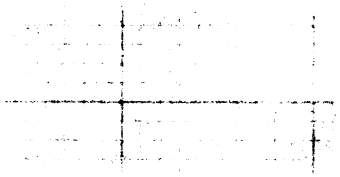
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

ARTIFICIAL INTELLIGENCE LABORATORY

# THE USE OF EQUALITY IN DEDUCTION AND KNOWLEDGE REPRESENTATION

David Allen McAllester

January 1980



*This blank page was inserted to preserve pagination.*

# The Use of Equality In Deduction and Knowledge Representation

by

**David Allen McAllester**

Massachusetts Institute of Technology

January 1980

This report is a revised version of thesis submitted to the department of Electrical Engineering and Computer Science on August 10, 1979 in partial fulfillment of the requirements for the degree of Master of Science.

This report describes work done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643 and in part by National Science Foundation Grant MCS77-04828. The author is currently receiving support as an IBM fellow.

## Abstract

This report describes a system which maintains canonical expressions for designators under a set of equalities. Substitution is used to maintain all knowledge in terms of these canonical expressions. A partial order on designators, termed the better-name relation, is used in the choice of canonical expressions. It is shown that with an appropriate better-name relation an important engineering reasoning technique, propagation of constraints, can be implemented as a special case of this substitution process. Special purpose algebraic simplification procedures are embedded such that they interact effectively with the equality system. An electrical circuit analysis system is developed which relies upon constraint propagation and algebraic simplification as primary reasoning techniques. The reasoning is guided by a better-name relation in which referentially transparent terms are preferred to referentially opaque ones. Multiple description of subcircuits are shown to interact strongly with the reasoning mechanisms.

A conceptual distinction is made between propositional deduction and the instantiation of quantified knowledge. A special purpose system is used for the simplified domain of propositional deduction which incorporates many useful features. This system, termed a truth maintenance system or TMS, keeps track of justifications for deductions which are used to generate explanations of deduced beliefs. The TMS also handles the retraction of assumptions when contradictions arise. When assumptions are retracted all deductions which depended on them are retracted in an efficient incremental manner. Assumptions that certain algebraic quantities are non-zero play an important role in reasoning about algebraic constraints.

## Acknowledgments

I owe much to the various people who have contributed to this work. First I would like to mention my thesis supervisor, Gerald Sussman, without whose guidance and criticism the ideas in this document would never have been generated. I would also like to thank my colleagues Charles Rich, Johan de Kleer, Howard Shrobe, Jon Doyle, and Gerald Roylance for the technical discussions which played a major role in the development of this work. I must also acknowledge the entire community at the MIT AI Lab for providing fine tools and a good intellectual environment.

## Contents


| | | |
|---|---|---|
| Overview | | 1 |
| CHAPTER I | The TMS, Canonical Naming, and Constraints | 5 |
| | Using the TMS | 5 |
| | Canonical Naming | 8 |
| | Algebraic Unknowns | 9 |
| | Constraint Propagation as Computation via Naming | 10 |
| CHAPTER II | Procedural Embedding | 16 |
| | Evaluation Functions | 16 |
| | Disequalities | 19 |
| | Plunks | 20 |
| CHAPTER III | Application to Electronics | 23 |
| | Basic Predicates | 23 |
| | The Better-Name Relation in Electronics | 27 |
| | A Simple Analysis | 28 |
| | Unconstrained Plunks | 30 |
| | Slices | 36 |
| | "Redrawing" Circuits | 42 |
| CHAPTER IV | Algorithmic Details and Relation to Other Work | 45 |
| | The Truth Maintenance System | 45 |
| | The Canonical Naming Algorithm | 47 |
| | Substitution | 49 |
| | Relation to Other Mechanisms for Handling Equality | 50 |
| APPENDIX 1 | A Basic TMS and Equality System User's Manual | 54 |
| APPENDIX 2 | The Equality System Code | 66 |
| APPENDIX 3 | The Algebraic Simplifier | 80 |
| APPENDIX 4 | Plunks and The Symbolic Algebra Interface | 92 |
| APPENDIX 5 | Lisp Utility Procedures | 97 |
| References | | 108 |

## Overview

In almost all reasoning systems, some method of handling the equality of objects must be devised. The problems associated with equality, or multiple expressions which have the same referent, are usually treated as an orthogonal issue to other problem solving or reasoning techniques. The approach taken here is that multiple descriptions for objects and the issues surrounding equality are of central importance in problem solving itself. A uniform algorithm for handling equality is developed here which assigns canonical names to equivalence classes of expressions. Specific reasoning techniques, such as propagation of constraints, are built up using this algorithm as a primary deductive mechanism. Propagation of constraints is then used as the major reasoning mechanism in an electronic analysis system. Figure 1 shows an overall breakdown of the reasoning system. The remainder of this section describes the various systems and the way they depend on one another.

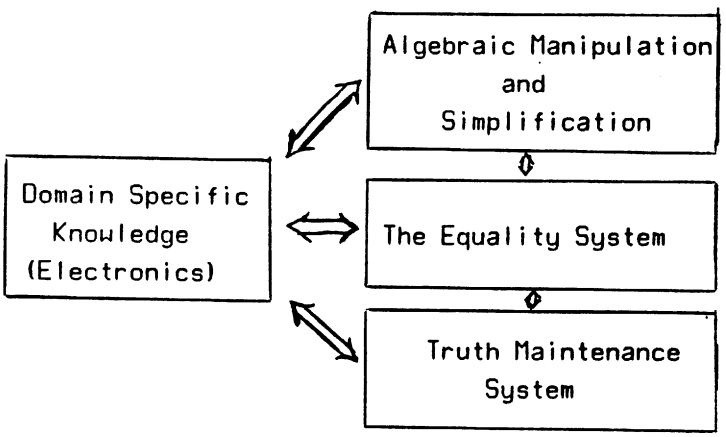



Figure 1. The Major Subsystems.

Underlying all the systems described here is a truth maintenance system (TMS) which handles all propositional deduction [Doyle 79] [McAllester 78]. The TMS keeps track of justifications for all deduced truth values. These justifications are used to generate explanations for any deduced truth value for a proposition. Such explanations give the user a better understanding of the deductions being performed and therefore more confidence in the results. The justifications are also used to incrementally update the truth value of all propositions when assumptions are added or retracted. When contradictions

occur the assumptions underlying them can be pinpointed and the negation of one of these assumptions can be deduced in a process termed dependency directed backtracking [Stallman & Sussman 77]. All of these functions, explanation generation, incremental modification, and dependency directed backtracking are performed by special purpose algorithms in the TMS.

All of the TMS functions described above were incorporated into the original version of a TMS formulated by Jon Doyle [Doyle 78]. In addition to these functions the TMS is here viewed as an important *deductive* component of the overall system. A distinction is made between propositional deduction and the instantiation of quantified knowledge. Quantified knowledge is taken to be any knowledge which can be usefully applied to a large number of specific individuals (knowledge concerning the existence of certain types of individuals is not dealt with). The instantiation of quantified knowledge is the act of generating specific knowledge about an individual corresponding to that quantified knowledge. Often instantiation can be made simply in the hope that the resulting propositional knowledge will be useful in propositional deduction. Since the propositional reasoning mechanisms in the TMS are rather straightforward the major control issue is the control of the instantiation of quantified knowledge.

A restriction on the current version of the systems is that it is not possible to add quantified knowledge dynamically during reasoning and have it retrospectively instantiated with all the appropriate items. None of the applications described here require such an ability, but there is no theoretical reason that this limitation could not be overcome within a very similar framework to the one discussed here.

The equality system described in chapter one deals with propositions asserting equalities between "designators". My use of the word "designator" is equivalent to the use of the word "term" in standard systems of formal logic. The word "designator" is used here for no better reason than that it seems to focus ones attention on the distinction between a designator and its referent, and therefore seems more natural in a system centered around equality and the use of multiple designators. Each designator is maintained as an independent entity even though an equality may state that it has the same referent as some other designator. The equalities in the system determine equivalence classes of designators. These classes are incrementally maintained as equalities are added and removed from the system. The equality system maintains a canonical name for each equivalence class which is one of the designators in that class. At any point the user of the system can ask for the canonical name of some designators equivalence class via a "what-is" function. A substitution process is also

controlled by the equality system which essentially insures that all knowledge is represented in terms of canonical names.

The equality system can be viewed as an algorithm for controlling the instantiation of a specific body of quantified knowledge. The quantified knowledge involved is the transitivity of equality and the validity of the substitution of equals for equals. The equality system instantiates this knowledge by creating new equalities and telling the TMS that the new equalities are implied by other equalities already in the system. Of course the mechanism for controlling this instantiation process is largely determined by the set of equalities believed by the system. An important observation about this particular control of instantiation is that it is object oriented. Thus, while the system is based on an assertional data base, it simulates the behavior of systems built on object oriented data structures.

A major reasoning technique employed by electrical engineers is propagation of constraints [Steele & Sussman 78]. It is shown in chapter one that this process can be implemented as a special case of the algorithms for handling equality. The basic step in constraint propagation is a shift in the canonical name of a class induced by a substitution of terms into some designator in that class. The shift in canonical name can then induce further substitutions thus propagating values around a constraint network.

There are several mechanisms which have been developed to allow the procedural embedding of knowledge. The most straightforward example of this is the use of arithmetic operators. For example consider the designator (+ 1 2). The arithmetic operator + can be applied to 1 and 2 to give a new numerical designator, 3, which has the same referent as the sum. Thus equalities can be generated by instantiating knowledge about the functions used in designators and this knowledge can be contained in procedures attached to these functions. Algebraic simplification is done essentially the same way. A few other mechanisms for the procedural embedding and special purpose controlled instantiation within the equality system are discussed in chapter two.

Chapter three introduces electronic analysis. Knowledge about simple electronic circuits is naturally embedded in TMS predicates and propagation of constraints provides the primary reasoning mechanism. Canonical names are chosen using a "better-name" relation on designators and several important points are made about the structure of the better-name relation necessary to produce a proper reasoning process in electronics. One key observation is that referentially transparent designators must be better names than referentially opaque ones. Another important observation is that arbitrary quantities (ones that stand for an unspecified and arbitrary value such as a ground potential or an input voltage)

should be better names than referentially opaque designators which are not arbitrary (arbitrary values are not used in this system to derive quantified knowledge).

A mechanism for using multiple descriptions of parts of electronic circuits, termed "slices", has been proposed by Gerald Sussman [Sussman 77]. This mechanism has been implemented in the equality system via the creation of equivalences between the terminals of equivalent circuits. The introduction of such equivalences fits naturally into the equality framework. A restructuring of constraints which corresponds to redrawing circuit topologies is also shown to be important for propagation of constraints and the effectiveness of slices. These techniques are discussed in chapter three.

Chapter four gives some of the algorithmic details which were glossed over in the previous chapters. A description of some related previous work is also presented.

Everything described here has been fully implemented and is currently (July 79) running on both the PDP-10 and the LISP machines at the MIT AI Lab.

# Chapter I
## The TMS, Canonical Naming, and Constraints

### Using the TMS

The truth maintenance system or TMS, is a special purpose system for handling propositional deduction [Doyle 79] [McAllester 78]. The TMS used here should be thought of as a system for enforcing a set of logical relations on a set of propositions. Each proposition is represented in the TMS by a TMS node. Each such node can be in one of three possible truth states, true, false, or unknown. The logical relations in the TMS should be thought of as constraints on the truth values associated with the nodes in that relation. Each relation (constraint) is represented internally as a disjunction of terms such as (or (not P) (not Q) R). The details of the TMS algorithms are discussed further in chapter four.

When a deduction is made within the TMS a justification for that deduction is recorded. The resulting justifications are used to generate explanations for any deduced assertion when such explanations are requested by the user. These explanations give the user insight into the way a deduction was made and therefore greater confidence in the results. The justifications are far more important than this however. They allow the TMS to incrementally modify the set of deduced truth values when assumptions are retracted. When contradictions occur in the system the assumptions underlying that contradiction can be pinpointed via the justifications, and the negations of one these assumptions can be deduced to remove the contradiction. This controlled backing out from a contradiction has been termed dependency directed backtracking [Stallman & Sussman 77].

With the introduction of a special purpose propositional deduction system (the TMS) a sharp distinction is being made between propositional deduction and the instantiation of quantified knowledge. Quantified knowledge is taken to be any knowledge which can be usefully applied to a large number of specific individuals (knowledge concerning the existence of certain types of individuals is not dealt with). The instantiation of quantified knowledge is the act of generating specific knowledge about an individual corresponding to that quantified knowledge. In the present system all such specific knowledge is represented as either a logical relation among TMS nodes or as the assignment of a truth value to such a node. For example consider the quantified knowledge represented by the sentence: ∀x(mammal(x) -> warm-blooded(x)). This could be

instantiated with "Fred" to give the propositional relation: mammal(Fred) -> warm-blooded(Fred). Now if the TMS believes that the proposition mammal(Fred) is true (the node corresponding to this assertion has a truth value of "true"), it will deduce that warm-blooded(Fred) is also true. However if the TMS does not believe that Fred is a mammal, i.e. either it believes that Fred is not a mammal or it has not committed itself one way or the other, then it will simply remember the implication and use it whenever it can. The control of instantiation is the major control issue in the reasoning system.

The simplest embodiment of quantified knowledge is in "TMS predicates". These predicates are lisp functions which return a TMS node representing a proposition. For example a mammal predicate can be defined as a function of one variable which returns the node representing the proposition that the argument passed is a mammal. Quantified knowledge is embodied in such predicates in that instantiations are done as side effects the first time a predicate is applied to some set of arguments. Thus the example of quantified knowledge above which states that all mammals are warm blooded could be attached to the mammal predicate and instantiated with every object to which the mammal predicate is applied. Notice that this is not "antecedent deduction" in the classical sense as no actual deduction need be involved at all. I will call this type of instantiation "reference instantiation" since the quantified knowledge is instantiated at the point reference is made to a certain entity (such as a proposition representing the application of the mammal predicate). A special form has been defined for the creation of TMS predicates and some examples of TMS predicate definitions are given below:

```
(defpred vertebrate (x))

(defpred flys (x))

(defpred bird (x)
        (vertebrate x)
        (flys x))
```

The body of a defpred must be a list of forms which return TMS nodes when evaluated. The TMS nodes thus created will be implied by the truth of the node returned by an application of the predicate being defined. Thus (bird 'John) returns a TMS node which implies (vertebrate 'John) and (flys 'John).

There are various TMS predicates which are supplied as primitives. The predor and predand predicates take any number of TMS nodes and return a node

which has been appropriately constrained in the TMS. The -> predicate takes two nodes and returns a TMS node which implies an implication relation between them. Prednot returns a node which is constrained to be equivalent to the negation of the node it is passed. Finally presumably returns a node representing the assertion that the node it was passed should be assumed to be true. This interpretation is implemented by using the -> predicate to create an assertion of the form: (-> (presumably P) P). The node representing this implication is made true as an assumption which can be retracted if contradictions arise. To see how these primitive predicates can be used consider the following definition of a mammal predicate:

```
(defpred hairy (x))
(defpred female (x))
(defpred bears-live-young (x))

(defpred mammal (x)
         (vertebrate x)
         (presumably (hairy x))
         (presumably (-> (female x)
                         (bears-live-young x))))
```

There is a TMS primitive for setting the truth value of TMS nodes which can be used to declare that some object is a mammal as follows:

```
(set-truth (mammal 'Joe) 'true 'premise)
```

or equivalently

```
(assert (mammal 'joe))
```

There is also a truth function which gives the current truth value associated with a TMS node. Thus (truth (mammal 'Joe)) would evaluate to true after the above assertion had been made. In other circumstances it might evaluate to false, or unknown. There is also a why primitive which takes a TMS node and gives an explanation for its truth value in terms of other truth values of TMS nodes which imply that value.

The use of TMS predicates as a knowledge representation mechanism results in comprehensible modular constructs which interface cleanly to a truth

maintenance system.

## Canonical Naming

The central theme of this document is equality. A TMS predicate has been defined, == which takes two designators and returns a TMS node representing the fact that the two designators have the same referent. As was mentioned earlier my use of the word "designator" is equivalent to the use of the word "term" in systems of formal logic. The word "designator" is used here for no better reason than that it focuses on the distinction between a term and its referent. The == predicate interprets lisp s-expressions as designators and designators will be uniformly represented by lisp s-expressions throughout.

One basic problem with the existence of more than one designator for an individual object is that knowledge about that individual is often only given in terms of a single designator. For example consider the situation defined by the following assertions:

```
(assert (== '(residence the-US-president)
            'the-white-house))

(assert (== 'Jimmy-Carter 'the-US-president))
```

Now suppose we ask for the truth of an equality between (residence Jimmy-Carter) and the-white-house. We would like the system to see that since Jimmy Carter is the U.S. president, his residence is the white house. There must be some way of merging the knowledge about a single object or individual which is given in terms of the various designators for that object or individual. This merging of knowledge can be accomplished via a canonical name for each individual. Any knowledge about an object can be stated in terms of its canonical name by substituting canonical names into expressions. This allows knowledge to interact with other knowledge which is similarly stated in terms of the canonical name.

The canonical name strategy for handling equality has been used in many previous problem solvers. The primary innovation here is the use of this mechanism as a primary reasoning strategy as will be discussed in the next few sections. In the present system only knowledge in the form of equalities fully benefits from canonical naming since substitutions are only performed on designators and not on TMS assertions. The details of the substitution process

will be given in a later section; the maintenance of canonical names being concentrated on here.

The == predicate creates auxiliary data structures which are used in maintaining canonical names for objects. A set of equalities defines an equivalence relation on designators. Since all of the designators in an equivalence class have the same referent, one of those designators can be chosen as the canonical name for that referent. Only equalities whose TMS nodes are true are used in this process and the canonical names are incrementally maintained when the truth values associated with equalities change. There is a "what-is" function which takes a designator and returns the canonical name for that designator to the user. This function is surprisingly useful, as will be seen in the next few sections.

### Algebraic Unknowns

Designators, and the concepts involved in equality are related to the concepts involved in the use of algebraic unknowns. Notice that algebraic unknowns can be separated from bound variables and arbitrary individuals which can be involved in instantiation processes. The ways in which algebraic unknowns take on values is entirely independent of such instantiation processes. Consider a system of n independent linear equations in m unknowns. If there are more unknowns than there are equations, then there are several degrees of freedom left in the system. However, it is possible to "eliminate" n of the unknowns by expressing each of them as a linear combination of the others. The "value" of each of these eliminated unknowns could then be set to the corresponding linear combination of the remaining unknowns.

It is possible to separate the issue of unknown elimination from algebraic manipulation. Algebraic manipulation is the deduction of new equalities from given equalities. For example, from the equality y = (+ (* a x) b) it is possible to deduce (* a x) = (- y b). Neither of these equations need have anything to do with the elimination of unknowns; they are simply assertions of equality. I will attempt here to relate the notion of an algebraic unknown, the act of giving unknowns values, and the use of equality, via a general notion of canonical naming.

Under the view of unknowns presented here, they are a type of designator. I am not immediately concerned with which designators qualify as unknowns but will deal only with the general notion of a designator. In light of the above discussion of canonical names the act of giving an unknown a value can be viewed as the act of choosing a designator, other than that unknown, as

the canonical name for the unknown's class. For example if y and (+ (* a x) b) are in the same class, i.e. y = (+ (* a x) b), and for some reason (+ (* a x) b) is chosen as a better designator than y to represent the referent of the class, then y can be said to have the value (+ (* a x) b).

The next few sections will develop propagation of constraints as an algorithm for solving systems of equations. It is shown that this algorithm can be viewed as a process of shifting canonical names.

### Constraint Propagation as Computation via Naming.

Lately there has been much interest in, and development of, constraint propagation as an efficient algorithm for reasoning about sets of mutually constrained quantities [Steele & Sussman 78]. This algorithm is also interesting in that it seems to simulate one of the ways human engineers reason. In general a constraint propagation system has a set of "cells" which can take on values, and a set of "constraints" which constrain those values. In constraint propagation whenever a deduction can be made from the previously determined values and a *single* constraint this deduction is made. Each such deduction assigns a new value to a cell. In what will be termed "simple" constraint propagation these are the only deductions which are made. Constraint propagation terminates when there are no deductions which can be made from the cell values and a *single* constraint. It is easy to see that such a deduction process can take no longer than linear time in the number of constraints.

A method of implementing constraint propagation is presented here which is based entirely on the algorithms for handling equality discussed above. The "cells" of traditional constraint propagation are implemented as equivalence classes of designators which take on "values" in the form of canonical names for those classes. Each constraint is implemented as a set of equalities, and the primary deductive mechanism employed is substitution controlled by a method of choosing the canonical names of equivalence classes. The constraint propagation techniques developed here are relied upon heavily in an electrical circuit analysis system described in a later chapter.

Constraints are created via constraint predicates. The basic constraint predicates are +constrained and *constrained, which are defined as follows:

```
(defpred +constrained (sum a1 a2)
         (== sum '(+ ,a1 ,a2))
         (== a1 '(- ,sum ,a2))
         (== a2 '(- ,sum ,a1)))

(defpred *constrained (prod m1 m2)
         (== prod '(* ,m1 ,m2))
         (-> (prednot (== m2 0))
             (== m1 '(// ,prod ,m2)))
         (-> (prednot (== m1 0))
             (== m2 '(// ,prod ,m1)))
         (presumably (prednot (== m1 0)))
         (presumably (prednot (== m2 0))))
```

The backquote macro is used to simplify the creation of designator s-expressions. The backquote macro is a form of quote in which items in the interior of the backquoted structure preceded by a comma are replaced by their value. Thus '(+ ,x ,y) is equivalent to (list '+ x y).

Note the use of the "presumably" predicate. If a *constrained is in force (its TMS node is true) then the node representing (== m1 0) will default to false. However if this leads to a contradiction (m1 is discovered to be 0), then the support for m1 not being 0 given via the default construct can be automatically retracted by the TMS.

Using the constraint predicates defined above, it is possible to construct constraint networks within the equality system. Given an appropriate method of deciding which designator of two given designators is "better" for use as a canonical name, the constraint nets formed in this way can interact with the canonical equality system to actually propagate values. To decide when one designator is a better designator than another it is necessary to define a partial order on designators which will be referred to here as the better-name relation. Two designators can be unordered by the better-name relation and in that case whichever becomes the canonical name first will remain so, thus avoiding undue substitution.

In this discussion of constraint propagation a better-name relation is assumed in which designators are divided into two classes, "known" and "unknown". For atomic designators a predicate is provided to determine whether the designator is known or unknown, and a functional expression is known iff its operator and all of its arguments are known. Known designators are always better than unknown designators. This convention along with substitution is

enough to give propagation of constraints.

The two methods by which designators are generated automatically in the equality system are substitution and evaluation. Only substitution will be considered here; evaluation (which is the way algebraic simplification gets done) will be discussed later. Whenever a new canonical name is assigned to a class that canonical name is substituted into designators which contain other members of that class. Such substitutions generate new designators from a pre-existing ones. Whenever this is done an equality is created between the original designator and the one resulting from the substitution. The truth of this equality is implied by the truth of the equalities used in the substitution. A more detailed description of the implementation of this process will be given in chapter III. Now consider the following constraint net taken from [Steele & Sussman 78]:

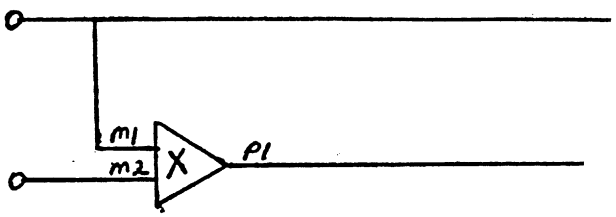



Figure 2. A Simple Constraint Network.

This constraint net can be created in the equality system by the following assertions:

```
(assert (*constrained 'p1 'm1 'm2))
(assert (+constrained 's1 'a1 'a2))
(assert (== 'p1 'a1))
(assert (== 'a2 'm1))
```

The initial equivalence classes are derived from the equalities and a canonical name is chosen for each class. I will assume that all designators used so far are classified as unknown and that, all other things being equal, smaller designators are a better choice for a canonical name than larger, more complex ones. The equivalence classes defined so far and their (somewhat arbitrary) canonical names are:

```
m2, (// p1 m1)                          m2
m1, (// p1 m2), a2, (- s1 a1)           m1
p1, (* m1 m2), a1, (- s1 a2)            p1
s1, (+ a1 a2)                           s1
```

The substitution process would add several equalities which are shown below (each equality has been associated with the canonical name for the class which contains its arguments).

```
(== '(+ a1 a2) '(+ p1 m1))              s1
(== '(- s1 a1) '(- s1 p1))              m1
(== '(- s1 a2) '(- s1 m1))              p1
```

Since none of the generated designators are known, the original canonical naming is stable and no propagation occurs. Now suppose however that s1 is made equal to 4 and a1 is made equal to 2. 4 and 2 are assumed to be known and therefore the canonical designators of the classes corresponding to s1 and a1 are changed to 4 and 2 respectively. Substitution then generates the following equalities:

```
(== '(+ p1 m1) '(+ 2 m1))               4
(== '(- s1 m1) '(- 4 m1))               2
(== '(- s1 p1) '(- 4 2))                m1
(== '(// p1 m1) '(// 2 m1))             m2
(== '(// p1 m2) '(// 2 m2))             m1
```

As the substitution process was described above, other substitutions would also be made (into (+ a1 a2) for example). However these designators have already been substituted into. The designator generated by the earlier substitution is equal in all its parts to the original designator, and can be further substituted into, making substitution into the original designator redundant. The details of the way in which the system avoids redundant substitutions will be given in chapter IV.

Given the above equalities a2 has been made equal to (- 4 2) since it was originally equal to (- s1 a1). Since (- 4 2) is a known designator it must become the canonical name of a2's class. This change in canonical name then leads to the following further equalities via substitution:

```
(== '(+ 2 m1) '(+ 2 (- 4 2)))          4
(== '(- 4 m1) '(- 4 (- 4 2)))          2
(== '(* m1 m2) '(* (- 4 2) m2))        2
(== '(// p1 m1) '(// 2 (- 4 2)))       m2
```

Now notice that m2 has been made equal to (// 2 (- 4 2)). Since the new value is known it becomes the canonical name and further substitution leads to no changes in canonical names. Let the "value" relation x=>y be defined as meaning y is the canonical name of the equivalence class of x. The results of the above propagation can be summarized by the following relations:

```
a1      =>      2
p1      =>      2
s1      =>      4
a2      =>      (- 4 2)
m1      =>      (- 4 2)
m2      =>      (// 2 (- 4 2))
```

If m2 and s1 had been specified instead of a1 and s1, no propagation would have occurred even though all values are completely determined. Such situations are discussed by Steele and Sussman and are due to loops in the constraint net. One technique which has been successfully used to handle involves propagating symbolic values which are classified as known, just as numeric values are propagated. This technique, called plunking, is discussed later in its own subsection.

A general feeling for the efficiency of this algorithm can be gotten by considering the number of substitutions performed. As mentioned above once a new designator has been derived via substitution the designator which was substituted into will not be substituted into again (assuming no equalities are retracted which invalidate the original substitution). Therefore each designator gives rise to a string of descendants generated by substitution, each descendant generating the next. The number of such descendants for a given designator can be no larger than the sum of the number of times the top level subdesignators change canonical names. The number of top level subdesignators is one plus the arity of the function involved, and is assumed quite small. The number of times the canonical name of a class changes can also be assumed to be quite small in practice, since it is usually due to a shift from an unknown to a known designator. Thus the total number of substitutions is bounded by a small constant times the number of original designators in the constraint net. The

evaluation of designators, which has not been discussed yet, can generate designators in ways not involving substitution. These other designators expand the set of names which are substituted into and can make constraint propagation somewhat more expensive.

It should be emphasized that the propagation process is really driven by the decisions about which designators are better canonical names. In chapter III the discussion of electronic circuit analysis will require a more elaborate better name relation which relies on a destinction between referentially transparent and referentially opaque designators.

# Chapter II
## Procedural Embedding

The three sections of this chapter deal with various ways in which procedures have been embedded in the equality system and the ways in which they are used. The first section introduces the major mechanism used in procedural embedding and shows how simple integer arithmetic can be done using this mechanism. The next section discusses some problems surrounding equalities which should have false truth values, such as equalities between distinct integers. The final section deals with the problem of solving for values which are determined by a set of constraints but can not be solved for directly via propagation of constraints.

### Evaluation Functions

Often quantified knowledge can be associated with a specific function or operator. Knowledge about the properties of algebraic operators is an example of this type of knowledge. To facilitate the control of the instantiation of such knowledge a special mechanism involving "evaluation functions" has been incorporated into the equality system. This section will first develop the general mechanisms involved in evaluation functions and then turn to applications involving algebraic operators.

Consider the knowledge that the mother of every animal is female. This information is associated with the mother function since it only applies to applications of that function. Formally such knowledge would be represented as ∀x(-> (animal x) (female (mother x))) which could then be instantiated with any term. Knowledge concerning a certain operator can be attached to that operator in the equality system in the form of evaluation functions. Each evaluation function for a given operator can be applied to any designator representing an application of that operator and performs an instantiation of the quantified knowledge it contains. The above knowledge about the mother function could be embedded in the following evaluation function:

```
(defpred animal (x))
(defpred female (x))

(defun mother-eval-1 (designator-s-expression)
   (if (eq (car designator-s-expression) 'mother)
       (let ((x (cadr designator-s-expression)))
            (assert (-> (animal x) (female designator-s-expression))))))
```

This function must be appropriately associated with the mother operator in the equality system for it to be useful. To simplify things a special mechanism has been created to create such functions and associate them with operators. Using this mechanism the above function could have been both defined and associated with the mother operator as follows:

```
(evalfun mother (x)
         (-> (animal x) (female '(mother ,x))))
```

As in the case of a defpred, the body of a evalfun is a list of forms which return TMS nodes when evaluated. The function created by a evalfun evaluates the forms of the body in an environment in which the argument list of the evalfun has been bound to the appropriate parts of a designator representing an application of the involved operator. The nodes created by the forms of the body are asserted as premises.

There can be more than one evaluation function for a given operator. For example the knowledge that the mother of an animal is also a parent of that animal could be represented in an evaluation function as:

```
(defpred parent (x y))

(evalfun mother (x)
         (parent '(mother ,x) x))
```

The control of the application of evaluation functions (and therefore the control of the instantiation of the quantified knowledge they contain) will be discussed in detail in chapter four. For now however there is no problem in assuming that it is simple reference instantiation, that is to say that the quantified knowledge is instantiated (i.e. the evaluation functions are applied) whenever reference is made to some application of the involved operator.

This mechanism can be used to achieve simplification of numerical

expressions in the equality system. Consider the following:

```
(defun all-numbers (things)
       (or (null things)
           (and (numberp (car things))
                (all-numbers (cdr things)))))

(evalfun + addends
        (if (all-numbers addends)
            (== '(+ ,@addends) (apply 'plus addends))))

(evalfun - args
        (if (all-numbers addends)
            (== '(- ,@args) '(apply 'minus args))))

        etc.
```

Here a slightly different syntax has been used to handle a variable number of arguments in which the argument list has been replaced by a single atom which is bound to a list of the designators representing the arguments of the operator. Also the form in the body of the evalfun can return nil, in which case nothing is done (this is also true in the body of a defpred). The use of ",@" in the interior of a backquote explodes the value of the form following (which must be a list) up into the list in which the ",@" appears. For example '(a ,(list 'a 'b) ,@(list 'a 'b)) evaluates to (a (a b) a b).

Using these definitions, evaluation of the designator (- 4 2) would result in an equality being added between it and 2. It is up to the better-name relation to determine that 2 is a better designator for the canonical name of the equivalence class than (- 4 2). If 2 becomes the canonical name of its class, then it will be substituted into other designators such as (// 2 (- 4 2)). The results of such substitutions can then be evaluated to yield further simplifications. The definition of an evaluation function for // must be careful not to state false equalities when two integers do not divide evenly.

It should be clear from the way the LISP plus and minus functions were used above that arbitrary procedures can readily be embedded into the evaluation functions. Special purpose algebraic simplification routines are embedded in the equality system in exactly this way. Thus an evaluation functions for +, -, etc. have been created which are useful when the arguments to the operator are symbolic. These evaluation functions create an equality between the expression

being evaluated and a symbolically simplified (or standardized) form of it. The specific nature of the algebraic simplification algorithms used is discussed in appendix two.

## Disequalities

As was mentioned in the section on constraint propagation, it is possible that a set of constraints determine values for designators, but that constraint propagation does not result in those values being found. This situation results when loops are present in the constraint net and is illustrated by the following example:

```
(*constrained 'y 2 'x)  (y = 2x)
(+constrained 3 'x 'y)  (x+y = 3)
```

Notice that even though the value of x and y are determined by the constraints, no constraint propagation occurs since all of the designators for x and y contain either x or y and are therefore classified as unknown (assuming x and y are classified as unknown). How the equality system actually finds values which are determined but not deduced via constraint propagation is the subject of the next section. This section is only concerned with the possible contradictions that can arise in these situations.

Suppose that an equality between x and 3 is added to the above constraints. Propagation would occur and the system would deduce that y equals (* 2 3) by substituting into (* 2 x). It would also deduce that y equals (- 3 3) by substituting into (- 3 x). The evaluation functions attached to the operators would then put both 0 and 6 into the equivalence class for y. Nothing in the system so far says that this is impossible (the names being used for numbers might be representing numbers in modular arithmetic, and indeed 0 does equal 6 if they refer to equivalence classes mod 3).

To rule out such interpretations of numeric designators in the equality system a special check for numeric designators has been placed in the == predicate. Where the designators passed to == are distinct numbers == insures that the node it returns to represent the equality is false. The equality system internally creates equality nodes between every designator in an equivalence class and the canonical name of that class. Thus if a numerical designator is the canonical name for some class the entrance of any other numerical designator into that class will result in a contradiction via the equality node which would be created between the two numerical designators.

This technique could easily be generalized to a method for ensuring the integrity of other disequalities. Certain designators could be called "truely canonical" designators in that a "truely canonical" designator must always be the canonical name for its equivalence class. Thus no two "truely canonical" designators could ever be equal since then both of them would have to be the canonical name of the same class. Therefore whenever an equality assertion is created between two "truely canonical" designators the system could force that equality to be false. The above method for handling numbers would then be implemented by simply making numeric designators "truely canonical", but the method is certainly not restricted to numbers and could be used to ensure that, for example, Earth is not the same planet as Mars.

### Plunks

One method for finding values which are determined but not found via constraint propogation is referred to as "plunking" and is discussed by Stallman and Sussman [Stallman & Sussman 77]. In their system plunking involves giving a variable a symbolic value which propagates through the network just as any numeric value would. Certain equalities generated during this propagation can then be passed to an equation solver which solves for the plunks in terms of other quantities.

The algorithm discussed here is slightly different from the one used in their constraint language due to the difference in the representation of the constraints. In the equality system a class which contains only unknown designators is chosen to be plunked. A new unique designator is generated to represent a symbolic value, which will be called a plunk. This designator is then made equal to some designator in the plunked class. For the plunking to work properly the better name relation must treat the plunks specially. A plunk is treated as a known designator, and it induces propagation. However designators containing plunks are considered worse names than other known designators so that designators with plunks are replaced by designators without them if the opportunity arises. The better name relation as it has been described so far divides designators into three classes and orders those classes as follows:

| | |
|---|---|
| | known designator without plunks |
| are better than: | known designators with plunks |
| are better than: | unknown designators |

Now consider the constraint net described in the previous section:

$$y = 2x$$
$$x+y = 3$$

In this constraint net it is possible to plunk x by adding an equality between x and a designator generated to represent the plunk, call it plunk-1. The canonical name for x's equivalence class would shift to plunk-1 which would in turn induce substitutions. y is equal to (* 2 x) which generates (* 2 plunk-1) under the substitution. Since this later designator is treated as known the canonical name for y's class becomes (* 2 plunk-1). But y is also equal to (+ x 1) which generates (+ plunk-1 1) when substituted into.

To solve for the plunks the notion of a coincidence has been adapted from previous constraint systems to use in the equality system. A coincidence occurs when a "known" designator enters a class whose canonical name was already a "known" designator. In such cases the pair of known designators is passed to a coincidence handler. The coincidence handler checks to see if they are simply substitution variants of each other and if not it then checks to see if either contain a plunk. If a plunk is present then the coincidence handler attempts to use the coincidence to solve for a plunk. In the above example the coincidence handler would be passed (* 2 plunk-1) and (+ plunk-1 1). Since the pair of designators which are passed to the coincidence handler are equal it can solve this equality for plunk-1 using a solver from the algebraic simplification system. The coincidence would now add an equality between plunk-1 and 1. Since known designators which do not contain plunks are better than known designators containing plunks, 1 would become the canonical name for plunk-1's equivalence class, and 1 would be substituted for plunk-1 wherever it occurred.

In general more than one plunk may be necessary to solve a set of constraints. In that case the designators which are passed to the coincidence handler may contain more than one plunk. To handle such situations each plunk is given a plunk-weight which is used by the better-name relation. A lower weight plunk is always considered a better name than a higher weight plunk, and the plunk-weight of a designator containing no plunks is considered to be zero. The plunk-weight of designators containing plunks is the maximum weight of the plunks in the designator. Now when the coincidence handler gets expressions which contain more than one plunk it solves for the highest weight plunk in terms of the others. Since any designator containing only the other plunks will have a lower plunk weight, the canonical name of the solved for plunk must shift, and therefore that plunk will be replaced wherever it occurs.

Plunking is not performed automatically in the present equality system. To invoke plunking a `solve-for` function has been defined which takes a single designator and plunks quantities which that designator can be expressed in terms of. The net effect is that the argument passed to `solve-for` is made equal to a "known" expression which may contain plunks. If plunks are present in the resulting expression the `solve-for` function can be used again on the plunks themselves. This can be done iteratively to solve for in expression which is indeed determined by a set of constraints. The following dialogue shows how this procedure can be applied to the above constraint net.

```
(assert (*constrained 'y 2 'x))
(*CONSTRAINED 'Y 2. 'X)

(Assert (+constrained 3 'x 'y))
(+CONSTRAINED 3. 'X 'Y)

(what-is 'x)
X

(what-is 'y)
Y

(solve-for 'x)
1

(what-is 'y)
2
```

The next chapter applies the mechanisms which have been developed so far to the domain of electrical circuit analysis.

# Chapter III
## Application to Electronics

In this section a system for analyzing electronic circuits is presented which has been implemented using the equality system. Only a minimum of knowledge about electronics is necessary for the comprehension of this section and the predicates defined here contain all the electronics knowledge used. Since many general properties of the equality system are exhibited, it is hoped that even readers with no background in electronics will take the time to read this section.

A set of basic electronic predicates are defined, which can be used to build arbitrary circuits. However, for the constraint propagation to work properly in reasoning about circuits defined with these predicates, a better-name relation must be used which is slightly more complex than those discussed so far. A distinction is made between referentially transparent and referentially opaque designators, the former being a better name than the latter. The need for this distinction, and its use, is discussed in the context of circuit analysis.

### Basic Predicates

The predicates which are defined below are used to create some of the constraints associated with various device types which the electrical analysis system deals with. Predicates are also given which are used to wire the device components together. The predicate c== has been written in lisp as a special constraint predicate. c== takes two designators and solves the implied equation for each internal designator in the two expressions. This results in a set of equalities which are all implied by a node returned by c==. Appropriate quantities are assumed not to be equal to zero as was done in *constrained. (c== 'a '(* b c)) is equivalent to (*constrained 'a 'b 'c) but less efficient.

The use of >> in designators gives a shorthand for the repeated functional composition of monadic functions. Thus (>> voltage r1 circuit3) is treated identically to (voltage (r1 circuit3)). This is often very convenient when dealing with complex structured objects.

```
(defpred inv-constrained (a b)
  (== a '(- ,b))
  (== b '(- ,a)))
```

```
(defpred one-port (element)
         (+constrained '(>> potential t1 ,element)
                       '(>> potential t2 ,element)
                       '(voltage ,element))
         (== '(current ,element)
             '(>> current t1 ,element))
         (inv-constrained '(>> current t1 ,element) '(current t2 ,element)))

(defpred resistor (r)
         (one-port r)
         (*constrained '(voltage ,r) '(current ,r) '(resistance ,r)))

(defpred voltage-source (vs)
         (one-port vs)
         (== '(voltage ,vs) '(strength ,vs)))

(defpred current-source (cs)
         (one-port cs)
         (== '(current ,cs) '(strength ,cs)))
```

There are no definitions for controlled sources given here. This is because a controlled source can be simply represented as a one-port with an appropriate current or voltage constraint.

The most general way to handle wiring is to write a "node" predicate which takes any number of terminals and creates equalities stating that that the potentials of all the terminals are the same and that the sum of the currents is 0. This predicate definition uses a slightly different syntax since it is a predicate on an arbitrary number of arguments. The atom representing its argument list is bound to a list of the values of the arguments in a call to the predicate. The two other wiring predicates given are a little more natural to use than the node predicate.

```
(defpred node terminals
         (c== 0 '(+ ,@(mapcar '(lambda (term) '(current ,term)) terms)))
         (let ((node-potential '(potential ,(car terms))))
           (predand (mapcar '(lambda (term2)
                               (== node-potential '(potential ,term1)))
                            (cdr terms)))))
```

```
(defpred connected (t1 t2)
        (== '(potential ,t1) '(potential ,t2))
        (== '(current ,t1) '(- (current ,t2))))

(defpred exists (x))

(evalfun composite (t1 t2)
        (-> (exists '(composite ,t1 ,t2))
            (predand (== '(potential ,t1) '(potential ,t2))
                     (== '(potential (composite ,t1 ,t2)) '(potential ,t1))
                     (c== '(current (composite ,t1 ,t2))
                          '(+ (current ,t1) (current ,t2)))))
        (presumably (exists '(composite ,t1 ,t2))))
```

These predicates can be used to create circuits. As a trivial first example a circuit which deals only with a single voltage source and resistor is defined below and shown in figure 3.

```
(defpred ohm-test (circuit)
        (voltage-source '(vs ,circuit))
        (resistor '(r ,circuit))
        (node '(>> t1 vs ,circuit) '(>> t1 r ,circuit))
        (node '(>> t2 vs ,circuit) '(>> t2 r ,circuit)))
```

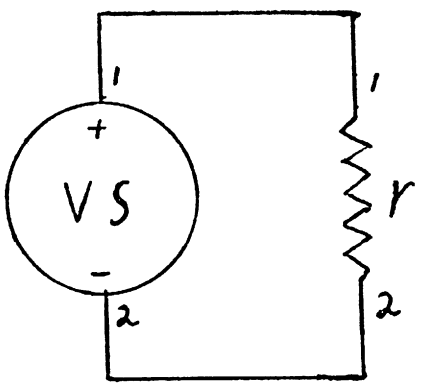



Figure 3. The "Ohm-test" Circuit.

A particular instance of this circuit is created by asserting an application of the above predicate to a specific circuit name. The following scenario shows a simple use of this predicate.

```
(assert (ohm-test 'c1))
(OHM-TEST C1)

(why (one-port '(r C1)))
((ONE-PORT (R C1)) IS TRUE FROM
 (1 (RESISTOR (R C1)) IS TRUE))

(why (resistor '(r c1)))
((ONE-RESISTOR (R C1)) IS TRUE FROM
 (1 (OHM-TEST C1) IS TRUE))
```

This method of circuit definition, while quite clean and semantically satisfying has the disadvantage that the circuit topology of an instance of the circuit definition can not be incrementally altered because the assertion that the circuit is an ohm-test completely determines the structure of the circuit. Such incremental alterations would be possible if the various assertions about the circuit were taken as independent premises as was done by Stallman and Sussman in their electronic analysis system ARS [Stallman & Sussman 77]. This can be done by using a simple lisp function to construct the circuit which might be defined as follows:

```
(defun make-ohm-test (circuit)
       (assert (voltage-source '(vs ,circuit)))
       (assert (resistor '(r ,circuit)))
       (assert (node '(>> t1 vs ,circuit) '(>> t1 r ,circuit)))
       (assert (node '(>> t2 vs ,circuit) '(>> t2 r ,circuit)))
       t)
MAKE-OHM-TEST

(make-ohm-test 'c1)
T

(why (restistor '(r c1)))
((RESISTOR (R C1)) IS TRUE AS A PREMISE)
```

Now it is possible to change the topology of this circuit by incrementally altering the basic premises which determining that topology. For example the following scenario shows how another resistor could be spliced into the circuit:

```
(assert (resistor '(r2 c1)))
(RESISTOR R2)

(retract (node '(>> t1 vs c1) '(t1 r c1)))
T

(assert (node '(>> t1 vs c1) '(>> t1 r2 c1)))
(NODE (>> T1 VS C1) (>> T1 R2 C1))

(assert (node '(>> t2 r2 c1) '(>> t1 r c1)))
(NODE (>> T2 R2 C1) (>> T1 R C1))
```

In the discussions of electronics that follows, circuits will be defined via the defpred mechanism but the reader should be aware that circuits which can be incrementally modified can be constructed and that derived knowledge about such circuits is also incrementally modified when changes are made.

### The Better-Name Relation in Electronics

The primary reasoning strategy used in the electronic analysis system described here is propagation of constraints. As propagation of constraints was defined above it is driven by the better-name relation, that is the ordering on designators which determines when one designator is preferred over another as the canonical representative for an equivalence class of designators. Here an investigation is made into the properties a better-name relation should have such that constraint propagation is done in a useful manner.

Consider the simple circuit defined in the previous section and shown in figure 3. A trivial problem in circuit analysis would be to make an ohm-test circuit, call it c1, and ask for the current in the resistor. The current in the resistor has many designators, the simplest of which is (>> current r c1). However the designator which we would like to get as a value for the current is the ratio of the strength of the voltage source to the resistance of the resistor. The better name relation should be such that designators in terms of source strengths and component parameters are better names that those in terms of

currents and voltages.

An interesting observation is that designators involving voltages and currents are sensitive to the context in which they occur. For example the voltage across a certain resistor is not purely a function of that resistor but is sensitive to the environment of the resistor. A designator whose referent is context sensitive is termed referentially opaque, otherwise it is termed referentially transparent. It seems that referentially transparent designators are better names, at least in the context of electronic analysis. The equality system presently perfoms a simple syntactic search for `potential`, `current`, or `voltage` to determine if a designator is opaque or transparent.

Numeric designators should be better names than transparent but non-numeric ones. Thus a numeric value for a component parameter is a better-name than a designator in terms of the parameter function applied to that component. If plunks are present then they are considered to be transparent so that propagation treats them properly. The classification of designators by the better name relation is now as follows:

```
                    numerical designators
are better than:    transparent designators without plunks
are better than:    transparent designators with plunks
are better than:    opaque designators
```

The better-name relation for the electronics system also takes the designator size into account when the designators are of the same type, smaller designators being better. Given this framework it is now possible to see how the equality system analyzes circuits.

## A Simple Analysis

The following simple dialogue represents an analysis of an instance of the simple ohm-test circuit defined above. In all the sample dialogs presented here things typed by the user appear in lower case while responses typed by the system appear in upper case.

```
(assert (ohm-test 'c1))
T
(what-is '(>> current r c1))
(// (>> STRENGTH VS C1) (>> RESISTANCE R C1))
```

To see how this analysis took place it is necessary to examine the original equalities which were provided to the system. Among these equalities are the following four:

```
(== '(>> potential t1 r c1)
    '(>> potential t1 vs c1))
(== '(>> potential t2 r c1)
    '(>> potential t2 vs c1))
(== '(>> voltage vs c1)
    '(- (potential t1 vs c1)
        (potential t2 vs c1)))
(== '(>> voltage r c1)
    '(- (potential t1 r c1)
        (potential t2 r c1)))
```

The first two equalities are created by the node predicate. A canonical name will be chosen for the potential of each node. Since the alternatives for these canonical names are roughly equivalent in their type and complexity, the choice is arbitrary, but let us assume that the canonical names are given in terms of the voltage source. So (>> potential t1 vs c1) is the canonical name for (>> potential t1 r c1) and similarly for the other potential. This choice of canonical names results in the following equality being generated via substitution:

```
(== '(- (>> potential t1 r c1) (>> potential t2 r c1))
    '(- (>> potential t1 vs c1) (>> potential t2 vs c1)))
```

This equality interacts with the last two of the first four equalities above making the voltage of the voltage source equal to the voltage of the resistor. By the definition of a voltage source, the voltage of the source is equal to the strength of the source which has a referentially transparent designator. The designator (>> strength vs c1) therefore becomes the canonical name for the voltage of both the voltage source and the resistor. Now consider a result of Ohm's law which was generated as a part of the constraints in the definition of a resistor:

```
(== '(>> current r c1)
    '(// (voltage r c1) (resistance r c1)))
```

Substitution will yield the equality:

```
(== '(// (voltage r c1) (resistance r c1))
    '(// (strength vs c1) (resistance r c1)))
```

The ratio of the strength of vs to the resistance of r is a transparent designator, and therefore becomes the canonical name for the current in the resistor and the analysis is complete. Further discussions about circuit analysis will be a little less formal about the actions of the equality system. Instead of referring to the addition of a referentially transparent designator to an equivalence class, it will simply be said that designators in that class have been determined. Thus, instead of saying that a referentially transparent canonical name for the current in r has been found, I will simply say that the current in r has been determined.

### Unconstrained Plunks

There are many circuits which cannot be analyzed using the definitions given so far without resorting to plunks. In many circuits the need for plunks is quite surprising since human engineers can easily analyze the circuit by inspection. The last three sections in this chapter discuss ways in which the electrical analysis system can be made more powerful. In this section the notion of an "unconstrained plunk" is developed for use in situations in which there are unconstrained degrees of freedom left in the quantities being reasoned about. The next section discusses some of the ways in which multiple descriptions can be used to aid reasoning in constraint propagation. The final section discusses the effect of restructuring constraints in ways which correspond to "redrawing" circuits.

As a first example of a circuit in which constraint propagation alone fails to perform the desired analysis, consider the circuit shown in figure 4. When an instance of this circuit is analyzed the current in each resistor is determined by the current through the current source. The current in each resistor determines the voltage across that resistor so it would seem that the voltage across the current source should also be determined by the system. This however does not happen since the voltage across the source is equal to the difference between the potentials of its two terminals, and neither of these potentials can be determined. Nor is the difference determined directly since

Figure 4. The "Series-1" Circuit.

```
(defpred series-1 (circuit)
         (current-source '(cs ,circuit))
         (resistor '(r1 ,circuit))
         (resistor '(r2 ,circuit))
         (series-1-topology '(cs ,circuit) '(r1 ,circuit) '(r2 ,circuit)))

(defpred series-1-topology (cs r1 r2)
         (node '(t1 ,cs) '(t1 ,r1))
         (node '(t2 ,r1) '(t1 ,r2))
         (node '(t2 ,r2) '(t2 ,cs)))
```

there is no statement that the voltage across the source is the sum of the voltages across the resistors. If there were some way in which potentials could be determined, instead of potentials differences, this problem could be solved. However there is a degree of freedom left undetermined in the node potentials and none of these potentials can be solved for.

In the framework of circuit analysis presented so far the above circuit could be analyzed via plunking. In such a scheme some node potential could be "plunked" by setting it equal to a plunk designator. Propagation would then take place and each node potential would take on a canonical name in terms of the

plunk. The voltage across the source would then be determined in terms of the plunk, which would cancel out of the resulting expression. Previous electrical analysis programs which relied on propagation of constraints have used plunking in this way [Stallman & Sussman 77] [de Kleer & Sussman 78].

In the use of a plunk to analyze the above circuit the plunk cannot be eliminated since the absolute values of the node potentials are not determined. Plunks which can never be eliminated will be called "unconstrained plunks". The special treatment the system gives to ordinary plunks by the better-name relation and the coincidence handler is designed to ensure that plunks can be eliminated. Since an unconstrained plunk cannot be eliminated, it need not be given this special treatment. In fact no distinction need be made whatsoever between this type of plunk and ordinary referentially transparent designators. However, since no attempt is made by the system to solve for unconstrained plunks, it is important that such unconstrained plunks be mutually independent, that is to say that no one of them is expressible in terms of the others.

Unfortunately the fact that a plunk can never be eliminated is an observation made outside the system (the equality system is not capable of realizing this simply from a given set of equalities). However since an unconstrained plunk can be treated exactly as a simple transparent designator the "plunk" can be made explicitly by the user of the equality system simply by stating an equality between the unconstrained quantity to be plunked and a referentially transparent designator which will act as the plunk. This gives the user more control over the reasoning process by explicitly stating a quantity to be plunked. The choice of which quantity is given an unconstrained plunk in a system of mutually constrained quantities can also be important to the human user who wishes to see results in terms of certain quantities.

In all electronic circuits there is at least one degree of freedom in the node potentials which is not determined by the circuit. This degree of freedom can be dealt with by defining a reference node to be given an unconstrained plunk. For convenience the node given the unconstrained plunk will be called the ground node. The ground is specified simply by setting its potential equal to the designator "ground-potential" (the unconstrained plunk) which is treated as a referentially transparent designator. To see how the specification of a ground node interacts with the analysis of the above circuit consider the following alternate definition of the circuit's topology:

```
(defpred series-1-topology (cs r1 r2)
         (node '(t1 ,cs) '(t1 ,r1))
         (node '(t2 ,r1) '(t1 ,r2))
         (== '(>> potential t2 ,r2) 'ground-potential)
         (== '(>> potential t2 ,cs) 'ground-potential))
```

or with the use of a ground predicate:

```
(defpred series-1-topology (cs r1 r2)
         (node '(t1 ,cs) '(t1 ,r1))
         (node '(t2 ,r1) '(t1 ,r2))
         (ground '(t2 ,r2) '(t2 ,cs)))
```

The current constraint for the ground node is redundant, and it seems that it is not very useful in constraint propagation, so it is omitted. Now the problem of determining the voltage across the current source in the series circuit is solved by determining the node potentials. The determined value for this voltage, not surprisingly, turns out to be independent of the ground potential (this independence is gotten via the algebraic simplifier).

In more complex circuits the choice of a reference node (ground) can be important to the reasoning process. The absence of an explicit KCL constraint for the ground node can prevent constraint propagation in some cases, in others the unconstrained plunk placed on the ground potential can fail to induce propagation. Whether or not these failures in the constraint propagation occur is largely dependent on which node is chosen for ground. Thus the choice of a ground node can greatly effect the ease with which circuits are analyzed by the system.

The extra degree of freedom in the node potentials is not present in the branch voltages, which are *differences* between node potentials. All currents in a circuit are functions of the branch voltages of that circuit and do not depend on the absolute potentials of the nodes. Thus circuit analysis is ultimately concerned only with branch voltages and not with the absolute potentials. Therefore all of the quantities which are of interest in electrical analysis do not depend on the unconstrained potential assigned to a reference node. Therefore the choice of the reference node does not effect the form of expressions for quantities of interest. However, as will be seen below there are cases in which the choice the quantity to be plunked has a great effect on resulting expressions for quantities of interest.

Another example of the use of unconstrained plunks is circuits with

external terminals. In such circuits one is interested in the constraints the circuit imposes on the currents and potentials of the external terminals. To put the discussion in a more concrete framework consider the following circuit:

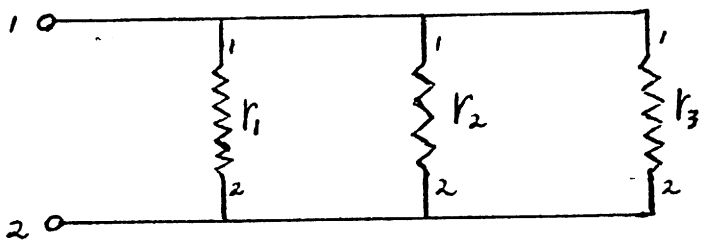



Figure 5. The "3-Parallel" Circuit

```
(defpred 3-parallel (circuit)
  (one-port circuit)
  (resistor '(r1 ,circuit))
  (resistor '(r2 ,circuit))
  (resistor '(r3 ,circuit))
  (term-eq '(t1 ,circuit)
           '(composite (>> t1 r1 ,circuit)
                       (composite (>> t1 r2 ,circuit)
                                  (>> t1 r3 ,circuit))))
  (term-eq '(t2 ,circuit)
           '(composite (>> t2 r1 ,circuit)
                       (composite (>> t2 r2 ,circuit)
                                  (>> t2 r3 ,circuit)))))
```

None of the branch voltages or branch currents can be determined if the environment of the circuit is left unspecified. However it is possible to reason about this circuit by giving the voltage across the terminals of the circuit an unconstrained plunk. The following dialog with the system demonstrates the use of the unconstrained plunk:

```
(assert (3-parallel 'c1))
(3-PARALLEL C1)

(what-is '(current c1))
(CURRENT C1)

(assert (== '(voltage c1) 'input-voltage))
(== (VOLTAGE C1) INPUT-VOLTAGE)

(what-is '(current c1))
(// (* INPUT-VOLTAGE
       (+ (* (RESISTANCE (R1 C1))
             (+ (RESISTANCE (R2 C1))
                (RESISTANCE (R3 C1))))
          (* (RESISTANCE (R2 C1))
             (RESISTANCE (R3 C1)))))
    (* (RESISTANCE (R1 C1)) (RESISTANCE (R2 C1)) (RESISTANCE (R3 C1))))
```

The constraint propagation to produce this result is straightforward. The substitution process ensures that the voltage across the external terminals of the circuit is in the same equivalence class as the voltage across each of the resistors. Thus when the external voltage is plunked the voltage across each resistor is determined and therefore the current through each resistor is determined via Ohm's law. The current through the external terminals is then determined as a function of the voltage across them. Unfortunately the system is not capable of generalizing such knowledge and applying it to situations in which this circuit is embedded in larger circuits. However a human engineer could use the system to analyze general instances of circuit fragments and then incorporate the results into TMS predicates for those fragments.

Other quantities might have been plunked in the above circuit. For example the following dialogue shows how the voltage could be solved for in terms of the current:

```
(assert (3-parallel 'c2))
(3-PARALLEL C2)

(assert (== '(current c2) 'input-current))
(== (CURRENT C2) INPUT-CURRENT)
```

```
(what-is '(voltage c2))
(VOLTAGE C2)

(solve-for '(voltage c2))
(// (* INPUT-CURRENT
       (RESISTANCE (R1 C1))
       (RESISTANCE (R2 C1))
       (RESISTANCE (R3 C1)))
    (+ (* (RESISTANCE (R1 C1))
          (+ (RESISTANCE (R2 C1))
             (RESISTANCE (R3 C1))))
       (* (RESISTANCE (R2 C1))
          (RESISTANCE (R3 C1)))))
```

Normal plunking had to be initiated via `solve-for` to perform the analysis but the desired result was obtained. The circuit can be viewed as providing a constraint on its terminals which allows the current to be derived from the voltage or the voltage from the current. However in each of the above analyses, one of these quantities was chosen as the "input" and the other was derived as an expression in terms of it. In more complex circuits particular terminals can be specified as input and output terminals. One is usually only interested in expressions for the output quantities in terms of the inputs. Thus the multidirectional view of a circuit as a constraint is replaced by a unidirectional relationship between inputs and outputs.

## Slices

This section develops the use of "slices" in circuit analysis. The term slice was coined by Gerald Sussman to refer to a form of multiple description which is useful in constraint propagation [Sussman 77] [Steele & Sussman 79]. The basic technique is to state an equivalence between a part of the structure being analyzed and a different structure which exhibits the same behavior. Such equivalences are quite naturally stated in the equality system. The simplest use of slices in electrical analysis is in series-parallel reduction. Consider two resistors in series connected to a voltage source as is shown in figure 6.

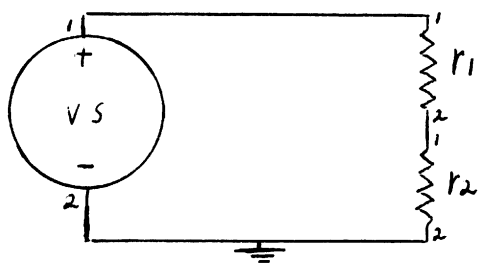



Figure 6. The "Series-2" Circuit.

```
(defpred series-2 (circuit)
    (voltage-source '(vs ,circuit))
    (resistor '(r1 ,circuit))
    (resistor '(r2 ,circuit))
    (node '(>> t1 vs ,circuit) '(>> t1 r1 ,circuit))
    (node '(>> t2 r1 ,circuit) '(>> t1 r2 ,circuit))
    (ground '(>> t2 r2 ,circuit) '(>> t2 vs ,circuit)))
```

Now if it is asserted that some circuit, say c2, is a series-2, and then the system is asked for the current of r1, the system will simply respond with (>> current r1 c2). In other words the circuit will not be analyzed. The potential of the top node can be determined to be the ground potential plus the strength of the source. However the potential of the node connecting r1 and r2 does not take on a determined value since the neither resistor has a determined voltage. There is no way to derive the resistor voltages since the currents are not known.

This circuit can be solved however with the use of slices. An slice is an alternate description of some portion of a structured object. In circuit analysis slices are installed by giving designators for terminal currents and potentials in terms of equivalent circuits. In this case r1 and r2 are equivalent to a single resistor with a resistance equal to the sum of their resistances. Figure 7 shows this circuit with the slice imposed.

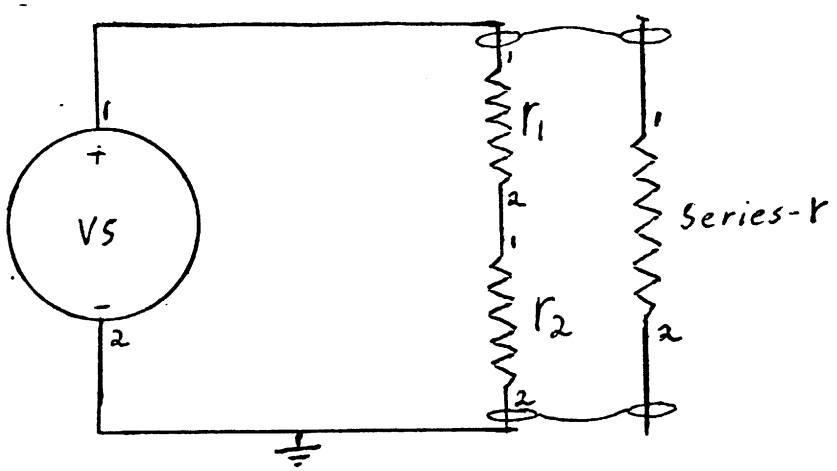



Figure 7. A Slice on the "Series-2" Circuit.

This equivalence is stated by defining a resistor with the appropriate resistance and then stating equalities between its terminal potentials and currents, and terminal potentials and currents in the circuit. This can be done in the equality system as follows:

```
(defpred term-eq (t1 t2)
         (== '(potential ,t1) '(potential ,t2))
         (== '(current ,t1) '(current ,t2)))

(assert (series-2 'c1))

(assert (c== '(resistance series-r)
             '(+ (>> resistance r1 c1) (>> resistance r2 c1))))

(assert (term-eq '(>> t1 r1 c1) '(t1 series-r)))

(assert (term-eq '(>> t2 r2 c1) '(t2 series-r)))
```

With such constraints installed the current in the equivalent resistor is determined from the strength of the voltage source and the resistance of that resistor. This current then determines the current in r1 and r2, which determines the voltages across them and therefore the potential of the central node.

The following predicate and operator definitions give a convenient means of creating series and parallel slices.

```
(defpred one-port-eq (t1 t2 one-port)
    (term-eq t1 '(t1 ,one-port))
    (term-eq t2 '(t2 ,one-port)))
```

```
(evalfun series-eq (r1 r2)
    (-> (predand (resistor r1) (resistor r2))
        (predand (resistor '(series-resistor ,r1 ,r2))
                 (c== '(resistance (series-resistor ,r1,r2))
                      '(+ (resistance ,r1) (resistance ,r2))))))

(evalfun parallel-eq (r1 r2)
    (-> (predand (resistor r1) (resistor r2))
        (predand (resistor '(parallel-resistor ,r1 ,r2))
                 (c== '(// 1 (resistance (parallel-resistor ,r1,r2)))
                      '(+ (// 1 (resistance ,r1))
                          (// 1 (resistance ,r2)))))))
```

Of course more complex circuits can be defined, and the machinery has now been developed to do series-parallel reduction. Consider the ladder network of resistors shown in figure 8.

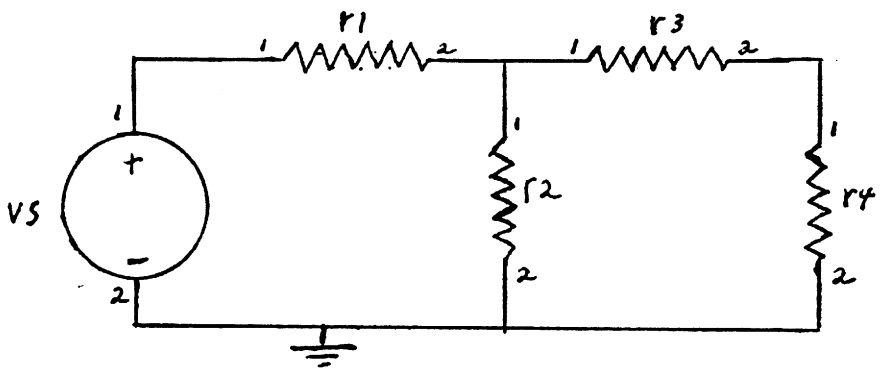



Figure 8. The "Ladder" Circuit.

```
(defpred ladder (c)
         (voltage-source '(vs ,c))
         (resistor '(r1 ,c))
         (resistor '(r2 ,c))
         (resistor '(r3 ,c))
         (resistor '(r4 ,c))
         (ladder-topology '(vs ,c) '(r1 ,c) '(r2 ,c) '(r3 ,c) '(r4 ,c)))

(defpred ladder-topology (vs r1 r2 r3 r4)
         (node '(t1 ,vs) '(t1 ,r1))
         (node '(t2 ,r1) '(t1 ,r2) '(t1 ,r3))
         (node '(t2 ,r3) '(t1 ,r4))
         (ground '(t2 ,r4) '(t2 ,r2) '(t2 ,vs))
         (ladder-topology2 r1 r2 r3 r4))

(defpred ladder-topology2 (r1 r2 r3 r4)
    (one-port-eq '(t1 ,r3) '(t2 ,r4) '(series-eq ,r3 ,r4))
    (one-port-eq '(composite (t1 ,r2) (t1 (series-eq ,r3 ,r4)))
                 '(composite (t2 ,r2) (t2 (series-eq ,r3 ,r4)))
                 '(parallel-eq ,r2 (series-eq ,r3 ,r4)))
    (one-port-eq '(t1 ,r1)
                 '(t2 (parallel-eq ,r2 (series-eq ,r3 ,r4)))
                 '(series-eq ,r1 (parallel-eq ,r2 (series-eq ,r3 ,r4)))))
```

Here is a dialog with the system about this circuit:

```
(assert (ladder 'c2))
T
(what-is '(>> current r4 c2))
(// (* (>> STRENGTH VS C2)
       (RESISTANCE (PARALLEL-EQ (R2 C2) (SERIES-EQ (R3 C2) (R4 C2)))))
    (* (RESISTANCE (SERIES-EQ (R1 C2)
                              (PARALLEL-EQ (R2 C2)
                                           (SERIES-EQ (R3 C2) (R4 C2)))))
       (RESISTANCE (SERIES-EQ (R3 C2) (R4 C2)))))
```

This value was derived by straightforward propagation. The potential difference across the equivalent resistor for the entire circuit is determined by the voltage source. The current through that resistor is then determined by Ohm's

law. This determines the current through r1 and through the equivalent resistance for the remainder of the ladder. The current through this equivalent resistor determines the potential across it via Ohm's law, which in turn determine the potential across the pair of resistors r3 and r4. The current through this pair of resistors is determined via the use of their equivalent resistance and Ohm's law. This finally determines the current in r4.

The value given for the current in r4 is in terms of the equivalent resistances used in the slices. It is not immediately clear whether the better-name relation should consider this a better or worse representation than an algebraic expression in terms of the resistances of the actual resistors in the circuit. It can be argued that the use of the equivalent resistances prevents the algebraic simplification of expressions. However in the above case the expression containing the equivalent resistances is actually shorter (by 6 symbols) than the result of simplifying an algebraic expression in the original resistances, which is shown below:

```
(// (* (+ (>> RESISTANCE R3 C2) (>> RESISTANCE R4 C2))
       (>> STRENGTH VS C2))
    (+ (* (>> RESISTANCE R1 C2)
          (+ (>> RESISTANCE R2 C2)
             (+ (>> RESISTANCE R3 C2)
                (>> RESISTANCE R4 C2))))
       (* (>> RESISTANCE R2 C2)
          (+ (>> RESISTANCE R3 C2)
             (>> RESISTANCE R4 C2)))))
```

Not enough experience has yet been had with the electronics system to tell whether the use of the equivalences in the results of analysis is always a good thing. The present system uses them.

## "Redrawing" Circuits

The equivalences used to solve the above ladder circuit do not always result in a complete circuit analysis. There are cases in which series parallel reduction should clearly be possible, but that the slices given so far are not adequate. Consider the circuit shown in figure 9.

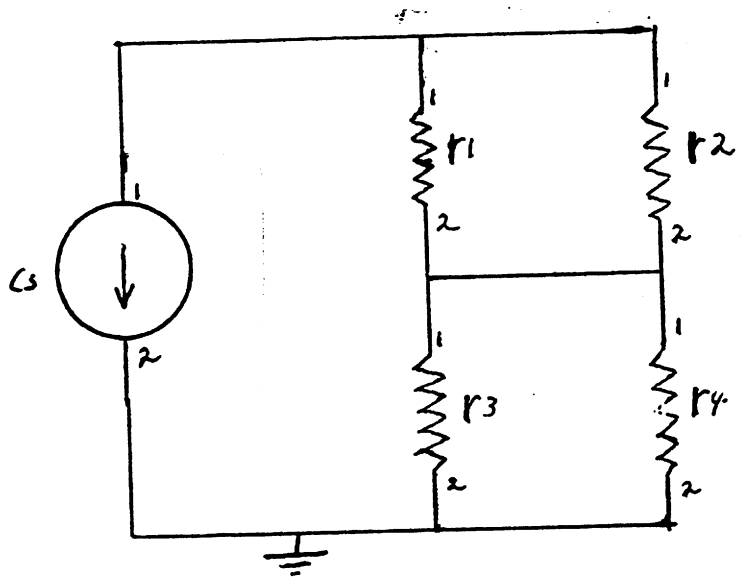



Figure 9. Two Sets of Parallel Resistors in Series.

A parallel series reduction of this circuit can be attempted by placing parallel slices on the two pairs of parallel resistors in the circuit. Since the strength of the current source is a transparent designator, propagation begins from that point. The sum of the currents into the upper terminals of r1 and r2 is determined via the current constraint at the top node. This determines the current in the parallel equivalent of those resistors, which determines the sum of the currents in the lower terminals of r1 and r2. Now it would seem that the sum of the current in the upper terminals of r3 and r4 should be determined. However if the node predicate has been used in the standard way to create the current constraint for the center node, then the constraint net does not constrain the sum of the currents in r3 and r4 in a manner which allows this determination to be used. The current constraint for the node connecting r1, r2, r3 and r4 might look like:

```
(== '(>> current t2 r1)
    '(- (+ (>> current t2 r2) (>> current t1 r3) (>> current t1 r4))))
(== '(>> current t2 r2)
    '(- (+ (>> current t2 r1) (>> current t1 r3) (>> current t1 r4))))
(== '(>> current t1 r3)
    '(- (+ (>> current t2 r1) (>> current t2 r2) (>> current t1 r4))))
(== '(>> current t1 r4)
    '(- (+ (>> current t2 r1) (>> current t2 r2) (>> current t1 r3))))
```

This constraint cannot give a better name to the sum of the currents in the upper terminals of r3 and r4 simply because none of the equalities representing the constraint deal with that sum. This prevents the current in the equivalent resistance of r3 and r4 from being determined. There is no other way for constraints to propagate around the circuit. Even though the voltage across r1 and r2 can be determined, and therefore the two branch currents, the potential difference across the current source has not been determined, so that the potential across the bottom resistors remains unknown. Also, since no current constraint is created at the ground node, the sum of the currents in r3 and r4 are not directly constrained. (If such a current constraint were present an example of the failure of parallel slices would only be slightly more complex.) A solution to this dilemma is suggested by the alternate drawing of the circuit shown in figure 10.

The parallel slices are somehow more strongly suggested in this drawing. This is due to the explicit terminals which can then be made equivalent to the terminals of the resistors in the slices. As a result of this, the current constraints implicit in the diagram more directly constrain the current in the equivalent resistance. It is possible to "redraw" a circuit by restructuring the current constraints to take into account terminals of slices. This can be done directly using the predicates and operators defined so far. For example a parallel-series reduction of the above circuit can be accomplished if the following is used to construct the central node:

```
(connected '(composite (t2 r1) '(t2 r2)) '(composite (t1 r3) (t1 r4)))
```

The present method of creating slices relies heavily on the user of the system. It would be far more satisfactory to have the system recognize the appropriate equivalences on its own. This has not been accomplished to date and might represent a fruitful area for further research.

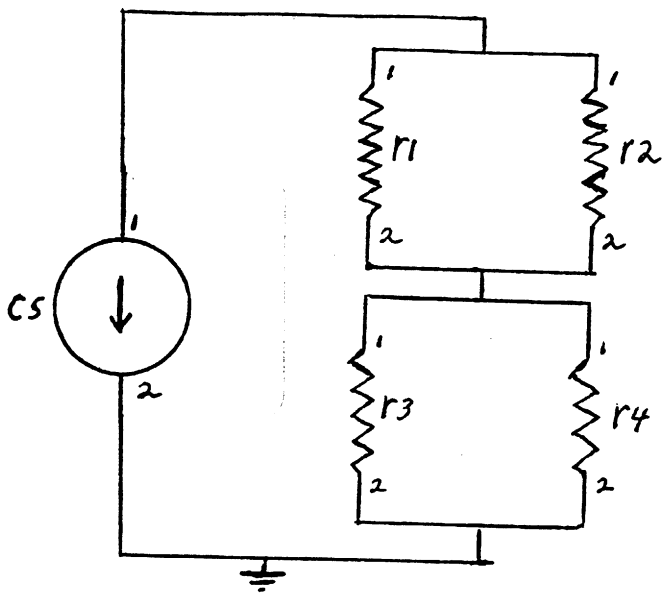



Figure 10. An Alternative Drawing.

The major point brought out in the discussion of electronics presented in this chapter is that an effective reasoning system can be derived from algorithms designed to handle the problems surrounding equality and multiple descriptions. The majority of the reasoning strategies used in these algorithms can be justified in terms of general principles without reference to electronics.

# Chapter IV
## Algorithmic Details and Relation to Other Work

### The Truth Maintenance System

A truth maintenance system or TMS is used in the equality system to record and enforce logical relations among propositions. The basic functions performed by the TMS were first introduced by Richard Stallman and Gerald Sussman in an electrical analysis system which made assumptions about transistor states [Stallman & Sussman 77]. A separate module, called a truth maintenance system, or TMS, was later designed by Jon Doyle [Doyle 78]. Doyle's TMS keeps track of propositional justifications and uses them to incrementally update beliefs and track down assumptions underlying contradictions. The propositional reasoner described here is a further refinement of these ideas and is very similar to a truth maintenance system developed by the author and described in a separate publication [McAllester 78]. The basic principles of that TMS are briefly stated here and some comments about its relation to other are given.

Each assertion or belief in the system is given a TMS node which can take on one of three truth values, true, false, or unknown. Logical relations among beliefs are recorded in disjunctive clauses which should be viewed as simple constraints on the truth values associated with nodes. For example the representation of an implication between a tms node n1, and a second tms node n2 would be (or (n1 . false) (n2 . true)). If the assertions represented by n1 and n2 were mutually contradictory (could not both be true at the same time), this would be represented in the TMS by the clause (or (n1 . false) (n2 . false)). In general a clause is a list of terms, each of which is an association of a TMS node with either "true" or "false". A clause states a constraint on the truth values of the TMS nodes which says that one of the nodes in the clause must have the associated truth value.

A clause can locally determine the truth value of a node. Whenever all the nodes in a clause except one have the opposite truth value from the one associated with it in the clause, the clause has only one chance left to be satisfied. When this occurs, and the truth value of the node which might still satisfy the clause is "unknown", the truth value of that node is set to the value associated with it in the clause. A pointer is constructed from the node whose truth value was deduced to the clause which deduced it. This pointer gives the well founded support for the truth value of the node and is used in generating explanations and during dependency directed backtracking.

The user of the TMS can add TMS clauses at will, but once in the

system they cannot be removed. This poses absolutely no problem since removable clauses can always be simulated by creating a TMS node to represent the truth of the clause. This is done by the -> predicate which creates a TMS node to represent an implication and adds the clause stating that the implication along with the implication's antecedents implies the conclusion. The TMS node representing the implication is then returned to the user who can set its truth value as he pleases, effectively adding and removing the implication represented by that node.

When the user sets the truth of a node to "true" or "false", as can be done with assert or set-truth, this truth value is taken by the system to be a premise. As premises are added the TMS enforces the constraints represented by its clauses and deduces other truth values. The user can also retract premises via a remove-truth function which makes a node unknown. When this is done the support for other truth values can become invalid. Whenever the supporting clause for the truth value of a node can no longer be used to deduce the value of that node the supported truth value is removed. Thus the removal of a truth value can propagate to generate the removal of a large number of other truth values. After this propagation has occurred each of the nodes whose truth value was removed must be checked for alternate supports for truth values. By doing this only after the removal phase is completed the system avoids the possibility of looping support structures.

Because constraints can be added at any time, and because loops can exist in the constraint set, it is possible for contradictions to arise. A contradiction is nothing more than a clause in which all the nodes have the opposite truth value from the one associated with it in the clause. When this happens the support pointers of the nodes in the clause can be used to track down the set of premises which underlie the truth values of those nodes. One of these premises can then be retracted to remove the contradiction.

Whenever a truth value for a node is an underlying support for the truth value of a node in a contradiction, i.e. whenever a truth value leads to a contradiction, it is valid to deduce the opposite truth value for that node. There may be loops in the constraints in the TMS which prevent this from happening naturally. For this reason a backtracking facility is provided which will take a contradiction and an underlying premise and use clause resolution to generate new clauses which bypass the loops preventing the deduction of the negation of the premise. Since small clauses are useful in more situations, the backtracker does not compute a "conditional proof" constraint involving all the premises underlying a contradiction as is done in other systems [Doyle 78]. It instead does a minimum amount of clause resolution which adds clauses just large enough to

break the constraint loops.

Demons can be attached to the TMS nodes which are invoked when the node changes truth value. There are three types of demons which can be attached to a node, true-noticers, false-noticers, and unknown-noticers, which are activated when the node takes on the suggested value. True-noticer demons are attached to equality nodes in the equality system which activate certain canonical naming functions when the equality becomes true. Nodes can be given unknown-noticer demons which set the node to a default truth value (as a premise) when the node would otherwise be unknown. These noticers are run only at times when the TMS is otherwise stable, making interactions between TMS functions and noticer actions comprehensible.

Assumptions are simply specially marked premises. This marking serves no other purpose than to act as a guide when choosing a premise to retract during backtracking. *There is no significant distinction in this system between an assumption and a simple premise.* Non-monotonic dependency structures have been used in other systems to make assumptions by justifying a belief in terms of a lack of knowledge to the contrary [Doyle 78]. Such dependency structures were found to be totally unnecessary in the current system and in the author's opinion they lead to considerable algorithmic complexity and conceptual obscurity. With a slight modification to the TMS as described here it can also be shown that there is no expressive power gained from non-monotonic dependencies.

## The Canonical Naming Algorithm

The basic mechanism which must be implemented is the determination of a canonical name for an equivalence class under the equalities provided to the system. The equalities can be though of as arcs between nodes representing the designators in a graph. Finding a canonical name involves finding a designator which is at least as good as any other designator in the class under the better-name relation. A marking algorithm is used to scan the designators in a class. The marker placed on a class of designators is maintained on each designator in that class and a pointer is maintained from the marker to the canonical name of that class. This makes access to the canonical name of any designator's equivalence class very efficient.

A major design goal is the ability to add and remove equalities at will, and maintain proper canonical names. This algorithm must therefore be extended to handle the maintenance of the canical names when the set underlying equalities changes. The freedom to add and remove equalities means that

equivalence classes can merge and split as the equalities change. I will first consider the case of adding equalities. When a new equality is added (its TMS node takes the value "true"), it can merge two previously distinct equivalence classes. Thus each new equality added must be checked to see if a marker can be propagated across it. If the two designators which become linked have distinct markers on them, then one of the markers, call it m1, is chosen to propagate across the equivalence class. The other marker, call it m2, can be removed from all designators on which it appears since m1 must propagate across the entire class previously marked by m2. As a marker propagates across new members of its equivalence class the canonical name pointer must be appropriately kept up to date.

Now consider the problems involved in retracting equalities. Each marker starts from a single designator and propagates to other names via equalities. The class marked by the marker is defined to be the class containing the marker of origin. This class can change as equalities are added and removed. In light of this definition, the presence of a marker on a designator can be considered valid only if the designator is in the same class as the designator from which the marker originated. Thus the marking of a designator with a marker is associated with a TMS node representing the equality between the marked designator and the origin of the marker.

As a marker is propagated across an equality from designator1 to designator2 a logical relation is added to the TMS which states that the equality between designator1 and the marker origin, along with the equality between designator1 and designator2, imply the equality between designator2 and the marker origin. The truth of the equality between designator2 and the origin is always checked when the marker is used to find a canonical name for designator2. If any equality is later removed, then the equalities which depend upon it are removed (their TMS nodes take on a truth value of "unknown") and marking associated with these equalities can be recognized as invalid.

Because equalities can be retracted at any time, it is possible that the canonical name pointed to by a marker could be removed from the class marked by the marker. Since the primary purpose of the markers on designators is to point to the canonical name for that designator, this would render the marker useless. For this reason a marker is only considered valid while the canonical name it points to is in its equivalence class. A canonical name function has been written which takes any designator, first ensures that that designator is validly marked by a valid marker and then returns the canonical name pointed to by that marker. In this way the canonical names are correctly maintained under the retraction of equalities.

## Substitution

The means of controlling the substitution process have not been discussed so far. An image function is defined on designators such that the image of a designator is the result of replacing all the top level subdesignators by their canonical names. The image of a name is equal to the name as long the equalities between the parts hold. The invariant which the equality system attempts to maintain is that the images of all designators are explicitly represented in that designators class. This allows the image of the designator to be a candidate for the canonical name of that class. Since canonical names are continuously changing, the image of a designator changes also. Thus a designator must be monitored in some way to ensure that its image is always in its class.

Before discussing the details of the way in which this process is controlled, first consider some general properties of the image function. An internally-equal relation can be defined on designators which is stricter than the standard equality relation used throughout this document. Two designators are internally-equal if they have the same number of top level subdesignators and each pair of corresponding top level subdesignators are equal. All internally-equal designators have the same image. For example (+ 1 2) is internally equal to (+ (- 2 1) (+ 1 1)) but not internally-equal to 3 or (+ 2 1), assuming the appropriate numerical equalities have been added. Both (+ 1 2) and (+ (- 2 1) (+ 1 1)) have the same image, which should be (+ 1 2). An equivalence class can be divided into a set of internally-equal subclasses. Only one element from each of these subclasses need be monitored with respect to substitution since all the elements in a given class have the same image.

All designators have their image computed at least once. A TMS node is constructed to represent the assertion that the image is internally-canonical, i.e. all its top level subdesignators are the canonical names of their equivalence classes. Each canonical name has a TMS node associated with it representing the assertion that it is the canonical name for its equivalence class, and the truth of this node is maintained by the equality system. These nodes can be used to construct a support for the truth of the node representing the assertion that an image is internally canonical. A demon is placed on this internally canonical node which recomputes the image of that designator when the node becomes unknown, thus monitoring that designator in a way that ensures its image will be recomputed when needed. This single monitoring is all that is needed to monitor the entire subclass of designators which are internally-equal to this designator. Thus when an image is computed only the image is monitored for future need to

recompute an image.

Because equalities can be retracted, it is possible that internally-equal subclasses get split up. When this happens the single designator which was being monitored is no longer sufficient to monitor all the designators which were in that subclass. To make sure that all designators are properly monitored, an internally-equal node is associated with every designator which is being monitored via its internal equality to another designator. This internally-equal node represents the assertion that the designator is internal equal to some other designator which is monitored. When an image is computed from a designator, and is not equal to that designator, a node representing the internal-equality between the two designators is created and associated with the designator whose image was computed. If this internally-equal node ever becomes unknown, then the image of that designator is recomputed. Thus it is possible to ensure that the image of every designator is always in that designators equivalence class and has a chance at becoming the canonical name of that class.

The present equality system runs unreasonibly slow, on the order of a minute per constraint in constraint propogation analysis of electronic circuits. The primary bottleneck in the process turns out to be inefficient hashing of designator expressions. Improving the hashing methods should yeild about an order of magnitude improvement in computation time, but the resulting six seconds per constraint is still too slow to yeild a practicle system for serious problems. This is a first pass implementation and it is hoped that future refinements can still further significantly reduce the computation times involved.

## Relation to Other Mechanisms for Handling Equality

This section discusses several classes of work related to the equality system described in this document. The first relates the work discussed here to previous work which used canonical expressions to represent equivalence classes. The second discusses an approach taken to equality in the resolution theorem proving tradition. A third approach to equality incorporates it into pattern matching (or unification) procedures. Finally, special attention is paid to an algorithm developed by Knuth and Bendix to decide equivalences in certain universal algebras.

Canonical forms have been used to represent classes of equivalent expressions in many previous systems. Ira Goldstein used canonical forms for geometric objects in a geometry theorem prover [Goldstein 73]. More recently Howard Shrobe used canonical representatives of equivalence classes in a more general purpose reasoning framework [Shrobe 79]. In Shrobe's system substitution

was performed in such a way that all reasoning took place in terms of the canonical expressions. There are certainly many other systems which employ similar methods that the author is unaware of, and forgiveness must be asked from those who are not referenced. The originality claimed here is not the concept of using canonical representatives of equivalence classes but a particular application of this mechanism to reasoning. In the systems mentioned above the existence of equivalent expressions is dealt with as a problem to be overcome in the reasoning process. Canonical expressions are used to unify the knowledge about an individual expressed in terms of different designators for that expression. If equality is only viewed from this perspective then one tends to minimize the role of multiple designators for the same referent in the reasoning process. In the applications of equality discussed here, aspects of the canonical naming mechanism which are ignored by these other systems, such as substitution into non-canonical expressions, and the detailed structure of a better-name relation, take on a great significance in the reasoning process. Here equality is viewed as playing a central role in reasoning, rather than as a necessary evil.

A method for handling equality, called paramodulation, has been developed in the context of resolution theorem proving [Robinson 65] [Chang & Lee 73]. This technique is a clause resolution rule which incorporates substitution of equals for equals. This allows deduction of anything deducible in a system of first order predicate calculus with equality. The paramodulation rule itself however provides no guidance as to when substitutions should be done and systems using little or no heuristic guidance become lost in combinatorial explosions. Several methods have been developed which cut down on the number of allowed applications of paramodulation, such has hyperparamodulation and linear paramodulation. However such restrictions on paramodulation do not give the detailed guidance of substitution provided by canonical naming schemes.

Another approach taken to equality involves incorporating equality axioms into pattern matching (or unification) [Fay 78]. A discussion of this is best done by an example. Suppose that op is an associative operator (i.e. (op (op x y) z) = (op x (op y z))). The match of (op (op a b) c) with (op x y), where x and y are variables and a, b, and c are constants, would yield two substitutions: {(x . (op a b)) , (y . c)} and {(x . a) , (y . (op b c))}. Thus the equality variants of the two expressions being unified are taken into account in generating the variable bindings. As in the case of paramodulation, such a scheme can be incorporated into a system that can make all valid deductions from a given set of axioms in first order predicate calculus. However, for this approach to be feasible, the equality axioms must be determined at the time the matching is done. This requirement makes this approach inappropriate

to the type of reasoning about equality presented in this document where the set of useful equalities is continually expanding.

The most notable of the algorithms for handling equality is one developed by Donald Knuth and Peter Bendix [Knuth & Bendix 69]. They give an algorithm for deciding whether two words (designators) are equivalent under equivalence relations defined by simple sets of algebraic axioms. A partial order on words, analogous to the better-name relation used here, is used to reduce a given word to a canonical form. Given two words it is possible to tell if they are equivalent by seeing if their canonical forms are the same. Knuth and Bendix represent the algebraic axioms as a set of reductions which take the form of rewrite rules and represent equalities. They also give a decision procedure for determining if a given set of reductions is complete, i.e. results in a complete decision algorithm for tests of equality under those axioms. Perhaps the most interesting aspect of their work is a method given for extending a reduction set to a complete set by adding derived reductions.

When comparing the algorithms developed by Knuth and Bendix to the equality system described here, a strong similarity emerges. Each equality in the equality system acts as a reduction in the sense that it can result in one designator being replaced by another inside a larger designator when substitution occurs. Reductions which contain variables can not be encoded as equalities in the equality system. However arbitrary reductions can be placed in the evaluation functions. Since the general equality system allowed for an arbitrary better name relation, the relations used by Knuth and Bendix could be easily incorporated. Therefore any partial order on designators and any complete reduction set used by a Knuth and Bendix type algorithm can be used in the equality system. Furthermore since each derived equality acts as a new reduction, any reduction set which can be extended to a complete set by the Knuth and Bendix algorithm will, in much the same way, automatically form a complete set in the equality system.

I do not mean to claim a rediscovery of the results on completeness found by Knuth and Bendix which involve the developement of a specific better-name relation and its interaction with specific sets of equality axioms. I am not particularly concerned with completeness however, and approached the problems presented by equality from a completely different perspective. They were concerned with deciding equality under a small set of algebraic axioms which remain fixed over time. I am concerned with the use of equality as a means of knowledge representation and as a mechanism for handling multiple descriptions. Thus I am concerned with a very large rapidly expanding set of equalities (or reductions) which can be manipulated as the reasoning process progresses.

An important fundamental distinction between the system described here and all previous work on equality (at least to the author's knowledge) is that this system views equality and multiple descriptions as a central part of the reasoning system. Thus it makes sense to ask the system for the value of an expression even though the expression itself is a designator for that value. In traditional theorem proving systems such a request could not even be properly stated, an expression for the desired quantity is trivially found and the system has no way of telling that this does not completely solve the problem at hand.

# Appendix One
## A Basic TMS and Equality System User's Manual

The basic utility functions of the TMS and the equality system are divided into three sections. First are the TMS utilities which are used in the code for the equality system. These primitives allow the direct creation of TMS clauses and the manipulation of the truth values associated with TMS nodes. However they are not intended to be used at the interactive user level. The next set of functions are TMS utilities which are intended to be used interactively by a system user, including the basic TMS predicates. The final section presents the basic equality system utilities.

### Internal TMS Utilities

#### DEPENDENCY-NODE

```
(dependency-node <assertion>)
```

This function takes an "assertion" in the form of an s-expression and returns a TMS node which has been associated with that assertion. The assertions are kept in a hash table which is searched each time dependency-node is called. Thus repeated calls with the same assertion yield the same node.

#### SET-TRUTH

```
(set-truth <node> <value> <setter>)
```

This function takes a node, a value which is either the atom true or the atom false, and an atom used to record the source of the set value. The node takes on the truth value and the TMS automatically propagates the results of that value. Noticer functions, which are meant to notice changes in truth values, are called automatically when the propagation has stabilized. A noticer function can induce further changes in truth values and the process continues until no noticers are left to run.

The most common value for the setter argument is the atom "premise". However, if the setter argument is the atom "default", then the

set truth value is marked as an assumption and is treated specially during backtracking.

If the node already has a truth value, the node is still given the new value with the new justification. This can lead to contradictions if the opposite value had been previously deduced. Such contradictions are treated the same as any other contradictions in the system.

### REMOVE-TRUTH

```
(remove-truth <node>)
```

This function is used to retract premises. The truth value of <node> will be retracted and the appropriate propagation of truth value removal and invocation of noticer functions is done.

### T-NOT

```
(t-not <term>)
```

A term is either a node or an association of a node with either the atom true or the atom false. It is often convenient to be able to bundle an node with a truth value in using primitive TMS functions. The t-not primitive returns an association of a node with the opposite value from the one associated with it in the argument to t-not. A single node is always treated as if it were associated with "true".

### IMPLIES

```
(implies <terms> <term>)
```

This function takes a list of antecedent terms and a consequent term and installs the constraint on node truth values corresponding to this implication. Once the constraint has been added it cannot be removed. For this reason the primitive implies should only be used to state tautologies, or relations which are true by definition.

## CONTRADICTORY

```
(contradictory <terms>)
```

This primitive is essentially the same as implies. It takes a list of terms and installs the constraint that at least one of the nodes in those terms must have the opposite value from the one associated with it in the term. Again this should be used only to state tautologies.

## TRUE-NOTICER, FALSE-NOTICER, UNKNOWN-NOTICER

```
(true-noticer <node> <form>)
```

These functions attach noticers to nodes. The noticer is a form to be evaluated when the node takes on the appropriate truth value.

## DEFAULT

```
(default <node> <value>)
```

If node has an unknown truth value then this sets the value to the one given with a setter of "default", which marks the value as an assumption to be treated specially by the backtracker. It also places an unknown noticer on the node which will set the node to the default value whenever it would otherwise be unknown.

## IMPLIES-UNI

```
(implies-uni <terms> <term>)
```

This is just like implies except that the constraint generated will not automatically deduce the negation of any antecedent term. If the constraint is violated however, it is treated as a contradiction and can then be used to deduce the negation of any antecedent term. This was implemented primarily to capture the full expressive power found in other systems which use non-monotonic dependency structures. The actual utility of this feature is an open question.

## User Level TMS Utilities

These utilities are intended to be used at the interactive user level. To achieve sufficient generality they take "propositions" as arguments. A "proposition" is either a term (a TMS node or a TMS node associated with a truth value) or an s-expression representing an assertion which is coerced to a node via dependency-node as described in the previous section.

### ASSERT

```
(assert <prop>)
```

This function takes a proposition and calls set-truth on the corresponding node. The atom premise is always used as the setter. Thus (assert <node>) is identical to (set-truth <node> 'true 'premise). And (assert 'p) is identical to (set-truth (dependency-node 'p) 'true 'premise).

### RETRACT

```
(retract <prop>)
```

If the node associated with <prop> has a truth value which is given as a premise then this function removes that truth value values (and all values which critically depended on that value). If the proposition has a deduced truth value or a default truth value then this function is effectively a no-op.

### ASSUME

```
(assume <prop>)
```

This sets the truth value of the node associated with the proposition to a truth value and marks that node as an assumption which can be recognized by the backtracker. It does this via a call to default as described above.

## UNLESS

```
(unless <prop1> <prop2>)
```

This gives prop2 a true default value and installs a unidirectional implication such that if prop1 becomes true prop2 will be deduced to be false. However since this implication is unidirectional the assumption that prop2 is true will not result in a deduction that prop1 is false. This primitive depends on the implies-uni facility described above and is of questionable utility.

## PREDNOT

```
(prednot <prop>)
```

If the proposition is a TMS node or an assertion s-expression this returns a dotted pair of the TMS node and "false". Thus (prednot <node>) is equivalent to (cons <node> 'false). And (assert (prednot <node>)) is equivalent to (set-truth <node> 'false 'premise). If the argument to prednot is already an association of a node and a value then a pair of the node and the opposite value is returned.

## WHY

```
(why <query>)
```

A query is either a proposition or a numerical reference to nodes mentioned in previous explanations generated via why. Applications of this function can be divided into three cases. First, if the node referenced in the query has an unknown truth value then a simple statement to that effect is printed. Second, if the node has a truth value which was given as a premise then the truth value is stated and the user is told that it is a premise and gives him the value of the setter argument used when the premise was made. Finally, if the node has a deduced truth value then the associated truth value is stated along with a numbered list of supporting terms. The number associated with each term can be used to reference that term in the next use of the why function.

The number 0 can be used to "pop" back to the previous justification and allow numerical references to terms in that justification. Therefore it is very easy to walk across the support tree for any assertion via numerical references to the supporting terms.

->

```
(-> <prop1> <prop2>)
```

If the propositions are simple nodes or assertions the above expression returns a TMS node which has the following TMS constraint imposed:

```
(or ((-> <node1> <node2>) . false)
    (<node1> . false)
    (<node2> . true))
```

If the propositions are simple nodes or assertions an appropriate constraint is installed for the associated truth values.

## PREDOR

```
(predor <prop1> <prop2> <prop3> ...)
```

If the arguments are simple nodes or assertions the above form returns a node with the below TMS constraints imposed. Otherwise analogous constraints are imposed for the associated truth values.

```
(or ((predor <node1> <node2> ...) . false)
    (<node1> . true)
    (<node2> . true)
    ...)

(or (<node1> . false) ((predor <node1> <node2> ...) . true))

(or (<node2> . false) ((predor <node1> <node2> ...) . true))

    ...
```

## PREDAND

```
(predand <prop1> <prop2> <prop3> ...)
```

This is analogous to predor.

## PRESUMABLY

(presumably <prop>)

This predicate performs the following side effect involving the node returned:

```
(assume (-> '(presumably <prop>) <prop>))
```

## DEFPRED

The defpred construct is a means of defining TMS predicates. A defpred can have the same syntax as a LISP defun. The node returned by a predicate defined with defpred represents the assertion that the predicate is true of the objects to which it was applied. The body of a defpred is a list of forms which evaluate to either a TMS node or nil. During an application of the defined predicate, if a form in the body evaluates to a TMS node then that node will be implied by the node returned as the value of the predicate application. If the form returns nil it has no effect on the node returned from the predicate application. Some examples are

given below:

```
(defpred vertebrate (x))
(defpred flys (x))

(defpred bird (x)
         (vertebrate x)
         (flys x))

(defpred hairy (x))
(defpred female (x))
(defpred bears-live-young (x))

(defpred mammal (x)
         (vertebrate x)
         (presumably (hairy x))
         (presumably (-> (female x)
                         (bears-live-young x))))

(defpred duck-bill (x)
  (mammal x)
  (prednot (bears-live-young x)))
```

The following is a dialogue with the system using the above predicates.

```
(assert (mammal 'joe))
(MAMMAL JOE)

(assert (female 'joe))
(FEMALE JOE)

(why (bears-live-young 'joe))
((BEARS-LIVE-YOUNG JOE) IS TRUE FROM)
(1 (FEMALE JOE) IS TRUE)
(2 (-> (FEMALE JOE) (BEARS-LIVE-YOUNG JOE)) IS TRUE)
T
```

```
(why 2)
((-> (FEMALE JOE) (BEARS-LIVE-YOUNG JOE)) IS TRUE FROM)
(1 (PRESUMABLY (-> (FEMALE JOE) (BEARS-LIVE-YOUNG JOE))) IS TRUE)
(2 (-> (PRESUMABLY (-> (FEMALE JOE) (BEARS-LIVE-YOUNG JOE)))
       (-> (FEMALE JOE) (BEARS-LIVE-YOUNG JOE)))
   IS TRUE)
T

(why 2)
((PRESUMABLY (-> (FEMALE JOE) (BEARS-LIVE-YOUNG JOE))) IS TRUE FROM)
(1 (MAMMAL JOE) IS TRUE)
(2 (-> (MAMMAL JOE)
       (PRESUMABLY (-> (FEMALE JOE) (BEARS-LIVE-YOUNG JOE)))))
T

(why 1)
((MAMMAL JOE) IS TRUE AS A PREMISE SET BY PREMISE)

(assert (duck-bill 'joe))
(DUCK-BILL JOE)

(why (bears-live-young 'joe))
((BEARS-LIVE-YOUNG JOE) IS FALSE FROM)
(1 (DUCK-BILL JOE) IS TRUE)
(2 (-> (DUCK-BILL JOE) (NOT (BEARS-LIVE-YOUNG JOE))) IS TRUE)
T
```

The repeated application of a predicate to the same arguments returns the same node, but the body of the defpred is only evaluated on the first call. Thus predicate applications can be used in such places as the antecedents of implications, or simply in getting hold of a TMS node to use in a query, without of generating multiple copies of TMS constraints.

There is an alternate syntax for defpred in which the bound variable list is replaced by an atom which is bound to a list of the values of the arguments in a application of the defined predicate. The following use of this syntax is from electrical circuit analysis:

```
(defpred node terminals
        (c== 0 '(+ ,(mapcar '(lambda (term) '(current ,term)) terms)))
        (let ((node-potential '(potential ,(car terms))))
          (predand (mapcar '(lambda (term2)
                              (== node-potential '(potential ,term)))
                           (cdr terms)))))
```

## Naming System Utilities

==

(== <designator1> <designator2>)

This is a predicate which returns a TMS node representing the equality of two designators. The node returned has a true-noticer attached to it that wakes up certain naming routines when the equality node becomes true.

### WHAT-IS

(what-is <designator>)

This returns the canonical name of <designator>'s equivalence class.

### SOLVE-FOR

(solve-for <designator>)

This function "solves for" a particular quantity. This is done by plunking quantities it can be expressed in terms of. The value found for the designator in question may contain plunks. If this happens the solve-for function can be applied to these plunks which initiates further plunking. If the constraints in the system do not determine the designator whose value is sought then its canonical name will be left in terms of plunks.

### EVALFUN

Evalfun is used to define evaluation functions for operators which are used in designators. This evaluation function can be applied to any designator which has the appropriate top level operator. Evalfun has the exact same syntax as defpred. The only difference between an application of a predicate defined via defpred and an application of an evaluation

function defined by evalfun, is that terms returned by forms in the body of an evaluation function are made true as premises instead of being implied by some other node. The following is an example of the use of evalfun for doing the addition of numbers with an evaluation function for a two place addition operator:

```
(evalfun + args
         (if (apply and (mapcar 'numberp args))
             (== '(+ ,@args) (apply plus args))))
```

# Appendix Two
## The Equality System Code

This appendix contains the LISP code for the basic equality system. There are functions and macros used, but not defined here, which are not standard lisp primitives. Some of these are adopted from the LISP MACHINE project at the MIT AI lab. Others are parts of the authors personal utility package. It is hoped that the reader can infer the actions taken by these functions and macros from their context and pnuemonic names. However if the reader feels that some ambiguity exists a description of these functions and macros is supplied in appendix five.

```
(declare (special :noticers* *removed-list* *contradictions* vtrace ctrace ntrace
                  *true-node* *designators* *think-queue* *name-queue*
                  think-trace
                  *better-pred* *coincidence*))

(deftype (designator)
  desig-hash-val
  expansion
  eq-links
  marker
  eq-origin-node
  translation
  sub-memo-node
  memoizations
  stimulate-list
  uniqueness
  opaqueness
  weight
  plunk-weight)

(defmacro primitive? (name)
     '(atom (expansion ,name)))

(defmacro number? (name)
     '(numberp (expansion ,name)))

(defstruct (marker)
           origin
           canonical
           canonical-node
           active-node)

(defun names-init (better-pred coincidence)
       (tms-init)
       (hash-array '*designators* 2047)
       (name-type 'designator)
       (name-type 'marker)
       (setq think-trace nil)
       (setq *think-queue* nil)
       (setq *name-queue* nil)
       (setq *better-pred* better-pred)
       (setq *coincidence* coincidence)
       (setq *true-node* (dependency-node 'true-by-definition))
       (set-truth *true-node* 'true 'definition))
```

```
(defun designator (exp)
       (designator2 (>>expand exp)))

(defun designator2 (exp)
    (if (designator? exp)
        exp
        (let# ((designator-exp (if (atom exp) exp (mapcar 'designator2 exp)))
               #as (table-assoc designator-exp '#designators#)))
          (if (cdr as)
              (cdr as)
              (let ((n-designator (make-designator)))
                   (rplacd as n-designator)
                   (setf (expansion n-designator) designator-exp)
                   (setf (translation n-designator)
                         (if (atom exp)
                             exp
                             (mapcar '(lambda (des) (translation des))
                                     designator-exp)))
                   (name-inspect n-designator)
                   n-designator)))))

(defun name-inspect (name)
       (if (not (primitive? name))
           (let ((op (expansion (car (expansion name)))))
                (if (atom op)
                    (mapc '(lambda (oper)
                                   (addf '(,oper ',name) (stimulate-list name)))
                          (operator-defs op))))))

(defun >>expand (exp)
       (cond ((or (atom exp) (designator? exp))
              exp)
             ((eq (car exp) '>>)
              (>>expand-2 (cdr exp)))
             (t (cons (>>expand (car exp))
                      (>>expand (cdr exp))))))

(defun >>expand-2 (exp)
       (cond ((null (cdr exp))
              (>>expand (car exp)))
             (t (list (>>expand (car exp))
                      (>>expand-2 (cdr exp))))))
```

```
(defun marked? (des)
   (let ((ml (marker des)))
     (and ml
          (true? (active-node ml))
          (true? (eq-origin-node des))
          (let ((can (canonical ml)))
            (and (eq ml (marker can))
                 --(true? (eq-origin-node can)))))))

(defmacro insure-marked (des)
     `(if (not (marked? ,des)) (initiate-marker ,des)))

(defun class-name (des)
    (insure-marked des)
    (canonical (marker des))

(defun what-is (exp)
    (let ((des (designator exp)))
         (insure-marked des)
         (think)
         (translation (class-name des))))

(defun insure-all-marked (desigs)
     (do ((rest desigs (cdr rest)))
         ((cond ((null rest) t)
                ((not (marked? (car desigs)))
                 (initiate-marker (car desigs))
                 (insure-all-marked desigs)
                 t)))))

(defun image (des)
    (cond ((primitive? des) des)
          (t (insure-all-marked (expansion des))
             (let ((exp (expansion des)))
               (let ((im-exp (mapcar 'class-name exp)))
                 (if (equal im-exp exp) des (designator im-exp)))))))
```

```
(defun canonical? (des)
  (insure-marked des)
  (let* ((m1 (marker des))
         (node1 (eq-origin-node des))
         (node2 (active-node m1))
         (node3 (canonical-node m1)))
        (if (and (true? node1)
                 (true? node2)
                 (eq des (canonical m1)))
            (list node1 node2 node3))))

(defun intern-canonical? (des)
   (if (primitive? des)
       (list *true-node*)
       (do ((desigs (expansion des) (cdr desigs))
            (nodes nil))
           ((null desigs) nodes)
           (let ((nodes2 (canonical? (car desigs))))
                (if nodes2
                    (setq nodes (nconc nodes nodes2))
                    (return nil))))))

(defun equal? (des1 des2)
       (cond ((eq des1 des2) (list *true-node*))
             (t (insure-all-marked (list des1 des2))
                (if (eq (marker des1) (marker des2))
                    (list (eq-origin-node des1) (eq-origin-node des2))))))

(defun intern-equal? (des1 des2)
       (if (eq des1 des2)
           *true-node*
           (let ((exp1 (expansion des1))
                 (exp2 (expansion des2)))
             (if (and (not (atom exp1))
                      (not (atom exp2))
                      (= (ncdrs exp1) (ncdrs exp2)))
                 (do ((desigs1 exp1 (cdr desigs1))
                      (desigs2 exp2 (cdr desigs2))
                      (nodes nil))
                     ((null desigs1) nodes)
                     (let ((nodes2 (equal? (car desigs1) (car desigs2))))
                          (if nodes2
                              (setq nodes (nconc nodes nodes2))
                              (return nil))))))))
```

```
(defun == (exp1 exp2)
       (equality (designator exp1) (designator exp2)))

(defun equality (name1 name2)
       (if (eq name1 name2)
           #true-node#
           (let ((eq-node (equality-node name1 name2)))
             (cond ((not (evaluated? eq-node))
                    (evaluated! eq-node)
                    (addf (cons name2 eq-node) (eq-links name1))
                    (addf (cons name1 eq-node) (eq-links name2))
                    (true-noticer eq-node '(link-check ',name1 ',name2 ',eq-node))
                    (let ((eq-nodes (equal? name1 name2)))
                         (if eq-nodes
                             (implies eq-nodes eq-node)))))
                   eq-node)))

(defun equality-node (des1 des2)
       (dependency-node (if (eq 'less (scomp (translation des1) (translation des2)))
                            `(== ,(translation des1) ,(translation des2))
                            `(== ,(translation des2) ,(translation des1)))))

(defun link-check (des1 des2 eq-node)
   (if (marked? des1)
       (if (marked? des2)
           (if (eq (marker des1) (marker des2))
               (implies (list (eq-origin-node des1)
                              (eq-origin-node des2))
                        eq-node)
               (if (true? eq-node)
                   (if (funcall #better-pred#
                                (canonical (marker des1))
                                (canonical (marker des2)))
                       (mark des2 (marker des1) (eq-origin-node des1) eq-node)
                       (mark des1 (marker des2) (eq-origin-node des2) eq-node))))
         . (if (true? eq-node)
               (mark des2 (marker des1) (eq-origin-node des1) eq-node)))
       (if (and (marked? des2) (true? eq-node))
           (mark des1 (marker des2) (eq-origin-node des2) eq-node))))
```

```
(defun initiate-marker (des)
    (let ((m1 (make-marker)))
         (setf (origin m1) des)
         (setf (canonical m1) des)
         (let ((node1 (dependency-node '(canonical ,m1 ,(translation des))))
               (node2 (dependency-node '(active ,m1))))
              (setf (canonical-node m1) node1)
              (set-truth node1 'true 'initiator)
              (setf (active-node m1) node2)
              (set-truth node2 'true 'initiator)
              (mark-2 des m1 #true-node#)
              m1)))

(defun mark (des m1 eq-origin-node1 eq-node)
       (let ((eq-origin-node2 (equality-node des (origin m1))))
            (implies (list eq-origin-node1 eq-node)
                     eq-origin-node2)
            (let ((can (canonical m1)))
                 (if (funcall #better-pred# des can)
                     (make-canonical m1 des))
                 (addf '(,#coincidence# ',can ',des) #think-queue#))
            (mark-2 des m1 eq-origin-node2)))

(defun make-canonical (m1 des)
       (let ((#noticers# nil)
             (#removed-list# nil)
             (node (dependency-node '(canonical ,m1 ,(translation des)))))
            (remove-2 (canonical-node m1))
            (setf (canonical m1) des)
            (setf (canonical-node m1) node)
            (set-2 node 'true 'premise 'marker)
            (removed-check)
            (run-noticers)))

(defun mark-2 (des m1 node)
       (let ((m2 (marker des)))
            (if (and m2 (not (eq m1 m2)) (true? (active-node m2)))
                (remove-truth (active-node m2))))
       (setf (marker des) m1)
       (setf (eq-origin-node des) node)
       (stimulate des)
       (mapc '(lambda (link) (link-check des (car link) (cdr link)))
             (eq-links des)))
```

```
001
002  (declare (special done-list))
003
004  (defun equiv-class (des)
005         (let ((done-list nil))
006              (equiv-class2 des)))
007
008  (defun equiv-class2 (des)
009         (cond ((mat (memq des done-list))
010                (setq done-list (cons des done-list))
011                (cons des
012                      (mapcan '(lambda (link) (if (true? (cdr link))
013                                                  (equiv-class2 (car link))))
014                              (eq-links des))))))
015
016  (defun view-all (exp)
017         (nmapcar 'translation (equiv-class (designator exp))))
018
019  (defun view-class (exp)
020         (lmapcar '(lambda (des2) (if (intern-canonical? des2) (translation des2)))
021                  (equiv-class (designator exp))))
022
```

```
(defun substitute (des)
       (let ((des2 (image des)))
         (let ((eq-nodes (intern-equal? des des2))
               (can-nodes (intern-canonical? des2)))
           (if (or (null eq-nodes) (null can-nodes))
               (break (substitution failure)))
           (if think-trace
               (print '(image ,(translation des) ,(translation des2))))
           (cond ((not (eq des des2))
                  (implies eq-nodes (equality des des2))
                  (let ((mem-node (dependency-node '(intern-equal
                                                     ,(translation des)
                                                     ,(translation des2)))))
                    (unknown-noticer mem-node '(stimulate ',des))
                    (implies eq-nodes mem-node)
                    (setf (sub-memo-node des) mem-node))))
           (auto-sub des2 can-nodes))))

(defun auto-sub (des can-nodes)
    (let ((mem-node2 (dependency-node '(intern-canonical ,(translation des)))))
      (unknown-noticer mem-node2 '(stimulate ',des))
      (implies can-nodes mem-node2)
      (setf (sub-memo-node des) mem-node2)))
```

```
(defun stimulate (des)
    (if (or (null *name-queue*)
            (not (funcall *better-pred* (car *name-queue*) des)))
        (addf des *name-queue*)
        (insert des *name-queue*)))

(deftail insert (des queue)
       (if (or (null (cdr queue))
               (not (funcall *better-pred* (cadr queue) des)))
           (addf des (cdr queue))
           (insert des (cdr queue))))

(deftail think ()
    (cond (*think-queue*
           (let ((closure (car *think-queue*)))
                (setq *think-queue* (cdr *think-queue*))
                (if think-trace (/#print (list-translation closure)))
                (eval closure)
                (think)))
          (*name-queue*
           (let ((des (car *name-queue*)))
                (setq *name-queue* (cdr *name-queue*))
                (if (and (not (primitive? des))
                         (not (true? (sub-memo-node des))))
                    (substitute des))
                (if (eq des (class-name des))
                    (think2 des))
                (think)))))

(deftail think2 (des)
     (let ((closure (car (stimulate-list des))))
       (cond (closure
              (delf closure (stimulate-list des))
              (if think-trace (/#print (list-translation closure)))
              (eval closure)
              (think2 des)))))

(defun print-val (exp)
       (let ((des (designator exp)))
            (insure-marked des)
            (think)
            (/#print `((the value of)
                       ,(translation des)
                       is
                       ,(translation (class-name des))))))
```

```
(defun fund-coincidence (des1 des2)
       nil)

(defun fund-better (des1 des2)
       (eq 'less (fund-comp des1 des2)))

(defun fund-comp (des1 des2)
       (comp (superior-class 'unique? des1 des2)
             'less
             (comp (superior-class 'opaque? des1 des2)
                   'greater
                   (num-comp (virt-weight des1) (virt-weight des2))
                   'less)
             'greater))



(defun unique? (des)
  (let ((unp (uniqueness des)))
       (or (eq unp 'true)
           (and (not (eq unp 'false))
                (cond ((numberp (expansion des))
                       (setf (uniqueness des) 'true)
                       t)
                      ((primitive? des)
                       (setf (uniqueness des) 'false)
                       nil)
                      ((all-unique? (expansion des))
                       (setf (uniqueness des) 'true)
                       t)
                      (t (setf (uniqueness des) 'false)
                        nil)))))))

(defun all-unique? (desigs)
       (or (null desigs)
           (and (unique? (car desigs))
                (all-unique? (cdr desigs)))))

(defun unique! (exp)
    (let ((des (designator exp)))
      (if (uniqueness des)
          (break (unique! applied to an uniqueness determined object))
          (setf (uniqueness des) 'true))))
           .
```

```
001
002  (defmac opaque? (des)
003    (let ((opq (opaqueness des)))
004      (or (eq opq 'true)
005          (and (not (eq opq 'false))
006               (cond ((primitive? des)
007                      (setf (opaqueness des) 'false)
008                      nil)
009                     ((some-opaque? (expansion des))
010                      (setf (opaqueness des) 'true)
011                      t)
012                     (t (setf (opaqueness des) 'false)
013                        nil)))))))
014
015  (defun some-opaque? (desigs)
016         (and desigs
017              (or (opaque? (car desigs))
018                  (some-opaque? (cdr desigs)))))
019
020  (defun opaque! (exp)
021      (let ((des (designator exp)))
022        (if (opaqueness des)
023            (break (opaque! applied to an opaqueness determined object))
024            (setf (opaqueness des) 'true))))
025
026
027
028
029
030  (defmac virt-weight (des)
031    (let ((wgt (weight des)))
032      (if wgt
033          wgt
034          (cond ((primitive? des)
035                 (setf (weight des) 1)
036                 1)
037                (t (let ((wgt (sum-weight (expansion des))))
038                     (setf (weight des) wgt)
039                     wgt)))))))
040
041  (deftail sum-weight (def)
042         (if (null def)
043             0
044             (+ (virt-weight (car def))
045                (sum-weight (cdr def)))))
046
```

```
001
002 (defmacro operator-defs (op)
003     '(get ,op 'operator-defs))
004
005 (defmacro defoper (oper args . body)
006     (let ((dos (namcopy 'dog)))
007       (addf '((lambda (,dos)
008                 (let ((,args (cdr (translation ,dos))))
009                   `,@(mapcar '(lambda (form) `(set-math ,form 'true 'definition))
010                              body)))
011               (operator-defs oper))
012       ))
013
```

```
==  ..... EXPR .... 005 002
>>EXPAND  ..... EXPR .... 002 031
>>EXPAND-2  ..... EXPR .... 002 039
ALL-UNIQUE?  ..... EXPR .... 010 035
AUTO-SUB  ..... EXPR .... 008 020
CANONICAL?  ..... EXPR .... 004 002
CLASS-NAME  ..... EXPR .... 003 014
DEFOPER  ..... DEFMACRO  012 005
DESIGNATOR  ..... EXPR .... 002 002
DESIGNATOR  ..... DEFTYPE  001 007
DESIGNATOR2  ..... EXPR .... 002 005
EQUAL?  ..... EXPR .... 004 024
EQUALITY  ..... EXPR .... 005 005
EQUALITY-NODE  ... EXPR .... 005 019
EQUIV-CLASS  ..... EXPR .... 007 004
EQUIV-CLASS2  .... EXPR .... 007 008
FUND-BETTER  ..... EXPR .... 010 005
FUND-COINCIDENCE  EXPR .... 010 002
FUND-COMP  ..... EXPR .... 010 008
IMAGE  ..... EXPR .... 003 032
INITIATE-MARKER  EXPR .... 006 002
INSERT  ..... DEFTAIL  009 008
INSURE-ALL-MARKED EXPR .... 003 024
INSURE-MARKED  ... DEFMACRO  003 011
INTERN-CANONICAL?  EXPR .... 004 013
INTERN-EQUAL?  ... EXPR .... 004 030
LINK-CHECK  ..... EXPR .... 005 024
MAKE-CANONICAL  .. EXPR .... 006 025
MARK  ..... EXPR .... 006 015
MARK-2  ..... EXPR .... 006 036
MARKED?  ..... EXPR .... 003 002
MARKER  ..... DEFSTRUCT 001 028
NAME-INSPECT  .... EXPR .... 002 023
NAMES-INIT  ..... EXPR .... 001 034
NUMBER?  ..... DEFMACRO  001 025
OPAQUE!  ..... EXPR .... 011 020
OPAQUE?  ..... DEFMAC .. 011 002
OPERATOR-DEFS  ... DEFMACRO  012 002
PRIMITIVE?  ..... DEFMACRO  001 022
PRINT-VAL  ..... EXPR .... 003 039
SOME-OPAQUE?  .... EXPR .... 011 015
STIMULATE  ..... EXPR .... 009 002
SUBSTITUTE  ..... EXPR .... 008 002
SUM-WEIGHT  ..... DEFTAIL  011 041
THINK  ..... DEFTAIL  009 014
THINK2  ..... DEFTAIL  009 031
UNIQUE!  ..... EXPR .... 010 040
UNIQUE?  ..... EXPR .... 010 019
VIEW-ALL  ..... EXPR .... 007 016
VIEW-CLASS  ..... EXPR .... 007 019
VIRT-WEIGHT  ..... DEFMAC .. 011 030
WHAT-IS  ..... EXPR .... 003 018
```

# Appendix Three
## The Algebraic Simplifier

Algebraic simplification is done in the equality system in the same way as numerical computation. Evaluation functions are placed on algebraic operators which generate equalities between algebraic expressions and simplified forms of those expressions. Simplified versions of algebraic expressions are generated with a simplification function which operates on algebraic expressions and is completely separate from the equality system and the TMS. This section discusses the algorithms employed by that simplifier.

There are some well known algorithms for handling the simplification of ratios of polynomials [Knuth 69]. These algorithms were implemented in a symbolic mathematical package developed at MIT [MACSYMA 77], and from there incorporated into a recent electrical circuit analysis system (SYN) developed by Johan de Kleer and Gerald Sussman [de Kleer & Sussman 78]. In the SYN system all expressions are converted to a canonical form which is a ratio of two polynomials. The variables are given global priorities and the polynomials in a canonical representation are polynomials in the highest priority variable occurring in them. The coefficients of such polynomials are canonical polynomials in the other variables. Synthetic division and greatest common divisor algorithms are used to remove common factors from the numerator and denominator of a ratio of two polynomials. This canonical form is truly canonical in that any two forms of a rational function will simplify to the same canonical expression. In this way simplification can be guaranteed to the extent that any expression can be placed in canonical form.

A major problem with the symbolic algebra routines used in SYN is that computation of the greatest common divisor (gcd) of two polynomials in many variables is a very expensive operation. It was found in fact that this was the major limitation on the complexity of the circuits which could be analyzed in SYN. While recent developments in polynomial gcd techniques may improve this situation [Zippel 79], the simplification routines used in the equality system represent an attempt to avoid heavy reliance on gcd algorithms.

Instead of reducing all expressions to a canonical polynomial form the algebraic simplifier used in the equality system is capable of maintaining factored forms of expressions. This allows factors to be removed from the numerator and denominator of a ratio by a simple search for common factors. It is not practical to factor all polynomials since this is much harder than computing gcds, but once

an expression appears in factored form that factorization is not lost. This has the disadvantage that expressions are not placed in a truly canonical form, i.e. two expressions which are really the same might be simplified differently if one was more factored than the other.

In the equality system simplifier expressions are placed in pseudo-canonical form which is actually very similar to the canonical form used in SYN. Each expression in this pseudo-canonical form is a ratio of two *products* of polynomials. The variables have global priorities and polynomials are always polynomials in the highest priority variables appearing in them. The coefficients of these polynomials are again *products* of polynomials in lower priority variables and factors appearing in all the coefficients are factored out of the polynomials. Factors which are common to both the numerator and denominator of an expression are removed from a ratio, but no attempt to find hidden common factors via a gcd algorithm has been incorporated.

It would be very simple to add the ability to factor simple quadratics. It would also be possible to incorporate a gcd algorithm to guarantee complete removal of common factors from the coefficients and cancellations of all factors common to the numerator and denominator of a ratio. The polynomials to which the gcd would be applied would be considerably simpler in the system developed here than in a system which does not maintain factored forms.

The simplification system uses an architecture developed by Gerald Sussman in which the basic simplification mechanism is a set of "symbolic operators" which compute the pseudo-canonical form for the sum, difference, product or ratio of expressions already in canonical form. A procedure for simplifying arbitrary expressions can be built which uses the symbolic operators to recursively simplifying subexpressions and then applies the symbolic operator corresponding to the top level operation. Products and ratios of expressions in pseudo-canonical form are very easy to handle since all that must be done is the merging of factors and the cancellation of factors common to the resulting numerator and denominator. Subtraction is trivially reduced to multiplication and addition, which is the most involved symbolic operation.

Addition is done in two phases. First the expression is converted, in a straightforward way, to a ratio of a sum of two products and a product. The second step is the addition of the two products in the numerator. The first step in the addition of products is to remove common factors. Next, the highest priority variable of the two products is found, and each product is expanded into a polynomial in this variable. The coefficients of these polynomials are then

recursively added term by term. Finally factors which are common to all the coefficients of the resulting polynomial are removed a ıd the result of the original addition is a product of polynomials.

If any product or polynomial can be simplified to zero this algorithm is guaranteed to do that simplification. This can be shown by observing that if any of the factors of a product simplify to zero then the product will also, and that all the factors are polynomials in some variable. A polynomial in a given variable will simplify to zero only if all of its coefficients do, and these coefficients are either numeric or are simpler products of polynomials which, by an induction hypothesis, simplify to zero if such a simplification is possible.

In the equality system each subexpression of an expression is a designator in its own right and can have properties attached to it. This can result in substantial space savings since all references to a particular expression point to the same data structure. This also makes it easy to memoize the simplification process which can result in significant time savings when an expression appears inside several different larger expressions or several times within the same expression.

Another important point about the algebraic simplification system is that subexpressions which do not have top level algebraic operators are treated as variables. This allows the algebraic system to manipulate arbitrary functional expressions, which it has no knowledge of. For example there is no problem in dealing with (sin x) or (ln (** x y)). These sub-expressions can be dealt with independently in the equality system, which is capable of substituting simplifications of them into algebraic expressions.

The code for the algebraic simplification routines follow.

```
(defun operator (exp)
       (if (not (algebraic? exp))
            nil
            (car exp)))

(defun operands (exp)
       (if (not (algebraic? exp))
            nil
            (cdr exp)))

(defun make-prod (scale sumlist)
       (cons '* (cons scale sumlist)))

(defun numer-prod (exp)
       (cond ((numberp exp) (make-prod exp nil))
             ((not (algebraic? exp)) (make-prod 1 (list exp)))
             (t (let ((op (operator exp)))
                     (cond ((eq op '// ) (numer-prod (cadr exp)))
                           ((eq op '*) exp)
                           (t (make-prod 1 (list exp))))))))

(defun denom-prod (exp)
       (cond ((not (algebraic? exp)) (make-prod 1 nil))
             (t (let ((op (operator exp)))
                     (cond ((eq op '// ) (numer-prod (caddr exp)))
                           (t (make-prod 1 nil)))))))

(defun pnf_zero? (exp)
       (= 0 (car (operands (numer-prod exp)))))

(defun algebraic? (exp)
       (and (not (atom exp))
            (not (numberp exp))
            (memq (car exp) '(+ - * // ))))

(defun numerical? (exp)
       (or (numberp exp)
           (let ((op (operator exp)))
                (cond ((eq '* op) (null (cdr (operands exp))))
                      ((eq 'rat (operator exp)) t)
                      ((eq '// (operator exp))
                       (and (numerical? (numer-prod exp))
                            (numerical? (denom-prod exp))))))))

(defun variable? (exp)
       (not (or (algebraic? exp)
                (numerical? exp))))

(defun sign (prod)
       (let ((n (cadr prod)))
            (make-prod (// n (abs n)) nil)))

(declare (special *var-comp*))

(setq *var-comp* 'scomp)

(defun make-best-var (var old-var-comp)
       `(lambda (var1 var2)
                (comp (superior-class '(lambda (var3) (eq var3 ',var))
                                      var1 var2)
                      'less
                      (,old-var-comp var1 var2)
                      'greater)))

(defun poly-comp (poly1 poly2)
       (comp (funcall *var-comp* (poly-var poly1) (poly-var poly2))
             'greater
             (scomp poly1 poly2)
             'less))
```

```
(defun *_*_* (prod1 prod2)
       (let ((factors1 (operands prod1))
             (factors2 (operands prod2)))
            (make-prod (* (car factors1) (car factors2))
                       (comb-factors (cdr factors1) (cdr factors2)))))

(defun comb-factors* (factors1 factors2)
       (cond ((null factors1) factors2)
             ((null factors2) factors1)
             (t (comp (poly-comp (car factors1) (car factors2))
                      (cons (car factors2)
                            (comb-factors factors1 (cdr factors2)))
                      (cons (car factors1)
                            (cons (car factors2)
                                  (comb-factors (cdr factors1) (cdr factors2))))
                      (cons (car factors1)
                            (comb-factors (cdr factors1) factors2))))))

(defun gcd_*_* (prod1 prod2)
  (if (= 0 (cadr prod1))
      (make-prod 1 nil)
      (make-prod (gcd (cadr prod1) (cadr prod2))
                 (prod-intersection (cddr prod1) (cddr prod2)))))

(defun prod-intersection (factors1 factors2)
       (cond ((null factors1) nil)
             ((null factors2) nil)
             (t (comp (poly-comp (car factors1) (car factors2))
                      (prod-intersection factors1 (cdr factors2))
                      (cons (car factors1)
                            (prod-intersection (cdr factors1) (cdr factors2)))
                      (prod-intersection (cdr factors1) factors2)))))

; this division algorithm assumes that prod1 is an explicit multiple of prod2.

(defun //_*_* (prod1 prod2)
       (make-prod (// (cadr prod1) (cadr prod2))
                  (factor-deletion (cddr prod1) (cddr prod2))))

(defun factor-deletion (factors1 factors2)
       (cond ((null factors1) nil)
             ((null factors2) factors1)
             (t (comp (poly-comp (car factors1) (car factors2))
                      (factor-deletion factors1 (cdr factors2))
                      (factor-deletion (cdr factors1) (cdr factors2))
                      (cons (car factors1)
                            (factor-deletion (cdr factors1) factors2))))))
```

```
(defun terms (poly)
       (if (variable? poly)
           (list (make-prod 1 (list poly)))
           (operands poly)))

(defun poly-var (poly)
       (cond ((null poly) nil)
             ((variable? poly) poly)
             ((numerical? poly) nil)
             (t (let ((first (cadr (operands (car (operands poly))))))
                     (cond ((variable? first) first)
                           (t (break non-polynomial-sum)))))))

(defun poly-mult (terms1 terms2 var)
       (cond ((null var) (break 'null-var-in-poly-mult))
             ((null terms1) nil)
             (t (let ((first (car terms1)))
                     (poly-add (mapcar '(lambda (prod) (*_*_* first prod))
                                       terms2)
                               (poly-mult (cdr terms1) terms2 var)
                               var))))))

(defun poly-add (terms1 terms2 var)
       (let ((result (poly-add2 terms1 terms2 var)))
            (cond ((pnf_zero? (car result))
                   (cdr result))
                  (t result))))

(defun poly-add2 (terms1 terms2 var)
      (cond ((null var) (break 'null-var-in-poly-add))
             ((null terms1) terms2)
             ((null terms2) terms1)
             (t (let ((order1 (order2 (cdr (operands (car terms1))) var))
                      (order2 (order2 (cdr (operands (car terms2))) var)))
                     (cond ((= order1 order2)
                            (cons (prodsum (numer-prod (car terms1))
                                           (numer-prod (car terms2)))
                                  (poly-add (cdr terms1) (cdr terms2) var)))
                           ((> order1 order2)
                            (cons (car terms1) (poly-add (cdr terms1) terms2 var)))
                           (t (cons (car terms2) (poly-add terms1 (cdr terms2) var)))))))))

(defun order (poly)
       (order2 (cdr (operands (car (operands poly)))) (poly-var poly)))

(defun order2 (factors var)
       (if (eq (car factors) var) (1+ (order2 (cdr factors) var)) 0))
```

```
001
002  (defun coeficients (poly)
003        (cond ((variable? poly) (list (make-prod 0 nil)
004                                      (make-prod 1 nil)))
005              ((numberp poly) (list (make-prod poly nil)))
006              (t (coef2 (reverse (operands poly))
007                        (poly-var poly)
008                    -   0))))
009                  --
010  (defun coef2 (terms var init-order)
011        (if (null terms)
012            nil
013            (let ((ol (order2 (cdr (operands (car terms))) var)))
014                 (if (= ol init-order)
015                     (let ((p-terms (operands (car terms))))
016                          (cons (make-prod (car p-terms)
017                                           (coef3 (cdr p-terms) var))
018                                (coef2 (cdr terms) var (1+ init-order))))
019                     (cons 0 (coef2 terms var (1+ init-order)))))))
020
021  (defun coef3 (terms var)
022        (if (eq (car terms) var)
023            (coef3 (cdr terms) var)
024            terms))
025
```

```
001
002  (defun -_pnf_1 (exp)
003         (let ((n (numer-prod exp)))
004              (list '//
005                    (make-prod (* -1 (cadr n)) (cddr n))
006                    (denom-prod exp))))
007
008  (defun -_pnf (args)
009         (if (cdr args)
010             (+_pnf (list (car args)
011                          (-_pnf_1 (cadr args))))
012             (-_pnf_1 (car args))))
013
014  (defun 1// (exp)
015         (list '// (denom-prod exp) (numer-prod exp)))
016
017  (defun //_pnf (args)
018         (*_pnf (list (car args) (1// (cadr args)))))
019
020  (defun *_pnf (args)
021         (cond ((= (length args) 1) (car args))
022               (t (simprod (car args) (*_pnf (cdr args))))))
023
024  (defun simprod (exp1 exp2)
025         (let ((top-prod (*_*_* (numer-prod exp1)
026                                (numer-prod exp2)))
027               (bottom-prod (*_*_* (denom-prod exp1)
028                                   (denom-prod exp2))))
029              (let ((common-prod (gcd_*_* top-prod bottom-prod)))
030                   (list '//
031                         (//_*_* top-prod common-prod)
032                         (//_*_* bottom-prod common-prod)))))
033
034  (defun +_pnf (args)
035         (cond ((= (length args) 1) (car args))
036               (t (simpsum (car args) (+_pnf (cdr args))))))
037
```

```
(defun simpsum (exp1 exp2)
        (let ((top1 (*_*_* (numer-prod exp1) (denom-prod exp2)))
              (top2 (*_*_* (numer-prod exp2) (denom-prod exp1))))
             (let ((top-prod (prodsum top1 top2))
                   (bottom-prod (*_*_* (denom-prod exp1) (denom-prod exp2))))
                  (let ((common-prod (gcd_*_* top-prod bottom-prod)))
                       (if (not (pnf_zero? top-prod))
                           (list '//
                                 (//_*_* top-prod common-prod)
                                 (//_*_* bottom-prod common-prod))
                           (if (pnf_zero? bottom-prod)
                               (break indeterminate-value)
                               (list '// top-prod 1)))))))

(defun prodsum (prod1 prod2)
       (let ((com-fact (com-factor (list prod1 prod2))))
            (let ((rest-prod (con-sum (//_*_* prod1 com-fact)
                                      (//_*_* prod2 com-fact))))
                 (if (= (car (operands rest-prod)) 0)
                     (make-prod 0 nil)
                     (*_*_* com-fact rest-prod))))))

(defun con-sum (term1 term2)
     (if (and (numerical? term1) (numerical? term2))
         (make-prod (+ (car (operands term1)) (car (operands term2))) nil)
         (let ((var1 (poly-var (cadr (operands term1))))
               (var2 (poly-var (cadr (operands term2)))))
              (cond ((eq var1 var2)
                     (factor (poly-add (expand term1) (expand term2) var1)))
                    ((or (null var2) (and var1 (eq 'less
                                                   (funcall *var-comp* var1 var2))))
                     (factor (poly-add (expand term1) (list term2) var1)))
                    (t (factor (poly-add (expand term2) (list term1) var2))))))))

(defun expand (prod)
       (let ((opers (operands prod)))
            (if (null (cdr opers))
                (list prod)
                (expand2 (car opers) (cdr opers)))))

(defun expand2 (scale factors)
       (cond ((null factors) (list (make-prod scale nil)))
             ((eq (poly-var (car factors)) (poly-var (cadr factors)))
              (poly-mult (terms (car factors))
                         (expand2 scale (cdr factors))
                         (poly-var (car factors))))
             (t (let ((rest (make-prod scale (cdr factors))))
                     (mapcar '(lambda (prod) (*_*_* prod rest))
                             (terms (car factors)))))))

(defun factor (terms)
       (cond ((null terms) (make-prod 0 nil))
             ((null (cdr terms)) (car terms))
             (t (let ((common (*_*_* (sign (car terms)) (com-factor terms))))
                     (*_*_* common
                            (make-prod 1 (list (cons '+ (mapcar '(lambda (prod)
                                                                   (//_*_* prod common))
                                                                terms)))))))))

(defun com-factor (sumlist)
       (com-factor2 (car sumlist) (cdr sumlist)))

(defun com-factor2 (factor sumlist)
       (cond ((null sumlist) factor)
             (t (com-factor2 (gcd_*_* factor (car sumlist)) (cdr sumlist)))))
```

```
;solve returns a dotted pair.
;the car of the pair is a list of sums whoes product must not be zero.
;the cdr of the pair is a list of roots of the equaltion, they are not in canonical
;form however and must be simplified before being passed as arguments to symop functions.

(defun solve (exp var)
      (let* ((*var-comp* (make-best-var var *var-comp*))
             (exp2 (simplification exp)))
            (if (or (pnf_zero? (numer-prod exp2))
                    (pnf_zero? (denom-prod exp2)))
                nil
                (let (((non-zero . roots)
                       (solve2 (cdr (operands (numer-prod exp2))) var)))
                     (cons (append (mapcar 'alg-trans (cdr (operands (denom-prod exp2))))
                                   non-zero)
                           roots)))))

(defun solve2 (factors var)
       (if factors
           (let (((non-zero . roots) (solve2 (cdr factors) var))
                 (sum (car factors)))
                (if (eq var (poly-var sum))
                    (let* ((coefs (coeficients sum)))
                          (cons (append (mapcar 'alg-trans (cdr (operands (car (last coefs)))))
                                        non-zero)
                                (cons (if (cddr coefs)
                                          `(a-root-of ,@(mapcar 'alg-trans coefs))
                                          (alg-trans (//_pnf (list (-_pnf_1 (car coefs))
                                                                   (cadr coefs)))))
                                      roots)))
                    (cons (cons (alg-trans sum) non-zero)
                          roots)))))
```

```
(defun variables (exp)
       (merge (variables-2 exp) nil))

(defun variables-2 (exp)
       (cond ((algebraic? exp)
              (mapcan 'variables-2 (operands exp)))
             ((numerical? exp) nil)
             (t_(list exp))))

(defun simplification (exp)
       (if (not (algebraic? exp))
           exp
           (let ((args (mapcar 'simplification (operands exp)))
                 (op (operator exp)))
                (cond ((eq op '+) (+_pnf args))
                      ((eq op '-) (-_pnf args))
                      ((eq op '*) (*_pnf args))
                      ((eq op '// ) (//_pnf args))))))

;the next function makes expressions more readible by removing factors of 1 etc.

(defun alg-trans (exp)
       (cond ((not (algebraic? exp))
              exp)
             (t (let ((op (operator exp))
                      (opers (mapcar 'alg-trans (operands exp))))
                     (cond ((eq op '// )
                            (cond ((equal (cadr opers) 1) (car opers))
                                  ((equal (cadr opers) -1)
                                   (cond ((numberp (car opers))
                                          (* -1 (car opers)))
                                         ((not (algebraic? (car opers)))
                                          '(* -1 ,(car opers)))
                                         ((eq (operator (car opers)) '*)
                                          '(* ,(* -1 (cadar opers)) ,(caddar opers)))
                                         (t '(* -1 ,(car opers)))))
                                  (t (cons '// opers))))
                           ((eq op '+) (cons '+ opers))
                           ((eq op '*)
                            (cond ((equal (car opers) 0) 0)
                                  ((equal (car opers) 1)
                                   (cond ((null (cdr opers))
                                          1)
                                         ((null (cddr opers))
                                          (cadr opers))
                                         (t (cons '* (cdr opers)))))
                                  ((null (cdr opers))
                                   (car opers))
                                  (t (cons '* opers)))))))))

(defun simpt (exp)
  (alg-trans (simplification exp)))
```

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *VAR-COMP* | SETQ | 001 | 056 | CON-SUM | EXPR | 007 | 024 | POLY-ADD2 | EXPR | 003 | 030 |
| *_*_* | EXPR | 002 | 002 | DENOM-PROD | EXPR | 001 | 023 | POLY-COMP | EXPR | 001 | 066 |
| *_PNF | EXPR | 006 | 020 | EXPAND | EXPR | 007 | 036 | POLY-MULT | EXPR | 003 | 015 |
| +_PNF | EXPR | 006 | 034 | EXPAND2 | EXPR | 007 | 042 | POLY-VAR | EXPR | 003 | 007 |
| -_PNF | EXPR | 006 | 008 | FACTOR | EXPR | 007 | 052 | PROD-INTERSECTION | EXPR | 002 | 026 |
| -_PNF_1 | EXPR | 006 | 002 | FACTOR-DELETION | EXPR | 002 | 041 | PRODSUM | EXPR | 007 | 016 |
| /_*_* | EXPR | 002 | 037 | GCD_*_* | EXPR | 002 | 020 | SIGN | EXPR | 001 | 050 |
| /_PNF | EXPR | 006 | 017 | MAKE-REST-VAR | EXPR | 001 | 058 | SIMPLIFICATION | EXPR | 009 | 011 |
| 1/ | EXPR | 006 | 014 | MAKE-PROD | EXPR | 001 | 012 | SIMPROD | EXPR | 006 | 024 |
| ALG-TRANS | EXPR | 009 | 023 | NUMER-PROD | EXPR | 001 | 015 | SIMPSUM | EXPR | 007 | 002 |
| ALGEBRAIC? | EXPR | 001 | 032 | NUMERICAL? | EXPR | 001 | 037 | SIMPT | EXPR | 009 | 052 |
| COEF2 | EXPR | 004 | 010 | OPERANDS | EXPR | 001 | 007 | SOLVE | EXPR | 008 | 007 |
| COEF3 | EXPR | 004 | 021 | OPERATOR | EXPR | 001 | 002 | SOLVE2 | EXPR | 008 | 019 |
| COEFICIENTS | EXPR | 004 | 002 | ORDER | EXPR | 003 | 044 | TERMS | EXPR | 003 | 002 |
| COM-FACTOR | EXPR | 007 | 061 | ORDER2 | EXPR | 003 | 047 | VARIABLE? | EXPR | 001 | 046 |
| COM-FACTOR2 | EXPR | 007 | 064 | PNF_ZERO? | EXPR | 001 | 029 | VARIABLES | EXPR | 009 | 002 |
| COMB-FACTORS | EXPR | 002 | 008 | POLY-ADD | EXPR | 003 | 024 | VARIABLES-2 | EXPR | 009 | 005 |

[redacted]

```
(declare (special *zero* plunk-weight-count))

(defun algsys-init ()
       (names-init 'algsys-better 'algsys-coin)
       (mapc 'uniquel '(+ - // *))
       (setq *zero* (designator 0))
       (name-type 'plunk)
       (setq plunk-weight-count 1))

(putprop '+ '+_pnf 'symop)
(putprop '- '-_pnf 'symop)
(putprop '* '*_pnf 'symop)
(putprop '// '//_pnf 'symop)
(addf 'create-simplification (operator-defs '+))
(addf 'create-simplification (operator-defs '-))
(addf 'create-simplification (operator-defs '*))
(addf 'create-simplification (operator-defs '// ))

(defun create-simplification (des)
       (let ((simp (des-simplification des)))
         (let ((des2 (designator2 (alg-trans simp))))
              (associate 'simplification simp (memoizations des))
              (set-truth (equality des des2) 'true 'algebra))))

(defun des-simplification (des)
    (let ((exp (expansion des)))
      (cond ((atom exp) exp)
            ((not (memq (expansion  (car exp)) '(+ - // *)))
             (list-translation des))
            (t (let ((simp (assq 'simplification (memoizations des))))
                 (if simp
                     (cdr simp)
                     (let ((result (funcall (get (expansion (car exp)) 'symop)
                                            (mapcar 'des-simplification (cdr exp)))))
                       (addf (cons 'simplification result)
                             (memoizations des))
                       result)))))))

(defun c== (x y)
       (let* ((z-exp (alg-trans (des-simplification (designator '(- ,x ,y)))))
              (c-node (equality (designator2 z-exp) *zero*)))
             (mapc '(lambda (var) (solve-constraint z-exp var c-node))
                   (variables z-exp))
             c-node))

(defun solve-constraint (exp var c-node)
    (let ((t-var (designator2 var))
          ((non-zero . roots) (solve exp var)))
         (if (and (car roots) (null (cdr roots)))
             (if (null non-zero)
                 (implies (list c-node)
                          (equality t-var (designator2 (car roots))))
                 (let* ((non-z (if (cdr non-zero)
                                   (designator2 '(* ,@non-zero))
                                   (designator2 (car non-zero))))
                        (eq-node (equality non-z *zero*)))
                       (default eq-node 'false)
                       (implies (list (t-not eq-node) c-node)
                                (equality t-var (designator2 (car roots))))))))))
```

```
(defun algsys-better (des1 des2)
       (eq 'less (algsys-comp des1 des2)))

(defun algsys-comp (des1 des2)
       (comp (superior-class 'unique? des1 des2)
             'less
             (comp (superior-class 'opaque? des1 des2)
                   'greater
                   (comp (num-comp (virt-plunk-weight des1)
                                   (virt-plunk-weight des2))
                         'greater
                         (num-comp (virt-weight des1) (virt-weight des2))
                         'less)
                   'less)
             'greater))

(defun virt-plunk-weight (des)
  (let ((bpw (plunk-weight des)))
    (if bpw
        bpw
        (cond ((primitive? des)
               (setf (plunk-weight des) 0)
               0)
              (t (let ((bpw (biggest-plunk-weight (expansion des))))
                   (setf (plunk-weight des) bpw)
                   bpw))))))

(defun biggest-plunk-weight (desigs)
  (if (null (cdr desigs))
      (virt-plunk-weight (car desigs))
      (let ((bpw1 (biggest-plunk-weight (cdr desigs)))
            (bpw2 (virt-plunk-weight (car desigs))))
        (if (not (and bpw1 bpw2)) (break (null plunk weight)))
        (if (> bpw1 bpw2) bpw1 bpw2))))

(defun plunk! (exp)
  (print `(plunking ,exp))
    (let ((des1 (designator exp))
          (des2 (designator (intern (gename 'plunk)))))
      (setf (plunk-weight des2) plunk-weight-count)
      (increment plunk-weight-count)
      (set-truth (equality des1 des2) 'true 'definition)))
```

```
(defun algsys-coin (des1 des2)
       (if (eq des1 des2) (break (eq dess in algsys-coin)))
       (if (and (eq des1 (image des1))
                (eq des2 (image des2)))
           (let ((t-des1 (translation des1))
                 (t-des2 (translation des2)))
                (cond ((and (numberp t-des1) (numberp t-des2))
                --       (set-truth (equality des1 des2) 'false 'algebra))
                  -     ((and (numerical? t-des1)
                              (numerical? t-des2))
                         (let ((diff (des-simplification (designator2 '(- ,des1 ,des2)))))
                           (if (not (pnf_zero? diff))
                               (set-truth (equality des1 des2) 'false 'algebra))))
                      ((or (> (virt-plunk-weight des1) 0)
                           (> (virt-plunk-weight des2) 0))
                       (handle-plunk (designator2
                                       (alg-trans
                                         (des-simplification
                                           (designator2 '(- ,des1 ,des2)))))
                                     (equality des1 des2)))))))

(defun handle-plunk (des c-node)
  (let ((plunk (base-plunk des (virt-plunk-weight des))))
    (if plunk (solve-constraint (translation des)
                                (translation plunk)
                                c-node))))

(defun base-plunk (des weight)
  (if (primitive? des)
      (if (= weight (virt-plunk-weight des))
          des)
      (base-plunk2 (expansion des) weight)))

(defun base-plunk2 (desigs weight)
  (if desigs
      (let ((bp (base-plunk (car desigs) weight)))
        (if bp bp (base-plunk2 (cdr desigs) weight)))))



(defun solve-for (exp)
    (think)
    (let ((des (designator exp)))
      (let ((can (class-name des)))
        (if (and (not (opaque? can)) (= (virt-plunk-weight can) 0))
            (translation can)
            (let ((desigs (equiv-class des)))
              (do ((rest desigs (cdr rest)))
                  ((null rest) '(it does not seem possible to do so))
                  (if (and (opaque? (car rest))
                           (algebraic? (translation (car rest))))
                      (progn (mapc 'plunk! (opaque-vars (car rest)))
                             (think)
                             (return (translation (class-name des))))))))))

(defun opaque-vars (des)
    (fmapcar '(lambda (var)
                (let ((des (designator2 var)))
                  (if (opaque? des) des)))
             (variables (translation des))))
```

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| ALGSYS-BETTER | EXPR | 002 | 002 | BIGGEST-PLUNK-WEIGHT | EXPR | 002 | 029 | [illegible]-VARS | EXPR | 003 | 050 |
| ALGSYS-COIN | EXPR | 003 | 002 | C== | EXPR | 001 | 048 | PLUNK! | EXPR | 002 | 037 |
| ALGSYS-COMP | EXPR | 002 | 005 | CREATE-SIMPLIFICATION | EXPR | 001 | 020 | [illegible]-CONSTRAINT | EXPR | 001 | 047 |
| ALGSYS-INIT | EXPR | 001 | 004 | DES-SIMPLIFICATION | EXPR | 002 | 025 | [illegible]-FOR | EXPR | 003 | 043 |
| BASE-PLUNK | EXPR | 003 | 029 | EQ-MODE | REPEAT | 001 | 000 | [illegible]-PLUNK-WEIGHT | EXPR | 002 | 010 |
| BASE-PLUNK2 | EXPR | 003 | 035 | HANDLE-PLUNK | EXPR | 003 | 023 | | | | |

# Appendix Five
## Utility Procedures

Basic utility functions and macros are described here which are used by the code in appendix two. The LISP definition of most of these utilities is given here. However only a brief english description of some of the basic utilities. Most of the basic concepts behind the utilities described here were developed by various people other than the author and many of them are documented in the LISP MACHINE MANUAL [Wienreb & Moon 78].

### If

`(if a b c)` is equivalent to: `(cond (a b) (t c))`. `(if a b)` is equivalent to: `(cond (a b))`.

### Backquote

The backquote feature provides a form of quote which replaces items preceded by a comma with their value. The following are some examples of the use of backquote:

```
`(foo a ,(+ 1 2))                    evaluates to: (foo a 3)
`(foo ,(list 'a 'b) (list 'a 'b))    evaluates to: (foo (a b) (list 'a 'b))
```

Items in the interior of backquoted expressions which are preceded by `,@` have there values exploded into the top level list structure. An example of the use of this feature is as follows:

```
`(foo ,@(list 'a 'b) ,(list 'a 'b) (list 'a 'b))
        evaluates to:
(foo a b (a b) (list 'a 'b))
```

### Defmacro

This form is used to define macros. A macro definition has a similar syntax to a function definition. When a form whose car is a macro is evaluated the macro definition is used to generate a new form whose value is the value

returned for the original form. The arguments to the macro are bound to the forms in the argument positions rather than their values as is done for functions. An example of a macro definition is given below:

```
(defmacro first-part (x)
      '(caar ,x))
```

Using this definition (first-part a) macro expands to: (caar a) and so (fist-part a) has the same value as: (caar a). A macro is often used instead of a trivial function definition because it is expanded within the compiler and results in more efficient compiled code.

It is sometimes convenient to allow the bound variable list of a macro to be an arbitrary list structure rather than a simple list. In this case atoms in the bound variable list (or bound variable *pattern*, since it need not be a simple list) are bound to corresponding parts of the expression using the macro. For example the new MACLISP form of do could have been defined as a macro along the following lines:

```
(defmacro do (variable-bindings (end-test . end-body) . do-body)
          ...)
```

The bound variable list may also be a single atom, in which case that atom is bound to the entire list of "arguments" to the macro.

## Defmac

defmac is identical to defun with the exception that a macro is created which the compiler can use to open code the function during the compilation of other functions. This is used purely for reasons of efficiency. The open coding is useful in getting the compiler (and other optimization macros such as deftail) to perform optimizations which would not otherwise be done. No function defined via defmac can be recursive however since this would lead to infinite expansion during open coding.

## Let

The let feature allows structured lambda binding. An example follows:

```
(let ((a 1)
      (b 2))
     (+ a b))

is equivalent to:

((lambda (a b) (+ a b)) 1 2)
```

The `let` macro allows the the bindee of a binding pair to be an arbitrary list structure whose parts are bound to the corresponding parts of the value being bound. This is convenient for dealing with functions which conceptually return more than one value.

## Setf

The `setf` macro gives a general method for side effecting data structures. The following equivalences give some examples of its use:

```
(setf a b)                    is equivalent to:    (setq a b)
(setf (get a b) c)                                 (putprop a c b)
(setf (car a) b)                                   (rplaca a b)
(setf (cdr a) b)                                   (rplacd a b)
(setf (cond (a b) (c d)) e)                        (cond (a (setf b e)) (c (setf d e)))
```

## Defsidmac

`defsidmac` is just like `defmacro` except that it is used to define macros which side effect their last argument and treats that argument position specially. Specifically it defines a macro which will embed the side effect in conditionals as does `setfaddf`.

```
(defsidmac addf (x list)
     `(setf ,list (cons ,x ,list)))

(addf x b)                    is equivalent to:    (setf b (cons a b))
```

but

```
(addf x (if a b c))     is equivalent to:     (if a (addf x b) (addf x c))
```

While it may seem obscure to write code which side effects conditional expressions, the ability to do so can be important when macros expand to conditionals. In such situations it is sometimes convenient to be able to side effect applications of these macros.

## Defstruct

The `defstruct` feature is used to define a type of structured object. A `defstruct` definition creates a set of macros. One of these macros is used to create objects of the defined type. The others are used to access the parts of that object. Consider the following example:

```
(defstruct (ship) x-pos y-pos (mass 200))
```

This defines four macros: `make-ship`, `x-pos`, `y-pos`, and `mass`. The `make-ship` macro creates a ship with its mass set to a default value of 200. The following dialogue illustrates a use of these macros:

```
(SETQ HERO (MAKE-SHIP))
(nil nil 200)

(MASS HERO)
200

(SETF (X-POS HERO) 10)
10

(X-POS HERO)
10
```

## Deftype

`deftype` is just like `defstruct` except that it defines another macro

which determines if any given object is of the defined type. In the above example for defstruct if deftype had been used instead then a ship? fmacro would have been created which could determine if any given object was created via make-ship.

### Deftail

This feature allows automatic tail recursion optimization. The way this is invoked is to use deftail instead of defun in a function definition. This feature actually does more than simple tail recursion optimization in that simple accumulations (functions which generate sums, products, or lists recursively) are also converted to iterative forms. The reader need not pay any attention to this feature and may assume that all functions defined via deftail are actually defined via defun in the standard fashion.

### Defarb

defarb is identical to defun except that it allows the bound variable list to be an arbitrary list expression. The atoms in this expression are bound to the corresponding parts of the list of values to which the defined is applied. The most common use of defarb is to have the bound variable ***pattern*** be a single atom in which case that atom is bound to the list of arguments to the function. A function so defined can take an arbitrary number of arguments.

The code for other utility functions and macros follows

```
001
002  (defun namcopy (atom)
003      (maknam (explode atom)))
004
005  (defun name-type (symbol)
006         (putprop symbol 0 'gen-count))
007
008  (defun genname (type)
009         (let ((count (1+ (get type 'gen-count))))
010              (putprop type count 'gen-count)
011              (maknam (append (explode type) (explode count)))))
012
013  (defmacro listp (item)
014            `(not (atom ,item)))
015
016  (defmacro logxor (x y)
017            `(boole 6 ,x ,y))
018
019  (deftail merge (list1 list2)
020         (cond ((null list1) list2)
021               ((member (car list1) list2)
022                (merge (cdr list1) list2))
023               (t (merge (cdr list1)
024                         (cons (car list1) list2)))))
025
026  (deftail integers-between (n1 n2)
027      (if (= n1 n2)
028          (list n2)
029          (cons n1 (integers-between (1+ n1) n2))))
030
```

```
(defmacro setf (form value)
          (let ((form (macroexpand form)))
               (cond ((symbolp form) `(setq ,form ,value))
                     ((numberp form) (break (attempt to set a number)))
                     ((eq (car form) 'cxr)
                      `(rplacx ,(cadr form) ,(caddr form) ,value))
                     ((eq (car form) 'car)
                      `(rplaca ,(cadr form) ,value))
                     ((eq (car form) 'cdr)
                      `(rplacd ,(cadr form) ,value))
                     ((eq (car form) 'get)
                      `(putprop ,(cadr form) ,value ,(caddr form)))
                     ((eq (car form) 'cond)
                      (cond-setf form value))
                     (t (break (unknown operator in setf))))))

;   (setf (cond (a b) (c d)))
;      expand to:
;   (cond (a (setf b e)) (c (setf d e)))

(defun cond-setf (form result)
       (cond-imbed `(lambda (form) `(setf ,form ,',result))
                   form))

(defun cond-imbed (side-effect cond-form)
   `(cond ,@(mapcar '(lambda (clause)
                        (if (or (cddr clause) (null (cdr clause)))
                            (break (illegal cond clause for cond side effect))
                            `(,(car clause)
                              ,(funcall side-effect (cadr clause)))))
                    (cdr cond-form))))

;defsidmac defines cond-imbedding side effect macros like setf

(defmacro defsidmac (effect vars . body)
    (let ((rvars (reverse vars)))
      (let ((ref (car rvars))
            (params (nreverse (cdr rvars))))
        `(defmacro ,effect ,vars
           (let ((ref2 (macroexpand ,ref)))
                (if (and (not (atom ref2))
                         (eq (car ref2) 'cond))
                    (cond-imbed `(lambda (ref) `(,',effect
                                                 ,@',(list ,@params)
                                                 ,ref))
                                ref2)
                    (progn ,@body)))))))

(defsidmac increment (x)
      `(setf ,x (1+ ,x)))

(defsidmac addf (val list-ref)
     `(setf ,list-ref (cons ,val ,list-ref)))

(defsidmac delf (val list-ref)
     `(setf ,list-ref (delete ,val ,list-ref)))

(defsidmac associate (x y assoc-ref)
    (let ((arg1 (namcopy 'x))
          (arg2 (namcopy 'y))
          (alist (namcopy 'alist))
          (as (namcopy 'as)))
      `(let ((,arg1 ,x)
             (,arg2 ,y)
             (,alist ,assoc-ref))
         (let ((,as (assoc ,arg1 ,alist)))
          (if ,as
              (rplacd ,as ,arg2)
              (setf ,assoc-ref (cons (cons ,arg1 ,arg2) ,alist)))))))
```

```
(defmacro hash-array (array-name dim)
      '(array ,(cadr array-name) t ,dim))

(defun hash (item range)
  (remainder (abs (ranhash item)) range))

(deftail ranhash (item)
  (cond ((null item) 0)
        ((symbolp item) (symbol-hash item))
        ((numberp item) (fix item))
        ((designator? item) (desig-hash item))
        ((node? item) (node-hash item))
        ((listp item)
         (logxor (ranhash (car item))
                 (ranhash (cdr item))))
        (t 0)))

(defun node-hash (node)
  (let ((val (node-hash-val node)))
    (cond (val val)
          (t (setq val (random))
             (setf (node-hash-val node) val)
             val))))

(defun desig-hash (desig)
  (let ((val (desig-hash-val desig)))
    (cond (val val)
          (t (setq val (random))
             (setf (desig-hash-val desig) val)
             val))))

(defun symbol-hash (symb)
  (let ((val (get symb 'ranhash)))
    (cond (val val)
          (t (setq val (random))
             (putprop symb val 'ranhash)
             val))))

(defun table-assoc (item table)
       (and (eq (typep table) 'symbol) (setq table (get table 'array)))
       (let* ((index (hash item (cadr (arraydims table))))
              (bucket (arraycall t table index))
              (result (assoc item bucket)))
             (cond (result result)
                   (t
                    (setq result (cons item nil))
                    (store (arraycall t table index) (cons result bucket))
                    result))))
```

```
001
002  (declare (special *query-stack*))
003
004  (defun push-query (query)
005     (w-push query *query-stack*)
006     (view-query (w-top *query-stack*)))
007
008  (defun pop-query ()
009      (w-pop *query-stack*)
010      (view-query (w-top *query-stack*)))
011
012  (defun view-query (query)
013     (cons (car query)
014           (mapcar '(lambda (n ass) (cons n (car ass)))
015                   (integers-between 1 (length (cdr query)))
016                   (cdr query))))
017
018  (defun answer (n)
019        (cdr (nth (1- n) (cdr (w-top *query-stack*)))))
020
021  (defun make-wrap-stack (depth)
022      (let ((last (cons nil (cons nil nil))))
023        (let ((first (mws2 last depth)))
024             (setf (car (cdr first)) last)
025             (setf (cdr (cdr last)) first)
026             first)))
027
028  (defun mws2 (last depth)
029     (if (= depth 1)
030         last
031         (let ((second (mws2 last (1- depth)))
032               (first (cons nil (cons nil nil))))
033              (setf (cdr (cdr first)) second)
034              (setf (car (cdr second)) first)
035              first)))
036
037  (defsidmac w-push (item wstack)
038      `(let ((wstack2 (cadr ,wstack)))
039            (setf (car wstack2) ,item)
040            (setf ,wstack wstack2)))
041
042  (defsidmac w-pop (wstack)
043      `(setf ,wstack (cddr ,wstack)))
044
045  (defmacro w-top (wstack)
046      `(car ,wstack))
047
048  (progn (setq *query-stack* (make-wrap-stack 30)) t) ;progn prevents infinite printing
049
```

```
001
002  ; the following macro is usefull in dealing with partial orders in which three conditional
003  ;  branches exist depending on the relation of the two items.
004
005
006  (defmacro comp (comparison greater-case unrel-case less-case)
007       (let ((comp-var (namcopy 'comp-var)))
008         `(let ((,comp-var ,comparison))
009               (cond ((eq ,comp-var 'greater) ,greater-case)
010                     ((eq ,comp-var 'less) ,less-case)
011                     (t ,unrel-case)))))
012
013  (defmac num-comp (n1 n2)
014          (cond ((> n1 n2) 'greater)
015                ((< n1 n2) 'less)
016                (t 'equal)))
017
018  (defmac alpha-comp (a1 a2)
019          (cond ((alphalessp a1 a2) 'less)
020                ((equal a1 a2) 'unrelated)
021                (t 'greater)))
022
023  (defun scomp (exp1 exp2)
024      (comp (num-comp (ncdrs exp1) (ncdrs exp2))
025            'greater
026            (lexical-comp exp1 exp2)
027            'less))
028
029  (deftail ncdrs (exp)
030         (if (atom exp) 0 (1+ (ncdrs (cdr exp)))))
031
032  (defun lexical-comp (exp1 exp2) ;they must have the same ncdrs
033         (cond ((or (numberp exp1) (numberp exp2))
034                (cond ((not (numberp exp2)) 'less)
035                      ((not (numberp exp1)) 'greater)
036                      (t (num-comp exp1 exp2))))
037               ((atom exp1) ;if one is then they both are
038                (alpha-comp exp1 exp2))
039               (t (comp (scomp (car exp1) (car exp2))
040                        'greater
041                        (lexical-comp (cdr exp1) (cdr exp2))
042                        'less))))
043
044  (defmacro superior-class (pred x y)
045          `(if (funcallm ,pred ,x)
046               (if (funcallm ,pred ,y)
047                   'unrelated
048                   'greater)
049               (if (funcallm ,pred ,y)
050                   'less
051                   'unrelated)))
```

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| ADDF ..... | DEFSIDMAC 002 053 | HASH-ARRAY ..... | DEFMACRO 003 002 | NUM-COMP ..... | DEFMAC .. 005 013 |
| ALPHA-COMP ..... | DEFMAC .. 005 018 | INCREMENT ..... | DEFSIDMAC 002 050 | POP-QUERY ..... | EXPR .... 004 008 |
| ANSWER ..... | EXPR .... 004 018 | INTEGERS-BETWEEN | DEFTAIL 001 026 | PUSH-QUERY ..... | EXPR .... 004 004 |
| ASSOCIATE ..... | DEFSIDMAC 002 059 | LEXICAL-COMP ... | EXPR .... 005 032 | RANHASH ..... | DEFTAIL 003 008 |
| COMP ..... | DEFMACRO 005 006 | LISTP ..... | DEFMACRO 001 013 | SCOMP ..... | EXPR .... 005 023 |
| COND-IMBED ..... | EXPR .... 002 026 | LOGXOR ..... | DEFMACRO 001 016 | SETF ..... | DEFMACRO 002 002 |
| COND-SETF ..... | EXPR .... 002 022 | MAKE-WRAP-STACK | EXPR .... 004 021 | SUPERIOR-CLASS | DEFMACRO 005 044 |
| DEFSIDMAC ..... | DEFMACRO 002 036 | MERGE ..... | DEFTAIL 001 019 | SYMBOL-HASH .... | EXPR .... 003 033 |
| DELF ..... | DEFSIDMAC 002 056 | MWS2 ..... | EXPR .... 004 028 | TABLE-ASSCR .... | EXPR .... 003 040 |
| DESIG-HASH ..... | EXPR .... 003 026 | NAMCOPY ..... | EXPR .... 001 002 | VIEW-QUERY ..... | EXPR .... 004 012 |
| EFFECT ..... | DEFMACRO 002 040 | NAME-TYPE ..... | EXPR .... 001 005 | W-POP ..... | DEFSIDMAC 004 042 |
| GENAME ..... | EXPR .... 001 008 | NCORS ..... | DEFTAIL 005 029 | W-PUSH ..... | DEFSIDMAC 004 037 |
| HASH ..... | EXPR .... 003 005 | NODE-HASH ..... | EXPR .... 003 019 | W-TOP ..... | DEFMACRO 004 045 |

## References

[Borning 77] Borning, Alan.
"Thinglab -- An Object Oriented System for Building Simulations Using Constraints."
Prc. Fifth International Joint Conference on Artificial Intelligence (IJCAI-5). MIT (Cambridge, August 1977), 497-498

[Chang & Lee 73] Chin-Laing Chang, Richard Char-Tung Lee
Symbolic Logic and Mechanical Theorem Proving
Academic Press, New York and London, 1973.

[de Kleer & Sussman 78] Johan de Kleer, Gerald Jay Sussman.
Propogation of Constraints Applied to Circuit Synthesis.
MIT AI Lab Memo 485 (Cambrige, September 1978).

[Doyle 77] Jon Doyle.
Truth Maintenance Systems for Problem Solving.
M.S. thesis (May 1977). Also MIT AI Lab Technical Report 419 (Cambridge, September 1978).

[Doyle 78] Jon Doyle.
A Glimpse of Truth Maintenance.
MIT AI Lab Memo 461a (Cambridge 1978).

[Fay 78] Micheal J. Fay.
"First Order Unification in an Equational Theory"
in Procedings of the Fourth Workshop om Automated Deduction. Austin, Texas, February 1-3, 1979

[Goldstein 73]
Elementary Geometry Theorem Proving
MIT AI Lab Memo 280 (Cambridge 1973).

[Grossman 76] Richard W. Grossman.
Some Data Base Applications of Constraint Expressions.
MIT Laboratory for Computer Science Technical Report 158 (Cambridge February 1976).

[Knuth 69] Donald E. Knuth
The Art of Computer Programing Vol. 2/ Seminumerical Algorithms pp. 360-377 Addison Wesley, 1969.

[Knuth & Bendix 69] Donald E. Knuth, Peter B. Bendix
"Simple Word Problems in Universal Algebras"
Reprinted from Computational Problems in Abstract Algebra, Pergamon Press, Oxford NY, 1969.

[London 78] Philip E. London.
Dependency Networks as a Representation for Modelling General Problem Solvers.
Ph.D. thesis U. Maryland, Dept. of Computer Science Technical Report 698 (College Park, Maryland, September 1978).

[MACSYMA 77] Mathlab Group.
MACSYMA Reference Manual, Version Nine.
MIT Laboratory for Computer Science (Cambridge, December 1977).

[McAllester 78] David A. McAllester.
A Three Valued Truth Maintenance System.
MIT AI Lab Memo 473 (Cambridge, May 1978).

[Robinson 65] J. A. Robinson
"A Machine-Oriented Logic Based on the Resolution Principle."
Journal of the Association for Computing Machinery, vol. 12, pp. 23-41. (Jan. 1965)

[Shrobe 79] Howard E. Shrobe
Dependency Directed Reasoning for Complex Program Understanding
MIT AI Lab Technical Report 405 (Cambridge June 1979).

[Stallman & Sussman 77] Richard M. Stallman, Gerald Jay Sussman.
"Forward Reasoning and Dependency Directed Backtracking in a System for Computer-Aided Circuit Analysis."
Artificial Intelligence 9 (1977), 135-196.

[Steele & Sussman 78] Guy Lewis Steele Jr., Gerald Jay Sussman.
Constraints.
MIT AI Lab Memo 502 (Cambridge, May 1978). Invited featured presentation to appear in Proc. APL79 Conference (Rochester, May 1979).

[Sussman 77] Gerald Jay Sussman.
Slices: At the Boundry between Analysis and Synthesis.
MIT AI Lab Memo 433 (Cambridge, July 1977). Also in IFIP W.G. 5.2. Working Conference on Artificial Intelligence and Pattern Recognition in computer aided design.

[Sutherland 63] Ivan E. Sutherland.
SKETCHPAD: A Man-Machine Graphical Communication System.
MIT Lincoln Laboratory Technical Report 296 (January 1963).

[Zippel 79] Zippel
Probabalistic Algorithms for Sparse Polynomials.
MIT Ph.D. Thesis (September 1979)

*This blank page was inserted to preserve pagination.*

**CS-TR Scanning Project**
**Document Control Form**

Date : 2/29/96

**Report #** AI-TR-550

Each of the following should be identified by a checkmark:
Originating Department:

- [x] Artificial Intellegence Laboratory (AI)
- [ ] Laboratory for Computer Science (LCS)

Document Type:

- [x] Technical Report (TR)
- [ ] Technical Memo (TM)
- [ ] Other:

# Document Information

Number of pages: 115 (122-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:

- [x] Single-sided or
- [ ] Double-sided

Intended to be printed as :

- [ ] Single-sided or
- [x] Double-sided

Print type:

- [ ] Typewriter
- [ ] Offset Press
- [ ] Laser Print
- [ ] InkJet Printer
- [ ] Unknown
- [x] Other:

Check each if included with document:

- [x] DOD Form (2)
- [x] Funding Agent Form
- [x] Cover Page
- [x] Spine
- [ ] Printers Notes
- [ ] Photo negatives
- [ ] Other:

Page Data:

Blank Pages (by page number):

Photographs/Tonal Material (by page number):

Other (note description/page number):

Description : Page Number:

IMAGE MAP: (1-115) UN#'ED TITLE & FUNDING AGENT PAGES, i-iii, 1-66, APPENDIX 2 PAGE 1-12, APPENDIX 2 SYMBOL PAGE 1, 80-82, APP3 PAGE 1-8, APP 3 SYMBOL PAGE 1, 92, APP.4 PAGE 1-3, APP.4 SYM PAGE-1, 97-101, APP. 5 PAGE 1-5, APP-5 SYM PAGE-1, 108-110 (116-122) SCANCONTROL, COVER, SPINE, DOD, TRGT'S (3)

Scanning Agent Signoff:

Date Received: 2/29/96 Date Scanned: 3/4/96 Date Returned: 3/7/96

Scanning Agent Signature: Michael W. Cook

Rev 9/94 DS/LCS Document Control Form cstrform.vsd

UNCLASSIFIED

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

| REPORT DOCUMENTATION PAGE | | READ INSTRUCTIONS BEFORE COMPLETING FORM |
|---|---|---|
| 1. REPORT NUMBER<br>AI-TR - 550 | 2. GOVT ACCESSION NO. | 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE *(and Subtitle)*<br>The Use of Equality In Deduction and Knowledge Representation | | 5. TYPE OF REPORT & PERIOD COVERED<br>Technical Report<br>6. PERFORMING ORG. REPORT NUMBER |
| 7. AUTHOR*(s)*<br>David Allen McAllester | | 8. CONTRACT OR GRANT NUMBER*(s)*<br>N00014-75-C-0643<br>MCS77-04828 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS<br>Artificial Intelligence Laboratory<br>545 Technology Square<br>Cambridge, Massachusetts 02139 | | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS |
| 11. CONTROLLING OFFICE NAME AND ADDRESS<br>Advanced Research Projects Agency<br>1400 Wilson Blvd<br>Arlington, Virginia 22209 | | 12. REPORT DATE<br>January 1980<br>13. NUMBER OF PAGES<br>115 |
| 14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*<br>Office of Naval Research<br>Information Systems<br>Arlington, Virginia 22217 | | 15. SECURITY CLASS. *(of this report)*<br>UNCLASSIFIED<br>15a. DECLASSIFICATION/DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT *(of this Report)*

Distribution of this document is unlimited.

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

18. SUPPLEMENTARY NOTES

None

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

Equality
Truth Maintenance
Artificial Intelligence
Electronic Circuit Analysis
Automated Deduction
Theorem Proving
Constraint Propogation
Problem Solving

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

This report describes a system which maintains canonical expressions for designators under a set of equalities. Substitution is used to maintain all knowledge in terms of these canonical expressions. A partial order on designators, termed the better-name relation, is used in the choice of canonical expressions. It is shown that with an appropriate better-name relation an important engineering reasoning technique, propagation of constraints, can be implemented as a special case of this substitution process. Special purpose algebraic simplification procedures are embedded

such that they interact effectively with the equality system. An electrical circuit analysis system is developed which relies upon constraint propagation and algebraic simplification as primary reasoning techniques. The reasoning is guided by a better-name relation in which referentially transparent terms are preferred to referentially opaque ones. Multiple description of subcircuits are shown to interact strongly with the reasoning mechanisms.

A conceptual distinction is made between propositional deduction and the instantiation of quantified knowledge. A special purpose system is used for the simplified domain of propositional deduction which incorporates many useful features. This system, termed a truth maintenance system of TMS, keeps track of justifications for deductions which are used to generate explanations of deduced beliefs. The TMS also handles the retraction of assumptions when contradictions arise. When assumptions are retracted all deductions which depended on them are retracted in an efficient incremental manner. Assumptions that certain algebraic quantities are non-zero paly an important role in reasoning about algebraic constraints.