist != NULL)
        {
                fprintf(stderr,"Tried to delete active symbol %s.0,elt-> sym);
                return;
        }
        if (elt-> left != NULL && elt-> right != NULL)
                elt = swap (elt);
        b = (branch (elt);
        dad = (b == LEFT) ? &(elt-> father-> left) : &(elt-> father-> right);
        *dad = (elt-> left != NULL) ? elt-> left : elt-> right;
        temp = elt;
        if (b != EQUAL)
                (*dad)-> father = elt-> father;
        else
        {
```

```
        elt = elt-> father;
        elt-> bal = EQUAL;
        b = branch (elt);
}
elt = elt-> father;
umk_avl (temp);
while (elt != NULL)
{
        if (elt-> bal == EQUAL)
        {
                elt-> bal = (b == LEFT) ? RIGHT : LEFT;
                break;
        }
        else if (elt-> bal == b)
        {
                elt-> bal = EQUAL;
        }
        else
        {
                /* Rotations on delete require knowledge of */
                /* Height conditions */
                Avl *other;
                other = (b == LEFT) ? elt-> right : elt-> left;
                if (other-> bal == EQUAL)
                {
                        s_rotate (elt, elt-> bal);
                        elt-> bal = (b == LEFT) ? RIGHT : LEFT;
                        other-> bal = b;
                        break;
                }
                else if (other-> bal == elt-> bal)
                {
                        s_rotate (elt, elt-> bal);
                        other-> bal = EQUAL;
                        elt-> bal = EQUAL;
                        /* CONTINUE TO LOOP! */
                        elt = elt-> father;        /* skip over new dad */
                }
                else
                {
                        Avl **above;
                        above = avl_ptr(elt);
                        d_rotate (elt, elt-> bal);
                        if ((*above)-> bal == EQUAL)
                        {
                                (*above)-> left-> bal = EQUAL;
                                (*above)-> right-> bal = EQUAL;
                        }
                        else if ((*above)-> bal == LEFT)
                        {
                                (*above)-> left-> bal = RIGHT;
                                (*above)-> right-> bal = EQUAL;
                        }
                        else
                        {
```

```
                        (*above)-> left-> bal = EQUAL;
                        (*above)-> right-> bal = LEFT;
                    }
                    (*above)-> bal = EQUAL;
                    elt = *above;
                }
            }
            b = branch (elt);
            elt = elt-> father;
        }
    }
```

Appendix J
Farenheit - Celsius Converson
fccf.c

```
#include < stdio.h>
#include "constraints.h"
#include "primop.h"

/*
 * Program to evaluate ferenheit to celsius conversion using
 * a constraint network
 * Needs files constraints.h, data.c, eval.c, cell.c, mult.c
 * and plus.c
 * Contains functions main, add_tab, clr_top, t_access, attach.
 *
 * Wires up the following network:
 *
 *      Farenheit                                       Celsius
 *      |                                               |
 *      |       _A------@------A_               _B------@
 *      @------C_                 _C------@------C_
 *              +                 *               *
 *             _B__32            _B__5           _A__(9)
 *
 */

Node *mk_node();
Unit *mk_unit();
Cell *mk_cell();
Queue *q_create();
List *mk_list();
Cell *dequeue();

/*
 * Create our global objects
 */

Queue *wake, *contra;

main ()
{
        char instr[20];    /* First some declarations. */
        int inval;
        Cell *ques, *ans, *sleepy;

        Node *n1 = mk_node();          /* Now declare and create */
        Node *n2 = mk_node();          /* the Nodes. */
        Node *n3 = mk_node();
        Node *n4 = mk_node();

        Unit *adder1;              /* Now declare a plus unit */
        Unit *mult1;               /* and two multiply units. */
        Unit *mult2;
```

```
Cell *faren;            /* Now two Cells for the two */
Cell *celsius;          /* Values of interest */

wake = q_create ();     /* Create the two queues */
contra = q_create ();

adder1 = mk_unit (plus);        /* Create adder and */
adder1-> pname = "adder1";      /* multipliers. Set */
mult1 = mk_unit (mult);                 /* their names */
mult1-> pname = "multiplier1";
mult2 = mk_unit (mult);
mult2-> pname = "multiplier2";

(adder1-> vars+1)-> value = 32;         /* Now set the constant */
(adder1-> vars+1)-> state = KING;       /* Values and state within */
(adder1-> vars+1)-> rule = IS_CONST;
(mult1-> vars+1)-> value = 5;  /* the units.  KING says */
(mult1-> vars+1)-> state = KING;        /* these cells have valid */
(mult1-> vars+1)-> rule = IS_CONST;
(mult2-> vars+0)-> value = 9;  /* Values. */
(mult2-> vars+0)-> state = KING;
(mult2-> vars+0)-> rule = IS_CONST;

faren = mk_cell ();             /* Now create the two Cells */
faren-> name = "Farenheit";     /* of interest to the */
celsius = mk_cell ();           /* outside world and set */
celsius-> name = "Celsius";     /* their names */

attach (faren, n1);             /* wire the network */
attach ((adder1-> vars+2), n1);
attach ((adder1-> vars+0), n2);
attach ((mult1-> vars+0), n2);
attach ((mult1-> vars+2), n3);
attach ((mult2-> vars+2), n3);
attach ((mult2-> vars+1), n4);
attach (celsius, n4);

fprintf (stderr, "Is the input farenheit or celsius?  (f or c) : ");
scanf ("%s", instr);
while (instr [0] != 'f' && instr [0] != 'c')
{
        fprintf (stderr, "f or c, try again ");
        scanf ("%s", instr);
}
fprintf (stderr,"What is the input value? ");
scanf ("%d", &inval);

if (instr [0] == 'f')           /* Determine which Cells */
{                               /* to use for the question */
        ques = faren;           /* (input), and the answer */
        ans = celsius;          /* (output) values. */
}
else
{
        ques = celsius;
```

```
                ans = faren;
        }

        ques-> value = (Value)inval;
        ques-> state = KING;
        fix_node (ques); /* assert node supplier */
        /* Awaken queued cells! */
        while ((sleepy = dequeue (wake)) != NULL)
                awaken (sleepy);
        while ((sleepy = dequeue (contra)) != NULL)
                fix_contra (sleepy);
        fprintf (stderr, "The %s value is %d.\n", ans-> name, (int) node_val(ans));
}

/*
 * Stubs for frames.c.  Very simple since we are only uning primops
 */

add_tab (t)
int *t;
{
        fprintf (stderr, "You should never add a table!\n");
        t = t;              /* this keeps lint quiet */
}

clr_top ()
{
        fprintf (stderr, "You should never clear a top!\n");
}

Cell *t_access (n, u)
int n;
Unit *u;
{
        return (u-> vars+n);     /* This is a pointer! */
}

/*
 * Simplified attach (as compared to the one in wire.c)
 * designed for wiring DEAD network structure together.
 */

attach (c, n)
Cell *c;
Node *n;
{
        List *l;
        if (c-> repository != NULL)
                fprintf (stderr, "Cell already has a node.\n");
        c-> repository = n;
        l = mk_list ();
        l-> l_cell = c;
        l-> l_next = n-> cells;
        n-> cells = l;
}
```

```
extern Core *plus;
extern Core *mult;
extern Rule *IS_CONST;
```

```
#include < stdio.h>
#include "constraints.h"
#include "primop.h"

/*
 * Definition of a constant Cell
 * in the event you dont want to make it from mk_cell in data.c.
 * Remember a Cell consists of the following:
 *
 * char *name;
 * Value value;
 * State state;
 * Node *repository;
 * Unit *owner;
 * int index;            set to NULL
 * Rule *rule;           set to IS_CONST
 * int mark;
 *
 */


/*
 * This is code to signal an error.  The constant rule should never be
 * applied
 */

Value is_const () {
        (fprintf stderr, "You can't apply the constant rule!");
        return (0);
}

/*
 * Hardwire in a constant rule
 * The purpose of this rule is to sit and be looked at if ever anyone
 * wants to know how the cell got its value.  It should never run.
 * However, since the list of ins has zero length, it will always
 * win in a duel of reasons why a cell got a value.
 */

int c_i [] = {-1};          /* Always win in a duel! */

Irule RIS_CONST = {
        PRIMOP,                     /* Yes, dear friends, Its a primop */
        c_i,                /* Nothing ever changes */
        0,                  /* No output pin */
        0,                  /* No delay */
        0,                  /* Flag bits start as zero */
        NULL,               /* No magical text */
        is_const        /* At last Executable code! */
};

Rule *IS_CONST = (Rule *)&RIS_CONST;
```

Appendix K
Oscillator Simulation
lsim.c

```
#include < stdio.h>
#include "constraints.h"
#include "lprimop.h"

/*
 * Program to wire up a simple logic circuit and simulate
 * its operation.
 * Needs files constraints.h, data.c, lseval.c, cell.c,
 * nand.c and not.c
 * Contains functions main, add_tab, clr_top, t_access, attach.
 *
 * Wires up the following network:
 *
 *
 *      ________________
 *      |               |
 *      in              |
 *      .NOT.           |
 *      out             |
 *      |               |
 *      |               |
 *      @               @__kicker T
 *      |       T       |
 *      |       |       |
 *      A       B       |
 *      | .AND. |       |
 *         Q            |
 *         |            |
 *      |---@            |
 *      |               |
 *      in              |
 *      .NOT.           |
 *      out             |
 *      |               |
 *      ________________
 *
 * Whis is about the cleanest tty representation you can get of
 * ring of inverters as an oscillator.  (the nand gate in the middle
 * is an added complexity to prove more things work.
 *
 * This program is designed to run endlessly interrupt it when you
 * get tired of watching it.
 *
 */

Node *mk_node();
Unit *mk_unit();
Cell *mk_cell();
Pqueue *pq_create();
List *mk_list();
Cell *pdequeue();
```

```
/*
 * Create our global objects
 */

Pqueue *agenda;

main ()
{
        Cell *sleepy;

        Node *n1 = mk_node();        /* Now declare and create */
        Node *n2 = mk_node();        /* the Nodes. */
        Node *n3 = mk_node();

        Unit *nand1;            /* Now declare a nand unit */
        Unit *not1;             /* and two inverter units. */
        Unit *not2;

        Cell *kicker;           /* cheat and MAKE it run! */

        agenda = pq_create();  /* create the agenda */

        nand1 = mk_unit (nand);              /* Create nand and */
        nand1-> pname = "NAND1";             /* inverters.  Set */
        not1 = mk_unit (not);           /* their names */
        not1-> pname = "NOT1";
        not2 = mk_unit (not);
        not2-> pname = "NOT2";

        /* Now set the constant Values and state within the units. */
        /* KING says these cells have valid Values. */
        (nand1-> vars+1)-> value = T;
        (nand1-> vars+1)-> state = KING;
        (nand1-> vars+1)-> rule = IS_CONST;

        kicker = mk_cell();             /* make the kicker */
        kicker-> name = "kicker";

        /* Wire the network. */
        attach ((not1-> vars+1), n1);
        attach ((nand1-> vars+0), n1);
        attach ((nand1-> vars+2), n2);
        attach ((not2-> vars+0), n2);
        attach ((not2-> vars+1), n3);
        attach ((not1-> vars+0), n3);
        attach (kicker, n3);

        fprintf (stderr,"Network wired, begining run.\n");

        kicker-> value = T;
        kicker-> state = KING;
        fix_node (kicker, 0);      /* assert node supplier */
        /* Awaken queued cells! */
        while ((sleepy = pdequeue (agenda)) != NULL)
                awaken (sleepy);
```

```
}

/*
 * Stubs for frames.c.  Very simple since we are only uning primops
 */

add_tab (t)
int *t;
{
        fprintf (stderr, "You should never add a table!\n");
        t = t;              /* this keeps lint quiet */
}

clr_top ()
{
        fprintf (stderr, "You should never clear a top!\n");
}

Cell *t_access (n, u)
int n;
Unit *u;
{
        return (u-> vars+n);     /* This is a pointer! */
}

/*
 * Simplified attach (as compared to the one in wire.c)
 * designed for wiring DEAD network structure together.
 */

attach (c, n)
Cell *c;
Node *n;
{
        List *l;
        if (c-> repository != NULL)
                fprintf (stderr, "Cell already has a node.\n");
        c-> repository = n;
        l = mk_list ();
        l-> l_cell = c;
        l-> l_next = n-> cells;
        n-> cells = l;
}
```

```
extern Core *nand;
extern Core *not;
extern Rule *IS_CONST;
```

```
#include < stdio.h>
#include "constraints.h"
#include "lprimop.h"

/*
 * Definition of a constant Cell
 * in the event you dont want to make it from mk_cell in data.c.
 * Remember a Cell consists of the following:
 *
 * char *name;
 * Value value;
 * State state;
 * Node *repository;
 * Unit *owner;
 * int index;              set to NULL
 * Rule *rule;             set to IS_CONST
 * int mark;
 *
 */


/*
 * This is code to signal an error.  The constant rule should never be
 * applied
 */

Value is_const () {
        (fprintf stderr, "You can't apply the constant rule!\n");
        return (F);                 /* keeps lint quiet */
}

/*
 * Hardwire in a constant rule
 * The purpose of this rule is to sit and be looked at if ever anyone
 * wants to know how the cell got its value.  It should never run.
 * However, since the list of ins has zero length, it will always
 * win in a duel of reasons why a cell got a value.
 */

int c_i [] = {-1};          /* Always win in a duel! */

Irule RIS_CONST = {
        PRIMOP,                         /* Yes, dear friends, Its a primop */
        c_i,                    /* Nothing ever changes */
        0,                      /* No output pin */
        0,                      /* No delay */
        0,                      /* Flag bits start as zero */
        NULL,                   /* No magical text */
        is_const            /* At last Executable code! */
};

Rule *IS_CONST = (Rule *)&RIS_CONST;
```

```
#include < stdio.h>
#include "constraints.h"
#include "lprimop.h"

/*
 * Hardwired plus constraint
 */

Cell *t_access ();
Value node_val();

/*
 * ns is the indexed array of pointers to names of pins
 */

char *not_ns [] = { "in", "out" };

/*
 * This function does the actual work.  It calls t_access to
 * grab a Cell and then mung its value.
 */

Value n_f (u)                   /* out = not(in) */
Unit *u;
{
        Cell *p;
        Value  t;
        p = t_access (0, u);
        t = node_val(p);
        switch (t)
        {
        case TRUE:
                return(F);
        case FALSE:
                return(T);
        default:
                fprintf (stderr, "The ");
                print_cell(p);
                fprintf (stderr, " has a *BAD* value .\n");
                fatal ();
                return(F);      /* keeps lint quiet */
        }
}

/*
 * This is the ins array.
 */

int n_i[] = { 0, -1};           /* Pin 0 must have a value. */

/*
```

```
 * Now the rule for pin 0.
 */

Irule n_r =
{
        PRIMOP,                 /* ty */
        n_i,                    /* ins */
        1,                      /* out */
        3,                      /* delay */
        0,                      /* flags */
        "invert rule",          /* text */
        n_f                     /* primop */
};

/*
 * Now, by hand, without forward references we create the
 * list of rules.
 */

Rlist n_l  = {(Rule *)&n_r, NULL };

Rlist *not_rs [] = {&n_l, NULL};

/*
 * Finally, the core!  The unit is created in real time
 * mk_unit.
 */

Core n_core =
{
        "not",
        2,
        not_ns,
        not_rs
};

Core *not = &n_core;
```

```
#include < stdio.h>
#include "constraints.h"
#include "lprimop.h"

/*
 * Hardwired plus constraint
 */

Cell *t_access ();
Value node_val();

/*
 * ns is the indexed array of pointers to names of pins
 */

char *nand_ns [] = { "a", "b", "q" };

/*
 * This function does the actual work.  It calls t_access to
 * grab Cells and then mung on their values.  Currently it does
 * the computation without using trick knowledge of the
 * representation of the enumerations of Value.
 */

Value na_f (u)                  /* q = a AND b */
Unit *u;
{
        Value a, b;
        a = node_val(t_access (0, u));
        b = node_val(t_access (1, u));
        if (a == T && b == T)
                return (F);
        else
                return (T);
}

/*
 * This the array of ins.
 */

int na_i[] = { 0, 1, -1};  /* Pins 0 and 1 must have values. */

/*
 * Now the actual rule for pins 0, and 1.
 */

Irule na_r =
{
        PRIMOP,                 /* ty */
        na_i,                   /* ins */
        2,                      /* out */
        4,                      /* delay */
        0,                      /* flags */
```

```
        "NAND rule",            /* text */
        na_f                    /* primop */
};

/*
 * Now, by hand, without forward references we create the
 * lists of rules complete with their pointers to next for
 * pins 0, 1, and 2 respectively.
 * Each pin has two rules.  Depending on which if either
 * of the remaining two pins have values, each pin may run
 * and assert a value on one or the other remaining pins.
 */

Rlist na_l = {(Rule *)&na_r, NULL };

Rlist *nand_rs [] = {&na_l, &na_l, NULL};

/*
 * Finally, the core!  The unit is created in real time
 * mk_unit.
 */

Core na_core =
{
        "nand",
        3,
        nand_ns,
        nand_rs
};

Core *nand = &na_core;
```

Appendix M
makefile

This file contains the mysterious makefile that compiled the logic simulator, the constraints system, and the thesis. It shows how the appendicies were generated from the programs that made up the system. The sed script called scr was a simple file that converted the backslash character (\) into something that the troff text processor could print.

```
CFLAGS = -g

OBJFILES = data.o plus.o mult.o cell.o eval.o fccf.o
OBJSIM = ldata.o lcell.o lseval.o nand.o not.o lsim.o
SRCSIM = data.c lcell.c lseval.c nand.c not.c lsim.c

NFILES = constraints.n data.n eval.n frames.n plus.n mult.n main.n \
commands.n wire.n symtab.n avl.n fccf.n cell.n lseval.n nand.n not.n \
lsim.n lcell.n makefile.n race.n

fccf: $(OBJFILES)
        $(CC) $(CFLAGS) $(OBJFILES) -o fccf
        rm -f core

lsim: $(OBJSIM)
        $(CC) $(CFLAGS) -DLOGIC $(OBJSIM) -o lsim
        rm -f core

race : lsim race.o
        $(CC) $(CFLAGS) -DLOGIC ldata.o lcell.o lseval.o nand.o not.o race.o -o race

race.o : race.c constraints.h lprimop.h
        $(CC) -c $(CFLAGS) -DLOGIC race.c

lseval.o : constraints.h
        $(CC) -c $(CFLAGS) -DLOGIC lseval.c

nand.o : constraints.h lprimop.h
        $(CC) -c $(CFLAGS) -DLOGIC nand.c

not.o : constraints.h lprimop.h
        $(CC) -c $(CFLAGS) -DLOGIC not.c

lcell.o : constraints.h lprimop.h
        $(CC) -c $(CFLAGS) -DLOGIC lcell.c

lsim.o : constraints.h lprimop.h
        $(CC) -c $(CFLAGS) -DLOGIC lsim.c

ldata.o : constraints.h ldata.c
        $(CC) -c $(CFLAGS) -DLOGIC ldata.c

ldata.c : data.c
        cp data.c ldata.c

data.o : constraints.h
```

```
plus.o : constraints.h primop.h

mult.o : constraints.h primop.h

eval.o : constraints.h

fccf.o : constraints.h primop.h

cell.o : constraints.h primop.h

clean :
        rm -f $(OBJFILES)
        rm -f $(OBJSIM)
        rm -f $(NFILES)

lint :
        lint data.c plus.c mult.c eval.c fccf.c cell.c

llint:
        lint -DLOGIC data.c lseval.c lcell.c nand.c not.c lsim.c

thesis: ctext
        dtroff title abstr cont intro motive hist impl1 impl2 impl3 \
impl4 impl5 end ref apb apc apd ape apf apg aph api apj apk apl apm \
apn apo

ctext : $(NFILES)

constraints.n : constraints.h
        sed -f scr < constraints.h > constraints.n
lseval.n : lseval.c
        sed -f scr lseval.c > lseval.n
lsim.n : lsim.c
        sed -f scr lsim.c > lsim.n
nand.n : nand.c
        sed -f scr nand.c > nand.n
not.n : not.c
        sed -f scr not.c > not.n
data.n : data.c
        sed -f scr data.c > data.n
eval.n : eval.c
        sed -f scr eval.c > eval.n
frames.n : frames.c
        sed -f scr frames.c > frames.n
plus.n : plus.c
        sed -f scr plus.c > plus.n
mult.n : mult.c
        sed -f scr mult.c > mult.n
main.n  : main.c
        sed -f scr main.c > main.n
commands.n : commands.c
        sed -f scr commands.c > commands.n
wire.n : wire.c
        sed -f scr wire.c > wire.n
symtab.n : symtab.c
```

```
	sed -f scr symtab.c > symtab.n
avl.n : avl.c
	sed -f scr avl.c > avl.n
fccf.n : fccf.c
	sed -f scr fccf.c > fccf.n
cell.n : cell.c
	sed -f scr cell.c > cell.n
lcell.n : lcell.c
	sed -f scr lcell.c > lcell.n
makefile.n : makefile
	sed -f scr makefile > makefile.n
race.n : race.c
	sed -f scr race.c > race.n
```

Appendix N
Logic Simulation Evaluator
lseval.c

```
#include < stdio.h>
#include "constraints.h"
#define MAX_NUM 14          /* Maximum chars in a number */

/*
 * This file Contains the functions for evaluating a constraint network.
 * We can diddle values here, awaken cells, render nodes consistent, run
 * rules, and do whatever housekeeping is neccessary.
 * getsup, boundp, print_val, node_val, cell_value,
 *  wake_slaves, sel_rule, duel,
 * fix_node, forget, apply, awaken, fix_contra.
 */

/*
 *  Remember to get t_access from frames.c or write one by hand.
 */

Cell *t_access();

/*
 * Return the cell who is the supplier of this cell's node.  If this
 * cell has no node, return this cell.
 */

Cell *getsup (c)
Cell *c;
{
        Node *n;
        if ((n = c-> repository) == NULL)
                return (c);
        else if (n-> supplier == NULL)
                return (c);
        else
                return (n-> supplier);
}

/*
 * Determine if this cell's node is bound.
 * First get the supplier of the node.  The funciton getsup
 * returns the cell itself if the node had no supplier.
 * If the supplier is king or puppet, the job is easy:
 * Return TRUE if KING, FALSE if PUPPET.  Fix_node will deal with
 * the case of this cell wanting to become the supplier of a supplier-
 * less node.  If the cell is a slave and there is no supplier,
 * we return FALSE.
 */

boundp (c)
Cell *c;
{
        State s;
```

```
        State s;
        char *str;
        c = getsup (c);
        s = c-> state;
        switch (s) {
        case KING:
                return (TRUE);
        case PUPPET:
                return (FALSE);
        case FRIEND:
                str = "Friend";
                break;
        case SLAVE:
                if (c-> repository-> supplier == NULL)
                        return (FALSE);
        case REBEL:
                str = "Rebel";
                break;
        default:
                str = "Impossible";
        }
        fprintf (stderr, "The ");
        print_cell (c);
        fprintf(stderr, " is unbound in state %s.\n", str);
        fatal ();
        return (FALSE);                 /* keeps lint quiet */
}

/*
 * Print out a value on the standard error.  This routine can get
 * arbitrarily complex as values get more complex.
 */

print_val (v)
Value v;
{
        switch (v)
        {
        case T:
                fprintf (stderr, "TRUE");
                return;
        case F:
                fprintf (stderr, "FALSE");
                return;
        default:
                fprintf (stderr, "*BAD*");
        }

}

/*
 * Given a cell return the value of its node.
 */

Value node_val (cell)
```

```
Cell *cell;
{
        cell = getsup (cell);
        if (cell-> state == KING)
                return (cell-> value);
        else {
                fprintf(stderr,"Supplier for %s is not king!\n",cell-> name);
                fatal ();
                return (F);       /* keeps lint quiet */
        }
}


/*
 * Return an appropriate cell value for this cell., Get a node value
 * if necessary.
 */

Value cell_val (cell)                   /* find a cell's value */
Cell *cell;
{
        State s;
        s = cell-> state;
        switch (s) {
        case KING:
        case FRIEND:
        case REBEL:
                return (cell-> value);
        case SLAVE:                     /* NEVER look at the value of a slave */
                return (node_val (cell)); /* Because I never clear it */
        case PUPPET:
                fprintf (stderr,"You tried to take the value of a PUPPET\n");
                return (F);
        default:
                fprintf (stderr, "Cell in impossible state!\n");
                fatal ();
                return (F);       /* keeps lint quiet */
        }
}

/*
 * When a node's king changes, all the slaves must be awakened to deal
 * with the new node value. This routine goes through a node and enqueues
 * all the slave cells onto the wake queue where awaken will eventually see
 * them. Changed values propagate through a network from the awakened cell as
 * follows: An awakened cell looks at the constraint that owns it and tries
 * to run all the rules it knows about. The rules look at nearby cells and
 * assert or retract values of nodes they output to, subject to the delay
 * offset time constraint. Each of these nodes is cleaned up in turn, by
 * setting an unambiguous king and queueing all slave cells to be awakened.
 * This order of propagation seems to be the most robust.
 */

wake_slaves (node, o)
Node *node;
```

```
int o;
{
        List *l;
        Cell *me;
        l = node-> cells;
        while (l != NULL)
        {
                me = l-> l_cell;
                if (me-> state == SLAVE)
                        penqueue (agenda, me, o);
                l = l-> l_next;
                me = l-> l_cell;
        }
}

/*
 * Given two rules r1 and r2 which are primops, return r1 if both lists
 * of ins are same length.  If r1's list is longer return r2 if its
 * list of ins  is a proper subset of r1's.
 */

Rule *sel_rule (r1, r2)
Rule *r1, *r2;
{
        int k, f, *kins, *fins;
        kins = r1-> ins;
        fins = r2-> ins;
        while (((f = *fins++) != -1) && ((k = *kins++) != -1))
                ;                               /* fly to the end of one list */
        if (k != -1) return (r1);  /* Implies fins > = kins */
        fins = r2-> ins;
        while ((f = *fins++) != -1)
        {
                kins = r1-> ins;
                while ((k = *kins++) != -1)
                {
                        if (f == k) break;
                }
                if (f != k) return (r1);
        }
        if (*fins == -1) return (r2);
        else return (r1);
}

/*
 * Given a friend and a king install the friend as king if he has a
 * value depending on a subset of the variables that gave the king his value.
 */

duel (kng, fnd)
Cell *kng, *fnd;
{
        Rule *k, *f, *a;
        if (kng-> repository-> supplier != kng)
                fprintf (stderr, "King not first in duel\n");
```

```
        k = kng-> rule;
        f = fnd-> rule;
        a = sel_rule (k, f);
        if (a == k)
                fnd-> state = FRIEND;
        else {
                kng-> state = FRIEND;
                kng-> repository-> supplier = fnd;
        }
}

/*
 * Function to take a cell and make its node consistant.
 * Each cell in the node is examined.  Its value and state are
 * updated, then it is enqueued to have its owner awakened
 * When called, cell is either a PUPPET if the value was just
 * forgotten or a KING if a new value was just set.  This routine
 * finds a new king if possible when the cell is a puppet, or deals
 * with instaling the new KING.
 */

fix_node (c, o)
Cell *c;
int o;
{
        Node *n;
        State s;
        Cell *other;
        s = c-> state;
        n = c-> repository;
        other = n-> supplier;
        if (s == KING)          /* trying to set node value */
        {       /* If there is no supplier, make us it. */
                if (other == 0)
                {
                        n-> supplier = c;
                        wake_slaves (n, o);
                }
                /* If we are already supplier wake slaves with new value */
                else if(other == c) wake_slaves (n, o);
                /* If c supplier is PUPPET, make us supplier */
                else if (other-> state == PUPPET) {
                        other-> state = SLAVE;
                        n-> supplier = c;
                        wake_slaves (n, o);
                }
                /* If we are a friend of supplier select best KING */
                else if (other-> value == c-> value)
                        duel (other, c); /* Battle for Supplier hood */
                /* Otherwise we disagree and are among the REBELS */
                /* In a logic simulator, that means the new supersedes */
                /* the old */
                else {
                        other-> state = REBEL;
                        n-> supplier = c;
```

```
                        fprintf (stderr, "The ");
                        print_cell (c);
                        fprintf (stderr, " with value ");
                        print_val (c-> value);
                        fprintf (stderr, " supersedes the\n");
                        print_cell (other);
                        fprintf (stderr, " with value ");
                        print_val(other-> value);
                        fprintf (stderr, ".\n");
                        wake_slaves (n,o);
                }
        }
        else {                          /* We must be a PUPPET */
                if (other = = c) {      /* If we were KING */
                        List *l;
                        l = n-> cells;  /* Look for a FRIEND to KING */
                        while (l != NULL) {
                                other = l-> l_cell;
                                if (other-> state = = FRIEND) {
                                        other-> state = KING;
                                        n-> supplier = other;
                                        return;
                                }
                                l = l-> l_next;
                                other = l-> l_cell;
                        }
                        l = n-> cells;  /* Look for a REBEL to KING */
                        while (l != NULL) {
                                other = l-> l_cell;
                                if (other-> state = = REBEL)
                                {
                                        other-> state = KING;
                                        n-> supplier = other;
                                        wake_slaves(n, o);
                                        return;
                                }
                                l = l-> l_next;
                                other = l-> l_cell;
                        }
                        n-> supplier = c; /* No hope forget node value */
                        wake_slaves(n, o);
                }
                else
                fprintf(stderr, "You called fix_node to forget a slave\n");
        }
}

/*
 * Forget the contents of a cell, taking into consideration
 * its old state.  This routine is called when a cell's value
 * was forgotten as a consequence a constraint being undone.
 */

forget (c, o)
Cell *c;
```

```
int o;
{
        State s;
        s = c-> state;
        switch (s) {
                case PUPPET:  /* No retraction needed */
                case SLAVE:   /* It CANT be our fault */
                        break;
                case KING:
                        c-> state = PUPPET;
                        if (c-> repository != NULL) fix_node (c, o);
                        break;
                case REBEL:
                        c-> state = SLAVE;
                        awaken (c);
                        break;
                case FRIEND:
                        c-> state = SLAVE;
                        break;
        }
}

/*
 * Recursive apply.  This routine relies on the magic contained in the
 * file frames.c to work.  It is imperative that, each time a new abstract
 * level is dug down into, the translataion table is pushed onto the translation
 * stack.
 */

apply (u, sup, rl)
Unit *u;
Cell *sup;
Rlist *rl;
{
        Rule *r;
        while (rl != NULL && (r = rl-> r_rule) != NULL)
        {
                if (r-> ty == CONSTR)
                {
                        Core *c;
                        Rlist *nr;              /* nr stands for new rules */
                        int nn, *table;         /* nn stands for new name */
                        table = r-> ins;        /* get ttab from Rule */
                        add_tab (table); /* Push onto stack of ttabs */
                        c = r-> op_ptr.ragmop; /* get core for lower level op */
                        nn = r-> out;           /* new name from Rule */
                        nr = *(c-> rules+nn);  /* new rule */
                        apply (u, sup, nr);     /* Apply it all to lower level op */
                        clr_top();              /* Flush ttab when finished */
                }
                else
                {
                        Value v, (*p)();
                        Cell *ans;
                        int o, ivar, *inpt;
```

```
                        o = r-> delay;
                        ans = t_access(r-> out, u);
                        if (sup-> state == PUPPET)
                        {
                                if (ans-> rule == r) forget (ans, o);
                        }
                        inpt = r-> ins;
                        /* First check to see if we can run rule */
                        while ((ivar = *inpt++) != -1)
                                if (!boundp (t_access(ivar, u))) goto next;
                        p = r-> op_ptr.primop;
                        v = p(u);       /* NOTE p can look at ALL vars */
                        if ( boundp (ans) && node_val (ans) == v )
                         { /* avoid work */
                           if (ans-> state == KING)
                                ans-> rule = sel_rule (ans-> rule, r);
                           return;
                         }
                        ans-> value = v;
                        fprintf (stderr, "The ");
                        print_cell (ans);
                        fprintf (stderr, " now has value ");
                        print_val (v);
                        fprintf (stderr, ".\n");
                        ans-> state = KING;
                        ans-> rule = r;
                        fix_node (ans, o);
                }
                next:           /* break out two levels */
                rl = rl-> r_next;
        }
}

/*
 * When a node's value has changed, each cell must be awakened to cause its
 * owner (a constraint unit) to run and generate (or retract) values
 * This routine assumes that the cell's value is already changed
 * and uses its current state to generate consequences
 */

awaken (cell)
Cell *cell;
{
        Cell *sup;
        Unit *u;
        Rlist *r;
        Core *c;
        int pin;
        if (cell-> repository == NULL) {
                fprintf(stderr,"Wake a lone cell?\n");
                return;
        }
        if ((u = cell-> owner) == NULL) return; /* If a constant, no rule */
        c = u-> ctype;
        pin = cell-> index;
```

```
        r = *(c-> rules+pin);           /* The rules that fire on this cell */
        sup = getsup(cell);
        apply (u, sup, r);
}

/*
 * Print out a cell name as either a global cell or a constituent of
 * of a constraint box.
 */

print_cell (c)
Cell *c;
{
        if (c-> owner == NULL)
        {
                fprintf (stderr, "global cell %s",c-> name);
        }
        else
        {
                fprintf (stderr, "cell %s of %s %s",
                        c-> name, c-> owner-> ctype-> symbol, c-> owner-> pname);
        }
}

/*
 * This routine runs when there is a contradiction.  Unfortunately, it
 * doesn't do much of anything yet.  It should MAKE the user do something
 * about the contradiction, but alas it just warns of the existance of
 * a contradiction, and then forgets about it.
 */

fix_contra (c)
Cell *c;
{
        fprintf (stderr, "The ");
        print_cell (c);
        fprintf (stderr, " has value: ");
        print_val (c-> value);
        fprintf (stderr, " which contradicts.\n");
        fprintf (stderr, " with node supplier  the ");
        print_cell (c-> repository-> supplier);
        fprintf (stderr, " which has value: ");
        print_val (c-> repository-> supplier-> value);
        fprintf (stderr, ".\n");
}
```

Appendix O
Race Condition Network
race.c

```
#include < stdio.h>
#include "constraints.h"
#include "lprimop.h"

/*
 * Program to wire up a simple logic circuit and simulate
 * its operation.
 * Needs files constraints.h, data.c, lseval.c, cell.c,
 * nand.c and not.c
 * Contains functions main, add_tab, clr_top, t_access, attach.
 *
 * Wires up the following network:
 *
 *
 *      ____________________
 *      |                   |
 *      out                 |
 *      .NOT1.              |
 *      in                  |
 *      |                   |
 *      |                   |
 *      @_kicker T          @
 *      |                   |
 *      |         T         |
 *      |         |         |
 *      A         B         |
 *      | .AND.   |         |
 *         Q                |
 *         |                |
 *      |---@                |
 *      |                   |
 *      in                  |
 *      .NOT2.              |
 *      out                 |
 *      |                   |
 *      ____________________
 *
 * This is a random network that proves priority queues work
 * correctly.
 *
 */

Node *mk_node();
Unit *mk_unit();
Cell *mk_cell();
Pqueue *pq_create();
List *mk_list();
Cell *pdequeue();

/*
 * Create our global objects
 */
```

```
Pqueue *agenda;

main ()
{
        Cell *sleepy;

        Node *n1 = mk_node();          /* Now declare and create */
        Node *n2 = mk_node();          /* the Nodes. */
        Node *n3 = mk_node();

        Unit *nand1;                   /* Now declare a nand unit */
        Unit *not1;                    /* and two inverter units. */
        Unit *not2;

        Cell *kicker;                  /* cheat and MAKE it run! */

        agenda = pq_create();  /* create the agenda */

        nand1 = mk_unit (nand);                 /* Create nand and */
        nand1-> pname = "NAND1";                /* inverters.  Set */
        not1 = mk_unit (not);          /* their names */
        not1-> pname = "NOT1";
        not2 = mk_unit (not);
        not2-> pname = "NOT2";

        /* Now set the constant Values and state within the units. */
        /* KING says these cells have valid Values. */
        (nand1-> vars+1)-> value = T;
        (nand1-> vars+1)-> state = KING;
        (nand1-> vars+1)-> rule = IS_CONST;

        kicker = mk_cell();            /* make the kicker */
        kicker-> name = "kicker";

        /* Wire the network. */
        attach (kicker, n1);
        attach ((not1-> vars+0), n1);
        attach ((nand1-> vars+0), n1);
        attach ((not1-> vars+1), n2);
        attach ((not2-> vars+1), n2);
        attach ((nand1-> vars+2), n3);
        attach ((not2-> vars+0), n3);

        fprintf (stderr,"Network wired, beginning run.\n");

        kicker-> value = T;
        kicker-> state = KING;
        fix_node (kicker, 0);      /* assert node supplier */
        /* Awaken queued cells! */
        while ((sleepy = pdcqueue (agenda)) != NULL)
                awaken (sleepy);
}

/*
```

```
 * Stubs for frames.c.  Very simple since we are only uning primops
 */

add_tab (t)
int *t;
{
        fprintf (stderr, "You should never add a table!\n");
        t = t;              /* this keeps lint quiet */
}

clr_top ()
{
        fprintf (stderr, "You should never clear a top!\n");
}

Cell *t_access (n, u)
int n;
Unit *u;
{
        return (u-> vars+n);     /* This is a pointer! */
}

/*
 * Simplified attach (as compared to the one in wire.c)
 * designed for wiring DEAD network structure together.
 */

attach (c, n)
Cell *c;
Node *n;
{
        List *l;
        if (c-> repository != NULL)
                fprintf (stderr, "Cell already has a node.\n");
        c-> repository = n;
        l = mk_list ();
        l-> l_cell = c;
        l-> l_next = n-> cells;
        n-> cells = l;
}
```