Technical Report No. 715

A Multiple-Context Equality-based Reasoning System

George Edward Barton, Jr.

MIT Artificial Intelligence Laboratory

A Multiple-Context Equality-Based Reasoning System

by
George Edward Barton, Jr.
S. B., Computer Science
Massachusetts Institute of Technology
(1979)

**Submitted in Partial Fulfillment
of the Requirements for the
Degree of
Master of Science**

at the
Massachusetts Institute of Technology
June, 1983

Signature of Author George Edward Barton Jr.
Department of Electrical Engineering and Computer Science
April 8, 1983

Certified By Peter Szolovits
Professor Peter Szolovits, Thesis Supervisor

Certified By [illegible]
Richard C. Waters, Thesis Supervisor

Accepted By ______________________
Professor Arthur C. Smith
Chairman, Department Committee

A Multiple-Context Equality-Based Reasoning System

by
George Edward Barton, Jr.

Submitted to the
Department of Electrical Engineering and Computer Science
on April 8, 1983 in partial fulfillment of the requirements
for the Degree of Master of Science

Abstract

Expert systems are too slow. This work attacks that problem by speeding up a useful system component that remembers facts and tracks down simple consequences. The redesigned component can assimilate new facts more quickly because it uses a compact, grammar-based internal representation to deal with whole classes of equivalent expressions at once. It can support faster hypothetical reasoning because it remembers the consequences of several assumption sets at once. The new design is targeted for situations in which many of the stored facts are equalities.

The deductive machinery considered here supplements stored premises with simple new conclusions. The stored premises include permanently asserted facts and temporarily adopted assumptions. The new conclusions are derived by substituting equals for equals and using the properties of the logical connectives AND, OR, and NOT. The deductive system provides supporting premises for its derived conclusions. Reasoning that involves quantifiers is beyond the scope of its limited and automatic operation. The expert system of which the reasoning system is a component is expected to be responsible for overall control of reasoning.

Thesis Supervisors:

Peter Szolovits
Associate Professor of Electrical Engineering
and Computer Science

Richard C. Waters
Principal Research Scientist
in the Artificial Intelligence Laboratory

Acknowledgements

I appreciate the support that was provided by my thesis advisors, Pete Szolovits and Dick Waters, during my investigations of this topic and the larger topic of alerting that I originally proposed for my Master's Thesis research. I would especially like to thank Dick Waters for his persistent but initially unheeded advice to keep my topic small.

I give special thanks to Professors Bob Berwick and Pat Winston for crucial assistance with organization, wording, planning, proofreading, and perseverance at critical times when help was unavailable from other sources. I especially appreciate Professor Berwick's reading of my early thesis drafts.

I thank David McAllester for many discussions and papers about equality reasoning and the Rup system.

I acknowledge and appreciate the aid and patience of the Fannie and John Hertz Foundation, which supported my studies during the period of this research. I would particularly like to thank Dr. Lowell Wood of the Foundation for his support, keen understanding, stimulating discussion, and helpful annual guidance.

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

Table of Contents

1. Introduction

This thesis describes a new design for a useful component of an expert problem-solving system. Like its historical predecessor, the new component remembers facts and tracks down consequences that follow from the facts by propositional and equality reasoning. The expert system in which the deductive component is embedded has responsibility for overall control of reasoning.

The new reasoning system developed from an earlier system in which equality mechanisms were added through demon mechanisms to a simpler underlying system. The design of the new system inverts the design of the earlier system by making equality mechanisms primary, eliminating the old underlying system, and extending the equality mechanisms to cover the functions of the eliminated system.

The new design will function more efficiently than the old in systems that deal with large numbers of equalities. It achieves efficiency by using a grammar-based representation to eliminate redundant consideration of the many forms that a "single fact" can take in the presence of equalities. The new reasoning system also has a context mechanism that can support fast hypothetical reasoning because it remembers the consequences of several assumption sets at once.

The next three sections contain a slightly more detailed preview of the new reasoning system. The last two sections of this chapter then place the new system in its historical and functional context by relating it to previous work and to expert problem-solving systems. Chapter 2 describes the grammar-based mechanisms that implement equality reasoning in the new system, while chapters 3 through 5 elaborate the basic mechanisms to provide other deductive functions. Chapter 6 describes the context mechanism. Chapter 7 proposes a simple implementation method, and Chapter 8 then refines the proposed implementation into the more practical form that was used in the actual implementation. Finally, Chapter 9 presents simple conclusions and some suggestions for further work.

1.1. Reasoning About Equality

When equality assertions are present, a single object or quantity can have many names. This multiplicity of names gives rise to many variant ways of expressing a

```
(G X X)
(G X Y)
(G Y X)
(G Y Y)

(P (F X))
(P (F Y))
(P (F Z))
(OR (P (F X))
    (P (F Y))
    (NOT (= (F X) (F Z))))

(= (H (F X)) 1)
(= (H (F Y)) 1)
(= (H (F Z)) 1)
```

```
(:-> (= (H X) X) (Q (F X)))
(:-> (= (H X) Y) (Q (F X)))
(:-> (= (H Y) X) (Q (F X)))
(:-> (= (H Y) Y) (Q (F X)))
(:-> (= (H X) X) (Q (F Y)))
(:-> (= (H X) Y) (Q (F Y)))
(:-> (= (H Y) X) (Q (F Y)))
(:-> (= (H Y) Y) (Q (F Y)))
(:-> (= (H X) X) (Q (F Z)))
(:-> (= (H X) Y) (Q (F Z)))
(:-> (= (H Y) X) (Q (F Z)))
(:-> (= (H Y) Y) (Q (F Z)))
```

Figure 1. The expressions in each of these four groups become variants of each other when the equalities `(= X Y)` *and* `(= (F Z) (F Y))` *become known. (":->" is used for the logical implication symbol.)*

single piece of information (see Figure 1). It is desirable for a reasoning system that deals with equalities to give consistent results regardless of which variant form is used to express a question.

1.1.1. Making equality mechanisms primary

The system described here starts from a design tradition that achieves this consistency by adding equality mechanisms to a simpler underlying system. (See section 1.4 for historical discussion.) The existing system that was the concrete starting point for the new system was called Rup, which is an acronym for "Reasoning Utility Package" (McAllester, 1982a). In Rup and similar systems, the simple underlying machinery handles logical connectives, records justifications for derived conclusions, and retracts conclusions when supporting assumptions are retracted. The equality system adds new conclusions to enforce consistency under substitution of equals for equals. It enforces consistency by choosing standard names for objects and using them consistently in internal operations.

The new design takes an experimental evolutionary step beyond Rup by inverting system structure. In the new reasoning system, the equality system is made primary. The underlying system that treats logical connectives is eliminated; extensions to the equality system take over its functions. The fundamental position of the standardizing equality mechanism allows non-standard variant forms of facts to be expunged from the internal database. Compensating adjustments ensure that the new system still derives secondary consequences that would have been derived while considering the variants that were removed.

1.1.2. Eliminating variant forms

This work began as an attempt to make Rup faster in an application that involves large numbers of equality assertions. It is expensive for a system to repeatedly give individual consideration to the many forms that a "single fact" can take in the presence of many equalities. New equalities can decrease the number of independent facts and multiply the variant forms of each fact by changing previously independent forms into variants of one another.

In the new system that standardizes variant forms, new equalities make future work easier because facts are eliminated from future consideration when they become superflous variants of others. In the old system that retains variant forms, new equalities make work harder because they multiply the number of separately maintained variant forms for each fact. As Figure 2 illustrates, the difference between these situations can be large in a knowledge base with many equality assertions and therefore many ways to express the same fact.

1.1.3. Grammars for equivalence classes

The new reasoning system deals with equalities by distilling stored facts into a grammar that uses standard forms systematically to describe sets of equivalent expressions. It operates directly on the grammar, thus effectively operating on whole classes of equivalent expressions at once. This technique is usually faster than treating each expression in a class individually.

The technique of dealing with possible variants through schematic representation instead of direct consideration amounts to dealing with variation intensionally instead of extensionally. It also alters the balance of computational effort between retrieving information and computing the consequences of new assumptions. It is easier to retrieve information when variants are maintained directly because the information stored with each variant is always correct. It is more difficult to compute consequences because many variants can require readjustment.

A. INITIAL ASSERTIONS

```
(= A1 A)          (= A2 A)          (= A3 A)          (= A4 A)
(= B1 B)          (= B2 B)          (= B3 B)          (= B4 B)
(= F G)

(= (F A1) X11)    (= (F A2) X12)    (= (F A3) X13)    (= (F A4) X14)
(= (G A1) X21)    (= (G A2) X22)    (= (G A3) X23)    (= (G A4) X24)

(= (F B1) X31)    (= (F B2) X32)    (= (F B3) X33)    (= (F B4) X34)
(= (G B1) X41)    (= (G B2) X42)    (= (G B3) X43)    (= (G B4) X44)
```

B. ADDITIONAL ASSERTION

```
(= A B)
```

C. OLD READJUSTMENTS

Congruence checks:

```
(G B)             (G B2)
(G B4)            (F B2)
(F B4)            (G B1)
(G B3)            (F B1)
(F B3)
```

Equality invariant checks:

```
A = B             (G A3) = X23
A = B             (G A2) = X22
B = B4            (G A2) = X22
B = B3            (G A1) = X21
B = B2            (G A1) = X21
B1 = B            (F A4) = X14
B = B4            (F A4) = X14
B = B3            (F A3) = X13
B = B2            (F A3) = X13
B1 = B            (F A2) = X12
(G A4) = X24      (F A2) = X12
(G A4) = X24      (F A1) = X11
(G A3) = X23      (F A1) = X11
```

D. NEW READJUSTMENTS

```
(G A4)
```

Figure 2. Adding new information can take significantly fewer operations in a system that eliminates variant forms of facts from its consideration. (A) shows the facts that were initially presented to the new system described here and to its historical predecessor. Note that only a few distinct objects are mentioned and hence the number of variant forms is large. (B) shows the single additional fact that was then added to each system. (C) lists the items that were subjected to readjustments technically known as congruence checks and equality invariant checks when the new fact was added to the old system. (D) shows the single item that was considered for the corresponding readjustment, described later as a right-hand forwarding adjustment, in the new system. (Each system also made other internal adjustments.)

The situation is reversed when variation is represented intensionally. It is harder to retrieve information because the standard version of a question must be computed before the answer will be accessible. It is easier to compute new consequences because it is easier to readjust a small schematic representation than to readjust numerous class members individually. The new system is designed to compute new consequences quickly; this choice is appropriate for situations where only a small fraction of a large knowledge base is examined between additions of information.

1.2. Using Multiple Contexts

Expert systems do hypothetical reasoning when they make temporary assumptions and investigate their consequences. When combined with permanently asserted facts, such assumptions can yield large numbers of derived conclusions. The set of consequences of a set of assumptions may be expensive to compute. A reasoning system that can remember the consequences of only one hypothesis at a time is forced to repeat that computation each time a given hypothesis or its equivalent is reconsidered. It must also spend time retracting the consequences of old assumptions whenever a new set of assumptions is to be considered. Such deductive thrashing can cripple an expert system that switches frequently among sets of assumptions.

The reasoning system described here does less deductive thrashing in some situations. Its mechanisms for hypothetical reasoning are intended particularly for the situation in which an expert system switches repeatedly among a small number of different assumption sets. It provides fast assumption switching in such cases because it remembers the consequences of several assumption sets at once. The consequences of a set of assumptions can be investigated simply by directing attention to the right set of consequences. If those consequences have already been computed and saved, it is not necessary to recompute them.

The mechanisms that provide hypothetical reasoning also restrict the kinds of hypothetical reasoning for which the system will be useful. The design of the system relies on a distinction between permanently asserted facts and temporary assumptions. The design does not allow permanently asserted facts to be retracted or assumed to be false. The method of associating sets of assumptions with sets of consequences assumes that the number of possible temporary assumptions is relatively small. That association method will also perform poorly if the number of different assumption sets is too large. (Exact recommended limits have not been investigated.)

The possibility of multiple assumption sets results from the use of a separate data structure called a context for each assumption set. Contexts can be retained to avoid rederivation of assumption consequences because each context encodes the consequences of its own associated assumptions and no others. The rederivation of consequences can be further reduced by using the same context for all variants of the assumption set. Sharing contexts in this way cannot yield incorrect consequences because substitution of a variant does not occur unless the variant assumption can be proved permanently equivalent to the original assumption. An equivalence is considered permanent if its proof uses only permanently asserted facts.

Although the multiple-context design provides fast switching between contexts, it also has associated costs. A system with a large number of active contexts requires much memory in which to store them. It also becomes quite slow at adding new permanent assertions because each new permanent assertion must be added to every context that exists when the assertion is made. For this reason the current implementation of the system will destroy a context if it discovers that it has spent more time adding new permanent assertions to the context than it spent creating it. At present it has no more refined way of identifying valuable contexts.

1.3. Implementation of the Reasoning System

The new reasoning system has been implemented in Lisp Machine Lisp* (Weinreb, Moon, and Stallman, 1983). The implemented system is called XRup because it was developed through a series of experimental changes to Rup (McAllester, 1982a). Chapters 7 and 8 will describe the implementation. See the appendix for a brief discussion of the performance of the implemented system in one application.

*It runs with Lisp Machine System Version 93.

1.4. Relation to Previous Work

The XRup system represents an experimental evolutionary step in a series of problem-solving systems that have been developed at M.I.T. over a period of several years. This section places XRup in its historical context by describing those related systems.

1.4.1. Using justifications for dependency-directed backtracking

XRup arises from a tradition of problem-solving systems that record justifications for their conclusions and use those justifications to help control their actions. Such systems were developed at M.I.T. as a result of experience with the automatic *chronological backtracking* performed by earlier problem-solving systems such as Micro-Planner (Sussman, Winograd, and Charniak, 1971).

It is often necessary for a problem-solving system to make assumptions and investigate their consequences. The system must *backtrack* and choose new assumptions whenever the assumptions in force are discovered to be untenable. With chronological backtracking, the system chooses a new set of assumptions by making a new decision at its most recent choice point. When a difficulty is discovered, however, the most recently chosen assumption may not be one of the assumptions that underlie the difficulty. In that case, chronological backtracking will cause the system to cycle through many alternative possibilities for recently considered but irrelevant questions before it finally tries alternative answers for the questions that are relevant to the difficulty.

Stallman and Sussman (1977) used *dependency-directed backtracking* instead of chronological backtracking in the design of the EL system for electronic circuit analysis. A system with dependency-directed backtracking records justifications for its deductions. When a set of assumptions leads to contradiction, the system uses the recorded justifications to trace back from the contradiction to the assumptions that were used in the reasoning that uncovered the contradiction. The system then attempts to resolve the inconsistency by making a new decision at one of the points where the relevant assumptions were chosen. It does not waste time considering alternative answers to irrelevant questions, since only those assumptions that participate in the contradiction are considered for retraction.

The EL system also uses the recorded justifications after it has decided which assumptions to modify. When an old assumption is retracted, the system must also retract the conclusions that were reached on the basis of the old assumption. However, it does not need to repeat its entire analysis, since it can use recorded justifications to identify and preserve conclusions that did not depend on the old assumption.

A problem-solving system that uses recorded justifications to help control its actions through dependency-directed backtracking can have a large advantage over a system that does chronological backtracking. The advantage can be exponential in the number of irrelevant choice points. As Stallman and Sussman point out, if EL derives a contradiction while analyzing a circuit, and if the circuit contains ten transistors that are not mentioned in the assumptions that underlie the contradiction, dependency-directed backtracking can eliminate $3^{10} = 59049$ combinations of assumptions that chronological backtracking might consider.

1.4.2. Truth maintenance systems for problem solving

The EL system can cease to believe a previously derived conclusion when the assumptions that underlie that conclusion are retracted. However, the old conclusion should come back into favor if later readjustments cause the relevant old assumptions to be adopted again. EL uses dependency information to save the effort of rederiving implications from scratch in such cases. At any time, certain facts are considered to be *in*, or believed, while other facts are considered to be *out*, or not believed. The dependency information linking a group of facts does not change as the facts go *in* and *out*.

Because EL does not forget facts and dependencies even when the facts are not believed, it does not need to repeat previous analytical work when facts once believed but later invalidated are validated once more. Instead of repeating an old analysis, it simply causes the conclusions of the old analysis to come back *in* when all of their underlying justifications come to be *in* once more.

Doyle (1977) generalized, improved, and gave explicit treatment to the *in/out* mechanism by describing the implementation of a *truth maintenance system* (TMS).

Doyle's TMS accepts as input justifications for beliefs in components of a program's knowledge. Like part of the EL system, it functions mainly to keep the *in/out* states of beliefs consistent with the support relationships that the justifications express. Unlike the EL system, however, Doyle's TMS may contain *non-monotonic dependencies* that allow belief in facts to be based in part on the *lack* of belief in other facts. Doyle's non-monotonic dependencies can model earlier non-monotonic primitives like the THNOT primitive of Micro-Planner, but have the advantage that the nature of the assumption is made explicit and available in future deductions.

Doyle characterizes *assumptions* as beliefs that are justified with non-monotonic dependencies. For example, suppose a system chooses to assume *P* because it has no evidence to the contrary and needs to make an assumption to continue its analysis. The system can indicate to the TMS that its belief in *P* is based in part on the fact that it has no reason to believe (NOT *P*) (that is, that (NOT *P*) is *out*). The TMS will then cause assumption *P* and its consequences to be automatically retracted if (NOT *P*) later comes to be believed. If such an arrangement is desired, it is possible to add other justifications that will cause the system to then automatically choose an alternative to assumption *P*. In this way, backtracking instructions can be encoded directly in the TMS.

Doyle shows how the TMS can be used to encode dynamically extendable, unordered sets of possible alternative assumptions, in addition to several other assumption structures. However, the TMS formalism does not help an expert system decide what form of assumption control it should use the TMS to implement:

> Truth maintenance systems do not directly address some of the problems of hypothetical reasoning. There is a large body of research on knowledge-based reasoning concerned with the proposal of hypotheses and differential diagnosis between them. These issues are beyond the immediate capabilities of truth maintenance systems because they require knowledge of the semantics of facts, and such knowledge is not available to the domain-independent methods described here. *(Doyle (1977), p. 31)*

The Rup and XRup systems share this characteristic.

As Stallman and Sussman (in effect) note, the operations that a TMS performs when assumptions are added and retracted can be viewed as context-switching between contexts that are associated with subsets of the set of possible assumptions. Doyle points out that many of the difficulties that are encountered with context mechanisms in systems like Micro-Planner result from the attempt to approximate dependency relationships by means of contexts. The deficiencies of chronological backtracking are one example. Shrobe (1978) gives another example that involves hypothetical reasoning. In Micro-Planner, a problem solver would investigate the consequences of an assumption A by creating a new context in which A was added to the assumptions of the old context. If conclusion D was then derived in the new context, the conditionalized conclusion (:-> A D) would be added to the old context. In cases where the derivation of D in the new context did not actually depend on A, it would be better to add D itself to the old context. Without a record of logical dependencies, however, it is not possible to identify those cases.

1.4.3. The TMS as an instrument of examination

McAllester (1981a) argues against the use of non-monotonic dependencies to model assumptions. In McAllester's view, the role of the TMS should be more narrowly conceived. The TMS should implement traditional propositional logic, maintain justifications for derived conclusions, incrementally update beliefs when premises are added and removed, and support dependency-directed backtracking by using justifications to track down the premises that underlie contradictions. It should not implement non-monotonic logic (McDermott and Doyle, 1978) or perform backtracking and assumption control as in Doyle (1977).

McAllester's first objection to the use of non-monotonic mechanisms in the TMS is that they can lead to computational and semantic problems:

> The first domain independent system which performed all of the basic truth maintenance functions was developed by Jon Doyle. Doyle's system used "non-monotonic" dependencies which justify a node's being *in* by the fact that some other node is *out.* Such dependencies are typically used to make assumptions. For example one might assume A by justifying A with the fact that (NOT A) is *out.* Thus if (NOT A) ever becomes *in*, the justification for A will no longer be valid and A will become *out.* This leads to problems however if the system is able to prove (NOT A) from the assumption of A. First (NOT A) comes *in* forcing A *out.* But because (NOT A) depends on A this in turn causes (NOT A) to become *out,* which, via the

> non-monotonic dependency, leads to A becoming *in*, which leads to (NOT A) becoming *in*, ad infinitum. While there may be ways to fix this problem, it seems hard to motivate the introduction of non-monotonic mechanisms which lead to unnecessary complications. *(McAllester (1981a), p. 19)*
>
> Another problem with non-monotonic systems is their obscure semantics. Attempts to formalize "non-monotonic logics" are plagued by "unsatisfiable" situations similar to the infinite computation described above. *(McAllester (1981a), p. 19)*

He goes on to observe that the usual use of non-monotonic justifications amounts to programming backtracking control into the dependency network rather than recording logical relationships:

> While it may be possible to debug [the above] problems, the fundamental motivation behind non-monotonic justifications is suspect. Certainly one cannot argue that an assumption is made *because* one cannot prove its negation. At any time there is an infinite number of assertions which the system cannot prove to be false, but one would certainly not want to assume all these things. Therefore a non-monotonic justification does not capture the true reason for making an assumption. It might capture what the system should do if it could prove the negation of an assumption, but this is a backtracking issue and should not be represented as a justification. *(McAllester (1981a), p. 19)*

In McAllester's view, such processes as backtracking and assumption control are not properly the province of the TMS. The TMS functions as an "instrument of examination" that allows one to view the simple consequences of premise sets. It performs well-understood propositional deduction and does bookkeeping tasks that are associated with support relationships. The larger system in which the TMS is embedded has responsibility for assumption control, backtracking, the use of quantified knowledge, and the overall control of reasoning.

This separation of functions refines and clarifies the role of the TMS and makes it fast enough to be used as an automatic instrument. It also makes the problem of assumption control explicit instead of confusing it with other matters and burying it in the structure of the dependency network:

> The problem with non-monotonic logics is that they bring in non-traditional formalisms too early, muddying deduction, justifications, and backtracking. The aspect of truth maintenance which cannot be formalized in a traditional framework is premise control, which has only just begun to be explored. *(McAllester (1981a), p. 20)*

> The TMS can be used to perform all propositional deduction in a general deduction framework. The primary aspect of general deduction which cannot be performed by a TMS is the instantiation of quantified formulae and axiom schemas. The position is taken here that those problems which are of a purely propositional nature can be solved to such a degree that the only difficult issues remaining in automated deduction involve the control of instantiation. *(McAllester (1981a), p. 1)*

> It is well known that automated deduction and theorem proving systems are subject to explosive computations. However the TMS described ... seems free of this problem. While it is possible to make the [larger system] do a great deal of backtracking ..., in practice this is not important because the number of assumptions is usually small and they do not interact in complex manners. The difference between the TMS and more general deductive systems is that the TMS deals with propositional logic only. All of the difficult problems in automated deduction involve instantiation of quantified formulae and axiom schemas. *(McAllester (1981a), p. 14)*

> From the point of view taken here *instantiation and deduction are separate processes.* Deduction is the process of assigning truth values to assertions based on other truth values already in the system. This can be done entirely by the TMS. Instantiation can be thought of as generating propositional formulae upon which the TMS can operate. This outlook on deduction leads to novel control strategies. *(McAllester (1981a), p. 14)*

McAllester's TMS performs propositional deductions by a technique that McAllester calls *propositional constraint propagation* and traces back to ultimate roots in an algorithm by Davis and Putnam (1960). Formulae are linked into a network of logical constraints when they are first encountered. (See section 4.3.1 for an example.) A deduction is made whenever a new truth value follows from given or previously determined values and a *single* constraint. Constraint propagation terminates when the constraint network is *relaxed* (no further deductions can be made from single constraints). The network is relaxed incrementally when new premises and constraints are added.

1.4.4. Rup and XRup

McAllester (1982a) describes Rup, which is an implemented reasoning utility package that is based on the kind of TMS that McAllester (1981a) describes. In addition to a TMS that reasons about the propositional connectives AND, OR, and NOT, Rup contains mechanisms for reasoning about equality, simplifying expressions under an external simplicity order, and controlling reasoning with demons.

XRup was designed in an attempt to speed up the equality reasoning that Rup performed. Section 1.1 has explained how the system structure of XRup differs from that of Rup. In brief, XRup results from eliminating the TMS, making a new set of equality mechanisms primary, extending the equality mechanisms to take over the old TMS functions, and adding multiple contexts.* (Note that the experimental XRup system is an *extreme* system in the sense that it starts with a single mechanism, a grammar-based mechanism for reasoning about equality, and attempts to stretch that mechanism as far as possible toward forming the basis of a reasoning system with wider capabilities.)

Later chapters will describe XRup in detail, and references to the pieces of previous work that played a direct role in the design of XRup will be provided when the relevant design features are discussed. Of particular note are the Rup system, McAllester's (1982b) expression grammars, the congruence-closure algorithms of Downey, Sethi, and Tarjan (1980), and McAllester's (1981b) application of those algorithms.

Although many previous reasoning systems have been capable of equality reasoning (see McAllester (1980) for some discussion), XRup appears to be the first implemented system that uses a grammar-based representation to eliminate redundant consideration of equivalent variants. Shrobe (1978) describes a somewhat similar mechanism:

> [Shrobe's REASON system] uses a rather unusual tactic [to handle equality]. The standard tactic is to build up equivalence classes of equal objects. This, however, imposes a price when searching for a match since one must check for variants of the desired assertion using any possible representative of the equivalence class. REASON instead eliminates this possibility by doing the work in advance; it makes one of the objects "disappear." *(Shrobe (1978), p. 91)*

Shrobe's mechanism is called *identification* and is implemented with "utility marks." An assertion with its utility mark set will not be returned in a normal match and will not trigger rules. Although Shrobe does not discuss the mechanism in great detail, it appears to be less general than the mechanisms of XRup. It is apparently used only

*The current implementation of XRup also differs from Rup in the omission of certain capabilities, including the use of demons and the simplification of expressions; see Chapter 9.

in conjunction with *anonymous objects* that are created when the system does not know the "true name" of an object. In addition, Shrobe's identification mechanism does not completely eliminate variant forms from internal readjustments. The TMS continues to separately adjust the truth values of variant forms:

> *In* and *out* deal with belief (or logical relationships) while the utility mark is strictly an issue of control (of heuristic value) .... A fact whose utility mark is set may support belief of other facts although its presence will otherwise be ignored. *(Shrobe (1978), p. 93)*

It seems probable that a system cannot completely eliminate redundant consideration of equivalent variants without making equality mechanisms fundamental as XRup does. Only by taking this step is it possible to use a specialized representation to operate on whole classes of equivalent expressions at once.

1.5. Application to Expert Systems

Stefik *et al.* (1982) define expert systems as problem-solving systems that solve substantial problems that are generally thought to be difficult and require expertise. EL, described in section 1.4.1, is one example of such a system. Stefik also cites examples of systems that do such tasks as the interpretation of mass spectrometer data, the diagnosis of infectious diseases, experiment planning in molecular genetics, errand planning, and speech understanding.

1.5.1. The difficulties of expert tasks

A problem-solving system that solves a limited and well-understood class of problems can have a simple design and may require only simple reasoning capabilities. However, as Stefik notes, the tasks performed by most expert systems have characteristics that make them more difficult.

For example, in many tasks, input data values may be missing, erroneous, or extraneous. Systems that perform such tasks must make assumptions in the presence of partial information, and they must form hypotheses about what evidence is believable in the presence of contradictory information. They may need to use long chains of hypothetical reasoning in forming and investigating their hypotheses, and they should be able to identify the assumptions on which their final analyses depend.

In many other tasks, prediction or planning is required in addition to the interpretation of data. Prediction requires reasoning about time, and a predictor must have a model of the way various events change the modeled situation over time. Planning problems are often sufficiently large that a planner does not immediately understand all of the consequences of proposed actions, and consequently a planner must be able to act tentatively and adjust plans when problems are discovered. In order to adjust defective plans, the planner must be able to trace back from the identification of a defect to the planning decisions that underlie it. Expert systems may also require other complex capabilities.

1.5.2. Using justifications and equality reasoning

The reasoning capabilities that Rup and XRup provide can be useful in the construction of expert systems because they help support attacks on the kinds of problems listed above. For example, Stefik points out that justifications for derived conclusions are essential in a system that must revise its beliefs in the face of new evidence:

> [The ideas in EL] were the intellectual precursors to work on belief revision systems [Doyle and London, 1980]. Belief revision can be used for reasoning with assumptions or defaults. For a problem solver to revise its beliefs in response to new knowledge, it must reason about dependencies among its current set of beliefs. New beliefs can be the consequences of new information received or derived .... An important question is "what mechanisms should be used to resolve ambiguities when there are several possible revisions?" It is clear this choice needs to be controlled, but the details for making the decision remain to be worked out .... Every approach depends critically on the kinds of dependency records that are created and saved. *(Stefik et al. (1982), p. 162)*

Reasoning about equality, on the other hand, is especially important in systems that plan or understand changes in modeled objects. If attribute $F(A)$ of object A changes or is modified in the event or action δ, it is important to know whether there are other expressions that refer to the same object as A. For example, if $A = B$, then the expression $F(B)$ will change value just as $F(A)$ does when δ occurs. In contrast, if it can be proven that C is distinct from every object that is modified in δ, then it follows that expressions like $F(C)$ must have the same value before and after δ.* Shrobe's (1978) system for reasoning about programs is an example of a system that must reason about equality for such reasons, during its analysis of side effects. Also, McAllester (1980) interprets another important reasoning technique, propagation of constraints, as a special case of equality reasoning.

It is likely that expert systems will in the future become more dependent on abilities like justifying conclusions and understanding equality. Davis (1982) observes that expert systems of the current generation are based largely on *empirical associations* rather than detailed understanding. Empirical associations justify correlations between different features of a situation on the basis of statistical properties of past experience ("previously it was observed that whenever A and B held, C was also true"). In contrast, with levels of understanding that go deeper than empirical associations, it is often possible to justify correlations on the basis of structure and function ("whenever A and B hold, C will also be true because of the way the object under consideration is constructed").

As Davis notes, empirical associations cannot by themselves support explanation, learning, and many other tasks that human experts can perform. True expertise can solve the problems that systems with a lower level of understanding can solve, but it can also explain results, learn, restructure knowledge, break rules when appropriate, determine the relevance of knowledge, and degrade gracefully. In order to construct expert systems that approach expertise in these ways, it will be necessary to make supporting reasoning more complex and subtle.

1.5.3. Making expert systems faster

XRup is designed for use in expert systems that can be divided, as section 1.4.3 describes, into two parts. One component of such a system functions as an instrument for examining the simple consequences of sets of premises. The larger system in which the simple deductive component is embedded has responsibility for the use of that instrument and the overall control of reasoning.

There are several possible ways to make such an expert system run faster. One domain-dependent approach, which often makes a great deal of difference when it

*The situation is more complicated for expressions that denote such attributes as the age of an object, which can change even when the object itself has not been altered.

can actually be carried out, is to shorten the overall course of reasoning by gaining a better understanding of the target problem and using this understanding to devise a better algorithm. Winston (1977) mentions the improvements that this approach can yield:

> It is generally true that more description yields better, faster performance. And more precisely, it is generally true that concentrating on the description of the atomic entities in a domain brings out constraints that substantially help analysis. *(Winston (1977), p. 71)*

The development of XRup represents a different, domain-independent approach, which can be carried out in conjunction with such efforts. The overall system will operate faster if it is provided with a faster simple-deduction component. It is interesting to note that within its focus on the domain-independent deductive component of the system, this approach has an affinity to the previously mentioned approach of finding good domain-dependent algorithms. Just as the previous approach tailors the structure and operation of the overall expert system to the characteristics of its particular domain task, the design of XRup tailors the structure and operation of the deductive component to the characteristics of its major task, which is reasoning about equality.

1.5.4. Incomplete equality reasoning

As section 1.1.2 explains, the operation of a system that does equality reasoning can be expensive in an application that involves many equalities. (See the appendix for a suggestive example.) Faced with such a situation, many system designers gain speed at the expense of deductive power by truncating the process of equality reasoning.

For example, consider an application built on a system that has equality links between nodes of a concept network like that described by Hawkinson (1980). The designer of such a system might choose to stop following equality links after three links have been traversed. This decision would probably make the system fragile and difficult to modify, since it would make system behavior dependent on details of network structure. The insertion of a new concept into a chain of equality links could cause the system to miss deductions that it previously was able to make.

From the point of view advocated here, it would be better to use a system that performs exhaustive equality reasoning but uses an efficient algorithm that does not require search. If the design of the XRup system proves to be practical, it will be a valuable contribution to the construction of expert problem-solving systems.

2. Basic Grammar-Based Equality Mechanisms

The heart of the XRup system is an internal representation that distills facts and assumptions* into a compact form. The representation is a kind of grammar that McAllester (1982b) has used to represent the consequences of equalities. Grammars were chosen as the foundation of XRup because a grammar can provide a small representation of a large set of expressions. A representation with that capability is essential if the system is to deal collectively rather than individually with equivalent variants of facts.

In XRup, all statements are treated internally as equalities. The equalities are represented in a way that makes it easy to recognize the consequences that follow by substitution of equals for equals. The first section of this chapter explains how external statements are translated into equalities. The second section tells how McAllester represents equalities and their consequences with grammars. The final sections illustrate how grammars can be efficiently constructed from sets of premises. Later chapters will tell how to extend the basic grammar-based mechanisms so that XRup can provide supporting premises for derived conclusions, reason about constants, truth values, and logical connectives, and consider temporary assumptions.

2.1. Representing Facts in XRup

XRup is an equality-based system. Its fundamental operations deal with equalities between expressions. They do not deal directly with other kinds of statements or with truth values. Equality mechanisms were made primary so that the system could treat equivalent expressions in an efficient, streamlined way. The grammar-based representation describes classes of equivalent expressions compactly.

Not every statement is an equality. Fortunately, though, a trivial convention allows any statement to be treated as one. A statement can be regarded as an expression whose value is either `TRUE` or `FALSE`. The assertion `(HAS-COLOR CLYDE GRAY)` is easily translated into the equality assertion `(= (HAS-COLOR CLYDE GRAY) TRUE)`. If the

*The term "fact" will generally be used for a statement that has been asserted and is not subject to possible retraction. In contrast, the term "assumption" will be used for a statement that has been asserted but *is* subject to possible retraction. When the term "premise" is used without qualification, it can refer to either kind of statement.

additional fact (= CLYDE ELEPHANT-32) is known, the mechanisms that reason about equivalent expressions can derive the consequence (HAS-COLOR ELEPHANT-32 GRAY) without needing special alteration.

The equality mechanisms must be extended, however, if the deductive system is to make much sense of TRUE, FALSE, and related symbols. The bare mechanisms cannot even derive the fact that (= X X) equals TRUE. Later chapters will tell how to give the equality system special knowledge about TRUE, FALSE, AND, OR, NOT, and the explicit symbol "=". This chapter describes the unextended equality mechanisms.

2.2. McAllester's Expression Grammars

The deductive machinery of XRup is built on the *expression grammars* that McAllester uses to encode equalities and their consequences. Although McAllester did not implement a reasoning system based on expression grammars, the representation efficiently supports reasoning that involves the substitution of equals for equals. This section describes that representation and the associated retrieval algorithms.

McAllester's expression grammars are context-free grammars of restricted form. They are subject to two restrictions. First, each rule in such a grammar must have one of the following forms:

- X ==> a, where X is a nonterminal symbol and a is a terminal symbol.
- X ==> $(Y_1 \ldots Y_n)$, where X is a nonterminal symbol and each Y_i is a nonterminal symbol.

Second, the grammar cannot have two different rules with the same right-hand side. It will be freely assumed from this point forward that these restrictions on grammars are in effect.*

2.2.1. Generating classes of equivalent expressions

A set of equalities divides the set of all possible expressions into classes of equivalent expressions. Two expressions are placed in the same equivalence class if

*These restrictions are similar to those defining McNaughton's (1967) *backwards-deterministic parenthesis grammars*. McAllester's definition is slightly more restrictive. McNaughton showed that the equivalence and inclusion problems for parenthesis grammars are solvable. See McAllester (1982b) for properties of McAllester's grammars.

the equalities imply that the two expressions must be equal. For example, the single equality `(= x (f x))` produces the following equivalence classes:

```
{ x, (f x), (f (f x)), ... }
{ (g x), (g (f x)), (g (f (f x))), ... }
...
{ y }
{ z }
...
{ (f y) }
{ (f z) }
...
```

When a grammar is used to represent the consequences of a set of equalities, each nonterminal in the grammar generates a different equivalence class. (For this reason, nonterminals will often be called class symbols.) If the same nonterminal generates two different expressions, then the expressions must be equal as a consequence of the equalities. Consider the following set of premises:

```
(= x (f (f (f x))))
(= y (f x))
(= z (f y))
```

The following grammar represents the consequences of those equalities:

```
X ==> x | (F Z)
Y ==> y | (F X)
Z ==> z | (F Y)
F ==> f
```

(The symbol "|" is used to abbreviate sets of rules with the same left-hand side. Capital letters are used here for nonterminals.) This proof shows the expressions `(f (f z))` and `(f x)` to be equal:

```
  (f x)
= (f (f (f (f x))))               ; because x = (f (f (f x)))
= (f (f (f y)))                   ; because y = (f x)
= (f (f z))                       ; because z = (f y).
```

In the grammar, the nonterminal `Y` generates both expressions:

```
   Y                 Y
  / \               / \
(F    X)          (F    X)
 |    |            |    |\
(f    x)           |    | \
                   |   (F   Z)
                   |    |   |
                  (f   (f   z))
```

Consequently, the expressions are correctly equated by the grammar.

2.2.2. *Extending generation to expression forms*

It is also a consequence of the above premises that `(g (f x))` equals `(g (f (f z)))` even though `g` does not appear in the grammar. McAllester captures such facts by extending the notion of generation beyond generation by nonterminals. An *expression form* is like an expression but may be ultimately built from both terminal and nonterminal symbols. (The terminal symbols need not appear in the grammar.) Expressions are generated from expression forms by using grammar rules to expand embedded nonterminals in the ordinary manner. The expression form `(g Y)` generates both `(g (f x))` and `(g (f (f z)))`:

```
(g    Y)               (g    Y)
 |   / \                |   / \
 | (F    X)             | (F    X)
 |  |    |              |  |    |\
(g (f    x))            |  |    | \
                        |  |   (F   Z)
                        |  |    |   |
                       (g (f   (f   z)))
```

Consequently, the grammar captures the equality between them when generation is extended to expression forms.

2.2.3. *Recovering generators in linear time*

A grammar equates two expressions if it generates them from the same expression form. Fortunately, this situation is easy to recognize. The following variant of a recursive algorithm by McAllester will work:

- To determine whether two expressions u_1 and u_2 are *equal according to the grammar*, compute the generators of u_1 and u_2 and see whether they are the same.

- If u is an expression then its *generator* $G(u)$ is defined in terms of its subexpression generator $S(u)$. If a rule X `==>` $S(u)$ appears in the grammar then $G(u) = X$; otherwise $G(u) =$ `(:EXPRESSION` $S(u)$`)`. (The symbol `:EXPRESSION` is a special marker that allows programs to easily identify generators that are not nonterminals.)
- If u is an expression then its *subexpression generator* $S(u)$ is defined as follows. If u is atomic then $S(u) = u$. Otherwise u must be of the form `(`$u_1 \ldots u_n$`)`, and $S(u) =$ `(`$G(u_1) \ldots G(u_n)$`)`.

(See Figure 3 for an example.) The algorithm runs in linear average time, according to McAllester's analysis.

The generator algorithm recursively transforms its input expression into a standard expression form that is used for that expression and all equivalent variants. The standard expression form that is recovered is what McAllester calls the *maximal generator.* The maximal generator of an expression is an expression form that generates the expression, but is not itself generated by another expression form. `(g Y)`, `(g (F X))`, and `(g (f (f z)))` are all expression forms that generate `(g (f (f z)))`, but `(g Y)` is the maximal generator. Only the maximal generator determines the correct equality consequences. If other generators are used, some equality consequences may be missed.

2.3. Building an Expression Grammar

Section 2.2 shows how to read grammars but not how to construct them. This section tells how to build a grammar that encodes the consequences of a set of equalities. The grammar is built from an empty grammar by assimilating premises sequentially.

2.3.1. Steps toward incremental grammar construction

McAllester ties grammar construction to a related problem that Downey, Sethi, and Tarjan (1980) have studied. The Downey, Sethi, and Tarjan algorithm can be used to construct a standardizing function that maps equivalent expressions to the same value just as the maximal-generator function does. McAllester shows how to convert this function into a grammar.

Unfortunately, translation of the standardizing function cannot be directly used for maintaining grammars in XRup. The reasoning system must assimilate new

```
G((g (f (f z))))
        |
        |----S((g (f (f z))))
        |            |
        |            |----G(g)
        |            |      |
        |            |      |----S(g)
        |            |      |      |
        |            |      |    S(g) = g
        |            |      |
        |            |    G(g) = (:EXPRESSION g)
        |            |
        |            |----G((f (f z)))
        |            |         |
        |            |         |----S((f (f z)))
        |            |         |         |
        |            |         |         |----G(f)
        |            |         |         |      |
        |            |         |         |      |----S(f)
        |            |         |         |      |      |
        |            |         |         |      |    S(f) = f
        |            |         |         |      |
        |            |         |         |    G(f) = F
        |            |         |         |
        |            |         |         |----G((f z))
        |            |         |         |       |
        |            |         |         |       |----S((f z))
        |            |         |         |       |       |
        |            |         |         |       |       |----G(f)
        |            |         |         |       |       |      |
        |            |         |         |       |       |      |----S(f)
        |            |         |         |       |       |      |      |
        |            |         |         |       |       |      |    S(f) = f
        |            |         |         |       |       |      |
        |            |         |         |       |       |    G(f) = F
        |            |         |         |       |       |
        |            |         |         |       |       |----G(z)
        |            |         |         |       |       |      |
        |            |         |         |       |       |      |----S(z)
        |            |         |         |       |       |      |      |
        |            |         |         |       |       |      |    S(z) = z
        |            |         |         |       |       |      |
        |            |         |         |       |       |    G(z) = Z
        |            |         |         |       |       |
        |            |         |         |       |    S((f z)) = (F Z)
        |            |         |         |       |
        |            |         |         |    G((f z)) = X
        |            |         |         |
        |            |         |    S((f (f z))) = (F X)
        |            |         |
        |            |    G((f (f z))) = Y
        |            |
        |    S((g (f (f z)))) = ((:EXPRESSION g) Y)
        |
G((g (f (f z)))) = (:EXPRESSION ((:EXPRESSION g) Y))
```

Figure 3. The algorithm for computing the generator of an expression goes through these steps when applied to the expression `(g (f (f z)))` *with the grammar described in the text. Invocations of the generator algorithm and the subexpression generator algorithm alternate and are labeled with G and S. The steps of the computation should be read downward. A vertical line connects the beginning of each computation with an indication of its returned value, and horizontal lines connect each computation with its subcomputations to the right. The steps of the completed generator computation could be read in reverse order, from bottom to top, to show a derivation of the expression from the generator according to the grammar.*

facts by incrementally modifying its internal representation. Reconstructing and retranslating the standardizing function each time a new equality is added would be too slow.

McAllester (1981b) has presented an incremental version of the Downey, Sethi, and Tarjan algorithm. At each step, the algorithm extends and modifies the current standardizing function to incorporate a new equality premise. Although incremental translation of an incrementally modified standardizing function is much faster than repeated reconstruction of the function and the grammar, it is still unsuitable for maintaining the grammar in XRup. McAllester's algorithm for assimilating a new equality is too slow because it exhibits the same individual consideration of equivalent variants that Figure 2 in section 1.1 illustrated. (See section 2.4 for details.)

2.3.2. Operating directly on the grammar

The correspondence between the standardizing function and the grammar can be used to transform the incremental algorithm for modifying the function into a new algorithm that manipulates the grammar directly. With the new method, the assimilation of a new equality between expressions X and Y involves two steps. In the first step, the grammar is expanded until it contains nonterminals X' and Y' to represent the equivalence classes of X and Y. This step does not affect the equality consequences represented by the grammar. In the second step, the two equivalence classes are merged by carefully replacing Y' with X'.

The replacement of Y' with X' merges the two equivalence classes by causing expressions that formerly were generated by Y' to be generated by X' instead. (In some cases the results are more complex.) The resulting grammar incorporates the desired equality because it generates X and Y from the same nonterminal. The replacement must be done carefully because one replacement may necessitate another. If the classes of A and B are merged, for example, the classes of (F A) and (F B) must also be merged.

2.3.3. Expanding the grammar

Grammar expansion requires the addition of new grammar symbols. For many purposes the spelling of the nonterminal symbols appearing in a grammar is arbitrary. The following two grammars are equivalent, for instance:

```
X ==> x | (F Z)          G1 ==> x | (G4 G3)
Y ==> y | (F X)          G2 ==> y | (G4 G1)
Z ==> z | (F Y)          G3 ==> z | (G4 G2)
F ==> f                  G4 ==> f
```

The spelling of each nonterminal used in XRup is derived from the spelling of some fixed expression chosen from the equivalence class that the nonterminal generates. The expression associated with a class symbol is called its *intrinsic term*. The rules of the grammar are maintained in such a way that the intrinsic term of a nonterminal is always in the equivalence class generated by the nonterminal if the nonterminal occurs in the grammar. This association between grammar symbols and expressions makes the grammar and its modification algorithms easier to understand than if symbols like `G1` above had been used.

The spelling of a class symbol is formed by placing brackets around the spelling of its intrinsic term. For example, a grammar equivalent to the one above could take the following form in XRup:

```
[X] ==> X | ([F] [(F Y)])
[Y] ==> Y | ([F] [X])
[(F Y)] ==> Z | ([F] [Y])
[F] ==> F
```

The notation $\tau(C)$ will be used for the intrinsic term of class C. That notation will also be extended to apply to all expression forms, including in particular the right-hand sides of grammar rules. If X is an expression form, $\tau(X)$ is obtained by replacing each occurrence of a nonterminal with its intrinsic term.

The grammar is expanded by using a modified version of the maximal-generator algorithm from section 2.2.3. In the implementation, the operation of enlarging the grammar to cover an expression is called computing the *forced maximal generator* of the expression because the operation forces the generator to be a nonterminal. The modified algorithm is listed here:

- To compute the *forced maximal generator* of an expression, set a flag to indicate that the generator algorithm should enlarge the grammar to ensure returning a nonterminal, then compute the generator of the expression.

- If u is an expression then its *generator* $G(u)$ is defined in terms of its subexpression generator $S(u)$. If a rule X ==> $S(u)$ appears in the grammar, then $G(u) = X$; otherwise, if a forced maximal generator is being computed, then let C be the class symbol with intrinsic term u, add the rule C ==> $S(u)$ to the grammar, and let $G(u) = C$; otherwise, let $G(u) =$ (:EXPRESSION $S(u)$).
- If u is an expression then its *subexpression generator* $S(u)$ is defined as follows. If u is atomic then $S(u) = u$. Otherwise u must be of the form $(u_1 \ldots u_n)$, and $S(u) = (G(u_1) \ldots G(u_n))$.

Consider adding the premises (= (F Y) X), (= (F X) Z), (= (G Y) 1), (= (G X) 1), and (= (H Y) 2), one by one, to an empty grammar. When the first premise, (= (F Y) X), is about to be added, grammar expansion yields the following grammar:

```
[F] ==> F
[Y] ==> Y
[(F Y)] ==> ([F] [Y])
[X] ==> X
```

The symbols [(F Y)] and [X] are returned as the generators of the expressions to be equated. Grammar expansion has changed the generators of expressions involving (F Y) but has introduced no new equality consequences. (In the empty grammar the expression (F (F Y)) has (F (F Y)) as its generator and is the only expression in its equivalence class; in the expanded grammar it has ([F] [(F Y)]) as its generator but is still the only expression in its class.)

2.3.4. Merging equivalence classes

After grammar expansion it is necessary to merge the equivalence classes of the newly equated terms. In the above example, the classes represented by the symbols [(F Y)] and [X] must be merged by replacing one symbol with the other. Assume that [(F Y)] is to be replaced with [X]. First the symbol [(F Y)] is replaced where it appears on the left-hand side of a rule:

```
[F] ==> F
[Y] ==> Y
[X] ==> X | ([F] [Y])
```

In this case the replacement is finished after this substep because no occurrences of [(F Y)] remain in the grammar. The premises (= (F X) Z), (= (G Y) 1), (= (G X) 1), and (= (H Y) 2) are added in similar fashion:

```
[F] ==> F
[Y] ==> Y
[X] ==> X | ([F] [Y])
[Z] ==> Z | ([F] [X])
[G] ==> G
[1] ==> 1 | ([G] [Y]) | ([G] [X])
[H] ==> H
[2] ==> 2 | ([H] [Y])
```

Now consider the addition of `(= X Y)`. Assume that `[Y]` is to be replaced with `[X]`. As usual, the replacement of left-hand occurrences is simple:

```
[F] ==> F
[X] ==> X | ([F] [Y]) | Y
[Z] ==> Z | ([F] [X])
[G] ==> G
[1] ==> 1 | ([G] [Y]) | ([G] [X])
[H] ==> H
[2] ==> 2 | ([H] [Y])
```

In this case, however, there are right-hand occurrences of the symbol to be replaced. Such replacements fall into three cases, and the example has been constructed to demonstrate them all.

Consider first the right-hand side `([H] [Y])`. The symbol `[Y]` has become obsolete and must be replaced with `[X]`. This case is simple because `([H] [X])` does not occur elsewhere in the grammar. The rule `[2] ==> ([H] [Y])` may simply be replaced with the rule `[2] ==> ([H] [X])`:

```
[F] ==> F
[X] ==> X | ([F] [Y]) | Y
[Z] ==> Z | ([F] [X])
[G] ==> G
[1] ==> 1 | ([G] [Y]) | ([G] [X])
[H] ==> H
[2] ==> 2 | ([H] [X])
```

Consider next the right-hand side `([G] [Y])`. The rule `[1] ==> ([G] [Y])` should be replaced with the updated version `[1] ==> ([G] [X])`. In this case, however, the desired new rule is already present. The only further action necessary is the deletion of the old rule:

```
[F] ==> F
[X] ==> X | ([F] [Y]) | Y
[Z] ==> Z | ([F] [X])
[G] ==> G
[1] ==> 1 | ([G] [X])
[H] ==> H
[2] ==> 2 | ([H] [X])
```

Consider finally the right-hand side `([F] [Y])`, which contains the only remaining occurrence of `[Y]`. The rule `[X] ==> ([F] [Y])` must be replaced with the rule `[X] ==> ([F] [X])`. Unfortunately, there is already a conflicting rule `[Z] ==> ([F] [X])`. The underlying problem is that although `Z` and `X` were not previously equal, they are now both equal to the expression `(F X)`. The solution is to merge the two conflicting classes. Replacement of `[Z]` with `[X]` requires only simple left-hand replacement:

```
[F] ==> F
[X] ==> X | ([F] [Y]) | Y | Z | ([F] [X])
[G] ==> G
[1] ==> 1 | ([G] [X])
[H] ==> H
[2] ==> 2 | ([H] [X])
```

Then the old rule mentioning `([F] [Y])` is deleted to produce the final grammar:

```
[F] ==> F
[X] ==> X | Y | Z | ([F] [X])
[G] ==> G
[1] ==> 1 | ([G] [X])
[H] ==> H
[2] ==> 2 | ([H] [X])
```

The equality `(= Y (F Z))` is one consequence that can be read from the new grammar. The following proof shows it to be a valid conclusion:

```
(= (F Y) X)              ; premise
(= (F X) X)              ; (= X Y)
(= Z X)                  ; (= (F X) Z)
(= (F Z) (F X))          ; corollary of (= Z X)

(= (F X) X)              ; above
(= (F X) Y)              ; (= X Y)

(= (F Z) Y)              ; consequence of sub-proofs
(= Y (F Z))              ; symmetry
```

2.3.5. Specific grammar construction algorithms

XRup uses roughly the following incremental procedure for adding an equality to the grammar. (Sections 3.3 and 3.4 will explain why the procedure can never introduce incorrect equality consequences.)

- To *assimilate an equality* (= X Y) into the grammar, first determine whether the grammar already equates X and Y. If X and Y are already equated, no further action is necessary. If X and Y are not already equated, let C_X and C_Y be the forced maximal generators of X and Y. Merge and replace C_X with C_Y in the grammar.
- To *merge and replace* nonterminal C_X with nonterminal C_Y, first replace every rule of the form C_X ==> r with the new rule C_Y ==> r. Next, record the fact that class symbol C_X has been "forwarded" to class symbol C_Y. Finally, as long as the grammar contains a rule C ==> r, where r contains C_X, do a right-hand forwarding adjustment on r.
- To do a *right-hand forwarding adjustment* on r, first recover the rule C ==> r containing r. Second, compute r' by following each class symbol in r to the end of its forwarding chain. Third, recover any rule C' ==> r' that occurs in the grammar. If there is no such rule, simply replace the rule C ==> r with the rule C ==> r'. If there is such a rule and $C' = C$, simply remove the rule C ==> r. Otherwise, recursively merge and replace C' with C, and then remove any rule involving r from the grammar.

The forwarding mechanism must be used when computing the new version of the right-hand side of a rule during nonterminal replacement. It is not always sufficient to simply substitute C_Y for C_X because a recursive merge operation may replace C_Y with some other symbol. Blind substitution of C_Y for C_X in such a case would reintroduce right-hand sides containing the obsolete symbol C_Y. (As later sections will show, the record of substitutions that the forwarding mechanism maintains is also useful in its own right.)

2.4. Grammars as Concise Representations

Grammar construction has now been described, and it is possible to illustrate the advantage of updating the grammar directly. Consider figure 2 from section 1.1. That example involves the following initial facts:

```
(= A1 A)         (= A2 A)         (= A3 A)         (= A4 A)
(= B1 B)         (= B2 B)         (= B3 B)         (= B4 B)
(= F G)

(= (F A1) X11)   (= (F A2) X12)   (= (F A3) X13)   (= (F A4) X14)
(= (G A1) X21)   (= (G A2) X22)   (= (G A3) X23)   (= (G A4) X24)

(= (F B1) X31)   (= (F B2) X32)   (= (F B3) X33)   (= (F B4) X34)
(= (G B1) X41)   (= (G B2) X42)   (= (G B3) X43)   (= (G B4) X44)
```

(Despite the large number of expressions, only a few distinct objects are mentioned; for example, `(G B2)` and `(F B4)` must name the same object because of the facts `(= F G)`, `(= B2 B)`, and `(= B4 B)`.) Consider first the function that McAllester's (1981b) incremental algorithm constructs to represent the consequences of these equalities. (A variant of this algorithm was implemented in the Rup system from which XRup developed.) Taking the equalities one by one, at each step it modifies the current standardizing function E, defined on the set of expressions D, by following this set of instructions:

- To *incorporate an equality* $(= u\ v)$, introduce the expressions u and v to D, then merge the classes of u and v.
- To *introduce an expression* u to D, if u is a member of D, do nothing, otherwise take the following steps. Recursively introduce each immediate subexpression of u to D. Let v be the subexpression image of u. If v is in D then let $v' = E(v)$; otherwise let $v' = v$. Add the expressions u and v to the set D, and set $E(u)$ and $E(v)$ to v'.
- To compute the *subexpression image* of a nonatomic expression $(u_1 \ldots u_n)$, return the expression $(E(u_1) \ldots E(u_n))$. To compute the subexpression image of an atomic expression, return the expression itself.
- To *merge the classes* of expressions u and v, if $E(u) = E(v)$, do nothing, otherwise take the following steps. Let S be the set of expressions $\{ x \in D{:}E(x) = E(u) \}$. For each term w in S, set $E(w)$ to $E(v)$. Then do a congruence check on each expression in D that contains an expression in S as an immediate subexpression. (A refinement of the algorithm switches u and v before merging, if that would result in fewer congruence checks here.)

- To do a *congruence check* on an expression u, first let u' be the subexpression image of u. If u' is in D then merge the classes of u' and u; otherwise add u' to the set D and set $E(u')$ to $E(u)$.

When all of the equalities have been incorporated, the standardizing function maps the expressions in D as follows:

```
A4, A3, A2, A1, A                  ==>     A
B4, B3, B2, B1, B                  ==>     B
G, F                               ==>     G

X24, (G A4), X23, (G A3),
X22, (G A2), X21, (G A1),
X14, (F A4), X13, (F A3),
X12, (F A2), X11, (F A1),
(G A)                              ==>     X24

X44, (G B4), X43, (G B3),
X42, (G B2), X41, (G B1),
X34, (F B4), X33, (F B3),
X32, (F B2), X31, (G B),
(F B1)                             ==>     X44
```

Now consider the addition of `(= A B)`. The only action necessary is merging the classes of `A` and `B`, since `A` and `B` are already in D. E(`A`) $=$ `A` and E(`B`) $=$ `B` are not the same; hence the algorithm proceeds to compute $S = \{$`A4, A3, A2, A1, A`$\}$. It changes E so that E maps each expression in S to `B` instead of `A`. It then performs congruence checks on the immediate parents of the expressions in S. This step involves individual consideration of a large number of equivalent variants. The first three lines of facts establish that `(G A4)`, `(F A4)`, `(G A3)`, `(F A3)`, `(G A2)`, `(F A2)`, `(G A)`, `(G A1)`, and `(F A1)` must all be equal, yet the instructions above require congruence checks on all of these expressions.

In contrast, consider the representation that XRup constructs to represent the same equalities. The implemented system produces the following grammar to represent the initial facts:

```
[A4] ==> A1 | A | A2 | A3 | A4
[B4] ==> B1 | B | B2 | B3 | B4
[G] ==> F | G

[X24] ==> ([G] [A4]) | X11 | X12 | X13 | X14
                     | X21 | X22 | X23 | X24

[X44] ==> ([G] [B4]) | X31 | X32 | X33 | X34
                     | X41 | X42 | X43 | X44
```

Consider again the addition of `(= A B)`. The steps taken by XRup parallel the steps taken in McAllester's algorithm. XRup first computes the generators of `A` and `B` as `[A4]` and `[B4]`. It then merges the classes that `[A4]` and `[B4]` represent by replacing `[A4]` with `[B4]`. Left-hand replacements are straightforward and correspond to McAllester's changes of standard form for the expressions in *S*. Right-hand replacements correspond to McAllester's congruence checks. McAllester's algorithm, however, represents the variants `(G A4)`, `(F A4)`, `(G A3)`, `(F A3)`, `(G A2)`, `(F A2)`, `(G A)`, `(G A1)`, and `(F A1)` separately; in XRup the grammar generates them all from `([G] [A4])`. XRup can replace the eight separate congruence checks with one adjustment to the generating form.

McAllester's algorithm stores and maintains correct standard versions of all known expressions; XRup stores and maintains a smaller representation that can generate correct standard versions as needed. In applications where only a few of the known expressions are examined between additions of facts, XRup will often be faster. In an application where every variant is always actively considered, however, XRup has no advantage. Although it avoids maintaining the standard version of every variant while facts are being stored, XRup must compute the standard version of each variant later when the application program considers it.

3. Recording Underlying Premises in Grammars

The algorithms of Chapter 2 tell how to build grammars and read consequences from them, but describe no way to trace derived conclusions back to underlying premises. This chapter remedies that lack by explaining how to add justifications to the grammar system. With the justification system in place, the system can justify derived conclusions by listing the premises that imply them.

The justification system works by associating a set of premises with each grammar rule. The procedure that reads consequences from the grammar is modified to accumulate premises as it traverses rules. The procedures that add rules to the grammar are modified to add appropriate justifications to the rules they construct.

3.1. Justifications for Grammar Rules

The grammar rules of earlier sections have the form

$$C \texttt{ ==> } r,$$

where C is a nonterminal and r is a terminal symbol or a list of nonterminals. Grammar rules with justifications have the form

$$C \texttt{ ==> } r \; ; \; j,$$

where j is a list of assertions called the *rule justification.* The elements of a rule justification should be facts and assumptions that have been declared true by the system that is using XRup as a component.

The rule C `==>` r ; j is said to be *valid* iff the assertions in j logically imply equality between $\tau(C)$ and $\tau(r)$. For example, the rule

```
[(F Y)] ==> ([F] [X])    ; (= X Y)
```

is valid because `(= X Y)` logically implies equality between `(F Y)` and `(F X)`. The rule should not appear in a grammar unless `(= X Y)` has been given to XRup as a permanent fact or temporary assumption, since it cites `(= X Y)` to justify its presence. (A later refinement to the justification mechanism relaxes that restriction slightly; see section 8.6.1.)

3.2. Building Justifications for Derived Conclusions

If each individual grammar rule is valid, rule justifications can be combined to produce correct justifications for consequences that are deduced from the grammar. The following recursive algorithm computes the *generator justification* of an expression:

- If u is an expression then its *generator justification* $J(u)$ is defined in terms of its subexpression generator $S(u)$ and its subexpression justification $J'(u)$. If a rule X ==> $S(u)$; j appears in the grammar then $J(u) = j \cup J'(u)$; otherwise $J(u) = J'(u)$.
- If u is an expression then its *subexpression justification* $J'(u)$ is defined as follows. If u is atomic then $J'(u)$ is the empty set. Otherwise u must be of the form $(u_1 \ldots u_n)$, and $J'(u) = J(u_1) \cup \ldots \cup J(u_n)$.

(This algorithm is parallel to the maximal-generator algorithm and formalizes the notion of accumulating rule justifications as the generator is computed. In practice the generator and the generator justification are computed at the same time, if the justification is required.) The important property of the generator justification is that it justifies the mapping from an expression to its generator:

- If u is an expression, $v = G(u)$, and all grammar rules are valid, then the assertions in $J(u)$ logically imply equality between u and $\tau(v)$.

In other words, the generator justification of an expression provides a set of premises that logically guarantee equality between the expression and its "standard form." Consequently, it is easy to produce enough premises to justify an equality consequence. To justify equality between expressions x and y, where x and y have the same maximal generator, produce the set of premises $J(x) \cup J(y)$.

3.3. Preserving Grammar Validity During Grammar Expansion

The only remaining question is how to construct rule justifications that make individual grammar rules valid. New rules are introduced when the grammar is expanded or new equalities are assimilated. The method of expanding the grammar needs only a small change. The maximal-generator algorithm, which is used for grammar expansion in addition to the computation of generators, should be altered to operate as follows:

- If u is an expression then its *generator* $G(u)$ is defined in terms of its subexpression generator $S(u)$. If a rule X `==>` $S(u)$; j appears in the grammar, then $G(u) = X$; otherwise, if a forced maximal generator for grammar expansion is being computed, then let C be the class symbol with intrinsic term u, add the rule C `==>` $S(u)$; $J'(u)$ to the grammar, and let $G(u) = C$; otherwise, let $G(u) =$ `(:EXPRESSION` $S(u)$`)`.

Suppose this procedure adds the rule C `==>` $S(u)$; $J'(u)$ to the grammar. If u is atomic then $J'(u)$ is empty, $S(u) = u$, and the new rule is trivially valid. Otherwise, u must be an expression $(u_1 \ldots u_n)$, and $J'(u) = J(u_1) \cup \ldots \cup J(u_n)$. The generator and generator justification of each u_i are computed solely on the basis of old rules, which may be assumed valid. Therefore, for each i, $J(u_i)$ is sufficient to imply equality between u_i and $\tau(G(u_i))$. Then all of the $J(u_i)$ taken together imply equality between u and $\tau(S(u))$, and the rule C `==>` $S(u)$; $J'(u)$ must be valid. Hence grammar expansion can never add an invalid rule where none were present before.

(The rule C `==>` $S(u)$; $J'(u)$ is not the only valid rule that could have been added above. The rule C' `==>` $S(u)$; $\emptyset$ could also have been added, where $\tau(C') = \tau(S(u))$; it would sometimes shorten proofs. The advantage of the above rule is that it causes the same class symbol to be shared among many grammars when a single new equality is incorporated into several different grammars.)

3.4. Preserving Grammar Validity During Premise Assimilation

It is harder to see how to maintain correct justifications during the assimilation of new facts. The following modified algorithm for digesting a new equality will work:

- To *assimilate a new equality* `(=` X Y`)` into the grammar, first determine whether the grammar already equates X and Y. If X and Y are already equated, no further action is necessary. If X and Y are not already equated, let C_X and C_Y be the forced maximal generators of X and Y. Merge and replace C_X with C_Y, justifying with $J(X) \cup J(Y) \cup \{$`(=` X Y`)`$\}$.
- To *merge and replace* nonterminal C_X with nonterminal C_Y, justifying with some justification J_{XY}, first replace every rule of the form C_X `==>` r ; j with the new rule C_Y `==>` r ; $j \cup J_{XY}$. Next, record the fact that class symbol C_X has been "forwarded" to class symbol C_Y with justification J_{XY}. Finally, as long as the grammar contains a rule C_X `==>` r ; j, where r contains C_X, do a right-hand forwarding adjustment on r.

- To do a *right-hand forwarding adjustment* on r, first recover the rule C `==>` r ; j containing r. Second, compute r' and J_{aux} by following each class symbol in r to the end of its forwarding chain and accumulating all forwarding justifications into J_{aux}. Third, recover any rule C' `==>` r' ; j' that occurs in the grammar. If there is no such rule, simply replace the rule C `==>` r ; j with the rule C `==>` r' ; $j \cup J_{aux}$. If there is such a rule and $C' = C$, simply remove the rule C `==>` r ; j. Otherwise, recursively merge and replace C' with C, justifying with $j \cup j' \cup J_{aux}$, and then remove from the grammar any rule involving r.

A few preliminary observations make it easier to explain why this procedure cannot introduce invalid rules if none were previously present.

First, if the rule X `==>` r ; j is valid and the premises in j' imply equality between $\tau(X)$ and $\tau(Y)$, then the rule Y `==>` r ; $j \cup j'$ is valid. For example, the rule

```
[(F Y)] ==> ([F] [X])    ; (= X Y)
```

can be transformed into the rule

```
[1] ==> ([F] [X])        ; (= X Y), (= (F Y) 1)
```

without loss of validity.

Second, if the rule C `==>` r ; j is valid and the premises in j' imply equality between $\tau(X)$ and $\tau(Y)$, then C `==>` r' ; $j \cup j'$ is valid, where r' is derived from r by substituting nonterminal Y for nonterminal X. For example, the rule

```
[(F X)] ==> ([F] [X])    ; ∅
```

can be transformed into the rule

```
[(F X)] ==> ([F] [Y])    ; (= X Y)
```

without loss of validity.

Third, if the rules X `==>` r ; j_X and Y `==>` r ; j_Y are valid, then $j_X \cup j_Y$ is sufficient to imply equality between the expressions $\tau(X)$ and $\tau(Y)$. For example, the fact that the rules

```
[A] ==> ([F] [X])        ; (= (F X) A)
[B] ==> ([F] [X])        ; (= (F X) B)
```

are both valid means that the premises `(= (F X) A)` and `(= (F X) B)`, taken together, imply that `(= A B)` must be true.

Proper use of the merge-and-replace procedure requires that all grammar rules be valid and that the premises in the argument J_{XY} be sufficient to imply equality between the intrinsic terms of the arguments C_X and C_Y. This condition is satisfied on the initial call from the assimilation procedure; $J(X)$ implies equality between $\tau(C_X)$ and X, `(=` $X\ Y$`)` implies equality between X and Y, and $J(Y)$ implies equality between Y and $\tau(C_Y)$. Consider now the left-hand replacement of C_X with C_Y. Each replacement of C_X `==>` r `;` j with C_Y `==>` r `;` $j \cup J_{XY}$ must preserve validity because the premises in J_{XY} imply equality between $\tau(C_X)$ and $\tau(C_Y)$. Consider next the right-hand replacements, which are performed by the right-hand forwarding adjustment procedure.

- The first kind of replacement that can occur involves changing C `==>` r `;` j into C `==>` r' `;` $j \cup J_{aux}$. In that case C `==>` r `;` j may be assumed to be valid, r' is derived from r by a substitution of class symbols, and the previously stored forwarding justifications that are accumulated into J_{aux} are sufficient to justify the substitutions; hence the new rule C `==>` r' `;` $j \cup J_{aux}$ is valid.
- The second kind of replacement that can occur involves deleting the rule C `==>` r `;` j. Such an action preserves validity because it introduces no new rules.
- The third kind of replacement that can occur involves a recursive call to the merge-and-replace procedure. In that case the rule C `==>` r `;` j appears in the grammar and may be assumed valid, r' is derived from r by a substitution of class symbols, and the previously stored forwarding justifications that are accumulated into J_{aux} justify the substitution that produces r'. Hence the rule C `==>` r' `;` $j \cup J_{aux}$ is valid. The rule C' `==>` r' `;` j' also appears in the grammar and may be assumed valid. Hence $j \cup j' \cup J_{aux}$ contains enough premises to guarantee equality between $\tau(C)$ and $\tau(C')$. The recursive invocation of the merge-and-replace procedure is proper because that equality is guaranteed.

This argument-sketch shows that the procedure for incorporating a new equality cannot add an invalid rule to a valid grammar. The first part of the analysis shows that the first invalid rule to be added would have to result from an improper invocation of the merge-and-replace procedure. The last part shows that no such improper invocations can occur. The assimilation of new facts cannot lead to improper conclusions. Because the grammar remains valid, the premises that the system produces in support of its conclusions are always logically sufficient to guarantee those conclusions.*

3.5. Assimilating Logically Necessary Premises

The above procedures for incorporating a new equality assume that there would be no reason to believe the equality if an external system had not declared it to be true. In some cases, however, the new equality may be known to express a logical truth. A logical truth needs no supporting premises and should not be listed as an underlying premise of the conclusions that it supports.

For example, when XRup is given the premise `(NOT (= (F X) (F Y)))`, its equality mechanisms will not immediately derive the conclusion `(NOT (= X Y))`. If `(= X Y)` is investigated as an assumption, XRup will discover that this assumption leads to a contradiction. In order to mark `(= X Y)` as false with proper underlying support, XRup will generate and equate to `TRUE` the following logically valid premise:

```
(OR (= (F X) (F Y))
    (NOT (= X Y)))
```

(Call that premise *A*.) The assimilation process will then derive the the conclusion `(NOT (= X Y))`, though the relevant deduction mechanism has not yet been described. Premise *A* can be added without harm to any set of assumptions because it is universally true. It can be omitted from any justification because, logically speaking, it adds nothing to a set of premises. In fact, premise *A* might confuse an external system if it were listed as a justifying premise, because that system did not supply premise *A* to XRup.

The procedure for assimilating an equality should be modified to give special treatment to known logical truths:

- To *assimilate a new equality* `(=` *X Y*`)` into the grammar, first determine whether the grammar already equates *X* and *Y*. If *X* and *Y* are already equated, no

*In a complete treatment of these algorithms, it would also be necessary to show that the grammar modification procedures preserve the integrity of the grammar as a whole in addition to preserving the validity of the individual rules. For example, it would be necessary to show that they never introduce two rules with the same right-hand side, or leave some nonterminal with right-hand occurrences but no left-hand occurrences. Such matters are not treated here.

further action is necessary. If X and Y are not already equated, let C_X and C_Y be the forced maximal generators of X and Y. Merge and replace C_X with C_Y, justifying with $J(X) \cup J(Y) \cup \{$ `(= X Y)` $\}$, except that if `(= X Y)` is known to express a logical truth, justify with $J(X) \cup J(Y)$ instead.

With this change, premises that are known to be logical truths will be omitted from justifications for derived conclusions.

3.6. An Example of Grammar Construction With Justifications

The operation of the modified procedures for grammar expansion and premise assimilation can be illustrated with the grammar-construction example from section 2.3. In that example, the premises `(= (F Y) X)`, `(= (F X) Z)`, `(= (G Y) 1)`, `(= (G X) 1)`, and `(= (H Y) 2)` are added sequentially to an empty grammar. When `(= (F Y) X)` is about to be added, grammar expansion yields the following grammar:

```
[F] ==> F
[Y] ==> Y
[(F Y)] ==> ([F] [Y])
[X] ==> X
```

(Empty rule justifications have been omitted.) The next step is to replace `[(F Y)]` with `[X]`, justifying with { `(= (F Y) X)` }:

```
[F] ==> F
[Y] ==> Y
[X] ==> X
     | ([F] [Y])          ; (= (F Y) X)
```

The premises `(= (F X) Z)`, `(= (G Y) 1)`, `(= (G X) 1)`, and `(= (H Y) 2)` are added similarly:

```
[F] ==> F
[Y] ==> Y
[X] ==> X
      | ([F] [Y])          ; (= (F Y) X)
[Z] ==> Z
      | ([F] [X])          ; (= (F X) Z)
[G] ==> G
[1] ==> 1
      | ([G] [Y])          ; (= (G Y) 1)
      | ([G] [X])          ; (= (G X) 1)
[H] ==> H
[2] ==> 2
      | ([H] [Y])          ; (= (H Y) 2)
```

As before, the addition of (= X Y) is more complicated. The replacement of left-hand occurrences of [Y] with [X] proceeds simply:

```
[F] ==> F
[X] ==> X
      | ([F] [Y])          ; (= (F Y) X)
      | Y                  ; (= X Y)
[Z] ==> Z
      | ([F] [X])          ; (= (F X) Z)
[G] ==> G
[1] ==> 1
      | ([G] [Y])          ; (= (G Y) 1)
      | ([G] [X])          ; (= (G X) 1)
[H] ==> H
[2] ==> 2
      | ([H] [Y])          ; (= (H Y) 2)
```

```
Forwarding table:

    [Y] forwarded to [X] because of (= X Y)
```

The first right-hand side to be adjusted is ([H] [Y]). Replacing forwarded symbols yields the new right-hand side ([H] [X]) and the associated justification (= X Y). Because ([H] [X]) does not occur in the grammar, it is only necessary to replace ([H] [Y]) with ([H] [X]) and adjust justifications:

```
[F] ==> F
[X] ==> X
      | ([F] [Y])          ; (= (F Y) X)
      | Y                  ; (= X Y)
[Z] ==> Z
      | ([F] [X])          ; (= (F X) Z)
[G] ==> G
[1] ==> 1
      | ([G] [Y])          ; (= (G Y) 1)
      | ([G] [X])          ; (= (G X) 1)
[H] ==> H
[2] ==> 2
      | ([H] [X])          ; (= (H Y) 2), (= X Y)
```

```
Forwarding table:

     [Y] forwarded to [X] because of (= X Y)
```

The next right-hand side to be adjusted is `([G] [Y])`. Ordinarily, the proper action would be to replace the rule

```
[1] ==> ([G] [Y])          ; (= (G Y) 1)
```

with the rule

```
[1] ==> ([G] [X])          ; (= (G Y) 1), (= X Y).
```

In this case, however, the rule

```
[1] ==> ([G] [X])          ; (= (G X) 1)
```

is already present (and valid); the only necessary action is the deletion of the obsolete rule involving `([G] [Y])`:

```
[F] ==> F
[X] ==> X
     |  ([F] [Y])       ; (= (F Y) X)
     |  Y               ; (= X Y)
[Z] ==> Z
     |  ([F] [X])       ; (= (F X) Z)
[G] ==> G
[1] ==> 1
     |  ([G] [X])       ; (= (G X) 1)
[H] ==> H
[2] ==> 2
     |  ([H] [X])       ; (= (H Y) 2), (= X Y)
```

```
Forwarding table:

    [Y] forwarded to [X] because of (= X Y)
```

Finally, the right-hand side `([F] [Y])` must be adjusted. The rule

```
[X] ==> ([F] [Y])       ; (= (F Y) X)
```

must be replaced with the rule

```
[X] ==> ([F] [X])       ; (= (F Y) X), (= X Y) .
```

Unfortunately, there is already a conflicting rule:

```
[Z] ==> ([F] [X])       ; (= (F X) Z)
```

The appropriate action is to replace `[Z]` with `[X]`, justifying with `(= (F Y) X)`, `(= X Y)`, and `(= (F X) Z)`. Only left-hand replacement is required:

```
[F] ==> F
[X] ==> X
     | ([F] [Y])        ; (= (F Y) X)
     | Y                ; (= X Y)
     | Z                ; (= (F X) Z), (= X Y),
                          (= (F X) Z)
     | ([F] [X])        ; (= (F X) Z), (= (F Y) X),
                          (= X Y)
[G] ==> G
[1] ==> 1
     | ([G] [X])        ; (= (G X) 1)
[H] ==> H
[2] ==> 2
     | ([H] [X])        ; (= (H Y) 2), (= X Y)

Forwarding table:

     [Y] forwarded to [X] because of (= X Y)
     [Z] forwarded to [X] because of (= (F Y) X), (= X Y),
                                     (= (F X) Z)
```

Then the old rule mentioning `([F] [Y])` is deleted to produce the final grammar:

```
[F] ==> F
[X] ==> X
     | Y                ; (= X Y)
     | Z                ; (= (F Y) X), (= X Y),
                          (= (F X) Z)
     | ([F] [X])        ; (= (F X) Z), (= (F Y) X),
                          (= X Y)
[G] ==> G
[1] ==> 1
     | ([G] [X])        ; (= (G X) 1)
[H] ==> H
[2] ==> 2
     | ([H] [X])        ; (= (H Y) 2), (= X Y)

Forwarding table:

     [Y] forwarded to [X] because of (= X Y)
     [Z] forwarded to [X] because of (= (F Y) X), (= X Y),
                                     (= (F X) Z)
```

After this grammar was constructed without justifications in section 2.3, the equality `(= Y (F Z))` was listed as a consequence that could be read from the grammar. Now that justifications have been added, it is possible to recover justifying premises from the grammar itself. The generator justification of `Y` is { `(= X Y)` }, the generator justification of `(F Z)` is { `(= (F Y) X)`, `(= X Y)`, `(= (F X) Z)` }, and the union of these two sets is { `(= X Y)`, `(= (F Y) X)`, `(= (F X) Z)` }.

3.7. The Inclusion of Unnecessary Justifications

In the above example the use of generator justifications recovers the same underlying premises that were used in the earlier hand-constructed proof. In general, however, the sets of underlying premises that a grammar produces will be larger than absolutely necessary. When the grammar is used to show two expressions equal and recover a set of underlying premises, that computation corresponds to the construction of a stylized equality proof. The system in effect proves that both expressions are equal to the same standard form. In many cases there are shorter proofs that do not have that structure.

With the above grammar, for example, the algorithms presented will list `(= X Y)` as an underlying premise for the conclusion `(= X X)`. They will list `(= X Y)`, `(= (F Y) X)`, and `(= (F X) Z)` as underlying premises for the conclusion `(= (F X) Z)`. There are simple methods for shortening some proofs, but in general the grammars described here should not be expected to produce minimal proofs for their conclusions. (As section 2.2.3 mentioned, it takes only linear average time to compute the generator of an expression. The generator justification can be accumulated as rules are traversed by the generator algorithm, though it takes some time to actually compute the union of the rule-justification sets. Searching for minimal proofs would be expected to take longer.)

4. Extending the Grammar Mechanism

The system described so far can derive consequences from equalities and trace consequences back to underlying premises. The scope of its reasoning, however, includes only the substitution of equals for equals. This chapter and the next two extend the grammar-based framework to encompass reasoning about truth values, logical connectives, constants, and temporary sets of assumptions.*

This chapter introduces refinements and elaborations in the framework and basic algorithms of the reasoning system. The next chapter uses the newly introduced features to give special treatment to such symbols as TRUE, FALSE, AND, OR, 1, and 2. Some refinements simplify system design by restricting the set of supported operations. Other refinements modify the grammar-based algorithms to allow logical symbols to have special properties.

4.1. Disallowing Retraction

Although previous sections tell how to add permanent facts to the grammar, XRup should also be able to direct attention to sets of premises that are only temporarily adopted. The ability to consider assumptions temporarily can be added to a deductive system by two major methods:

- Temporary assumptions can be added to the system in the same way as permanently asserted facts. When a new set of assumptions is considered, the old assumptions and their consequences must be removed.†
- The system can remember the consequences of several assumption sets at once and switch assumption sets by directing attention to different sets of

*McAllester (1982b) tells how to compute a grammar for the equivalence class of an expression under any finite disjunction of finite sets of equalities. His algorithms could conceivably be used as the basis for reasoning with propositional connectives. However, that approach will not be pursued here because the algorithms, though untested, appear to be computationally intractable. Note that a database containing disjunctions can become very large when converted to disjunctive normal form.

†The retraction of assumptions and their consequences is relatively easy in a TMS without equality reasoning because recorded justifications explicitly indicate which conclusions must be retracted. However, when the TMS is augmented with machinery for complete equality reasoning, retraction must cause the equality system to re-examine retracted equality conclusions and determine whether other valid justifications can be constructed for them. Consequently, in a system like Rup, removing assumptions may be roughly as difficult as adding them; it cannot be assumed that retraction is cheap.

remembered consequences. No direct removal of assumptions or consequences is necessary.

XRup records the underlying premises on which grammar rules depend. Although this seems to lay the foundation for the ability to remove premises from the grammar, a method of retracting premises is not immediately apparent. Let $A =$ { `(= X Y)`, `(= (F X) 1)`, `(= (F Y) 1)` }. Consider adding the premises in A to an empty grammar, one by one in the order listed. When only `(= X Y)` and `(= (F X) 1)` have been added, the following grammar will have been constructed:

```
[X] ==> X
      | Y                ; (= X Y)
[F] ==> F
[1] ==> ([F] [X])        ; (= (F X) 1)
```

When the premise `(= (F Y) 1)` is considered, no grammar alteration is necessary or possible; `(F Y)` and `1` are already equated by the grammar because they are both generated by the nonterminal `[1]`. Exactly the same grammar would be produced by adding only the premises in $B =$ { `(= X Y)`, `(= (F X) 1)` }.

Now consider retracting the premise `(= X Y)`. No retraction algorithm can work if it operates only on the grammar and uses the previously described grammar construction methods! Since premise set A and premise set B produce exactly the same grammar, a retraction algorithm that operates directly from the grammar cannot know whether `(= (F Y) 1)` should be true in the grammar it produces. `(= (F Y) 1)` should be true after `(= X Y)` is retracted from premise set A, but not after `(= X Y)` is retracted from premise set B. Because the expressions `(F X)` and `(F Y)` are equivalent variants of each other while the assertion `(= X Y)` is in effect, it seems unlikely that XRup could distinguish correctly between premise sets A and B without giving individual consideration to large numbers of equivalent variants.

Retraction is not implemented in XRup. Instead of using retraction to implement temporary assumption capabilities, XRup remembers the consequences of different assumption sets separately. It directs attention to a temporary assumption set by examining the grammar that encodes the consequences of that assumption set. (This mechanism will be explained in Chapter 6.) This technique avoids repeated recomputation of assumption consequences when an expert system switches repeatedly among a small number of assumption sets.

4.2. Disallowing Inconsistent Assumption Sets

Another refinement to the framework of XRup disallows the consideration of sets of assumptions that are internally inconsistent or inconsistent with permanently asserted facts. When a grammar is discovered to contain an inconsistency, the grammar modification that is in progress is aborted and the grammar is considered to become unusable.

It is not always apparent that the addition of a new premise will cause the grammar to become inconsistent. If the addition of a premise to a context leads to unforeseen inconsistency, information about the inconsistency is propagated through the system so that the premise will be marked as false in similar contexts.

Inconsistent contexts are discarded in XRup because they are expensive and inconvenient to maintain but seem to be of relatively little value. Consider the grammar that encodes a long list of facts of the following form:

```
(= (latitude Boston) ...)
(= (longitude Boston) ...)
(= (population Boston) ...)
(= (seasonal-average-temperature Boston summer) ...)
(= (seasonal-average-temperature Boston fall) ...)
(= (seasonal-average-temperature Boston winter) ...)
(= (seasonal-average-temperature Boston spring) ...)
(= (zip-code (main-post-office Boston)) ...)
(= (containing-state Boston) Massachusetts)
(= (mayor Boston) Kevin-White)
...
(= (latitude San-Francisco) ...)
...
(= (containing-state San-Francisco) California)
(= (mayor San-Francisco) Diane-Feinstein)
...
```

Suppose also that the following two facts are known:

```
(= (home-city X) Boston)
(= (home-city Y) San-Francisco)
```

(Assume further that `(= Boston San-Francisco)` is known to be false, though no way of representing that fact has yet been presented.) Now suppose an expert system is considering whether `X` and `Y` could be the same person.

It will not be obvious from the grammar that `X` and `Y` cannot be the same. The expert system can explore the question further by creating an assumption context in which `(= X Y)` is true. When the assumption is incorporated into the grammar and `(home-city X)` becomes equal to `(home-city Y)`, complete grammar update will include collapsing together all of the pairs of corresponding facts about Boston and San Francisco. That collapse, in turn, will necessitate collapsing together all of the pairs of corresponding facts about Massachusetts and California. It will require the system to smash together Kevin White and Diane Feinstein in the same way, and then proceed to their corresponding relatives.

The many questionable conclusions that complete grammar update produces in this case are all derivable by substitution of equals for equals from the given facts and assumptions. However, they seem entirely useless and irrelevant, they clutter up storage, and they are expensive to derive. Also, since XRup does not allow retraction, a grammar that becomes inconsistent will always remain so. Instead of carrying grammar update to the bitter end in the above case, XRup aborts grammar update and discards the partially modified context when it discovers that it is about to merge the classes of Boston and San Francisco. It salvages from the dying context the information that under any set of assumptions, one of the following statements must be true:

```
1. (not (= X Y))
2. (not (= (home-city X) Boston))
3. (not (= (home-city Y) San-Francisco))
4. (= Boston San-Francisco)
```

Statements 2, 3, and 4 are ruled out by known facts. The only remaining possibility is `(not (= X Y))`, and after the assumption `(= X Y)` has been investigated once, XRup will realize that the assumption is inconsistent with the facts. It will realize that `X` and `Y` cannot be equal.

(The fact that XRup does not consider inconsistent grammars is also convenient in other ways, which will be pointed out when appropriate.)

4.3. Upward and Downward Implications

XRup tries to keep its grammars small and expand them only when necessary. Its recursive algorithms project the effects of equalities upward to expressions that are larger than those that the grammar explicitly represents. Only the expressions X and Y are explicitly represented by the grammar that encodes the consequences of (= X Y), but the generator algorithm projects the effects of (= X Y) upward so that (F X) and (F Y) are equated.

Upward implications are consequences that are projected to an expression from its subexpressions. In the above example, (= (F X) (F Y)) is an upward implication of (= X Y). Similarly, (= (OR P Q) TRUE) is an upward implication of (= P TRUE). *Downward implications* are consequences that propagate downward from an expression to its subexpressions. (= P FALSE) is a downward implication of (= (AND P Q) FALSE).

4.3.1. Integrated algorithms and mechanical triggering

Different deductive systems use different mechanisms to give logical symbols their special properties. In some systems, deductions involving logical symbols are integrated into basic storage and retrieval algorithms. In other systems, basic algorithms have no knowledge of the special properties of logical symbols. Such systems can implement logical deductions by installing procedural links between logical expressions like (OR P Q) and related expressions like P and Q. The procedural mechanisms will be triggered to restore consistency whenever the truth values associated with (OR P Q), P, and Q change.

For example, Rup uses two kinds of triggering mechanisms to give special properties to symbols like "=" and OR. The interface between basic storage algorithms and the equality system is implemented by associating an explicitly procedural *change noticer* with each equality. The change noticer is triggered whenever the truth value of the equality changes. The noticer is responsible for restoring consistency between the equality and other statements.

Rup performs deductions involving OR by combining stylized clauses like the following with a simple mechanism, propositional constraint propagation, that is capable only of making a deduction whenever just one possibility in a clause remains (McAllester, 1981a, 1982a):

```
Either P is false, or (OR P Q) is true.
Either Q is false, or (OR P Q) is true.
Either P is true, or Q is true, or (OR P Q) is false.
```

Although they can be stated in declarative form, the above constraints are actually procedural in nature. Rup uses clauses for only one purpose; it uses a clause only to actively draw a conclusion when every possibility except one has been ruled out by previous conclusions. It does not use clause information in other ways; for example, the presence of the first clause above will not cause it to conclude that (OR (NOT P) (OR P Q)) is true, though it would eventually notice a contradiction if the opposite were assumed. Because clauses are primarily interpreted only in this limited and active way, a clause in effect constitutes a program for the clause deduction mechanism to execute when truth values change.

In a system like this, the upward implication from (= P TRUE) to (= (OR P Q) TRUE) is handled in the same way as the downward implication from (= (OR P Q) FALSE) to (= P FALSE). Consistent truth values among the expressions involved are maintained through the actions of the triggering network that is installed when the expressions are first seen by the deductive system.

With the triggering approach, an underlying database records the truth values and change noticers associated with each expression, and triggering mechanisms mechanically adjust stored truth values to maintain consistency. In contrast to this approach, when deduction patterns are well understood, it is often possible to integrate logical deduction into basic storage and retrieval mechanisms. When the underlying database functions as more than a memory, it is less necessary to mechanically propagate the consequences of new information throughout system storage. New information can be distilled into a representation that will let retrieval mechanisms read off certain consequences without search.

4.3.2. Upward and downward implications in XRup

The design philosophy of XRup does not allow it to use the triggering approach for upward implications. With that approach, the triggering network grows to include each new logical expression. This makes it harder to assimilate new information because it multiplies the amount of internal information whose consistency must be restored after a premise is added. XRup is designed to compute new consequences quickly by adjusting only a small, schematic internal representation. It does not maintain consistent information about all expressions that have been encountered; it projects the implications of stored premises onto external expressions only as required. XRup will not expand its internal database just because the expression `(OR P Q)` has been mentioned. (It would, however, expand its database to include `(OR P Q)` if a fact such as `(:-> (OR P Q) PHI)` were declared to be true.)

Instead of using the triggering approach for upward implications, XRup must integrate upward implications into the generator mechanism. It is the generator mechanism that projects consequences upward from the grammar to expressions that the grammar does not explicitly cover. However, the triggering approach is acceptable for downward implications. Because a grammar explicitly covers subexpressions whenever it explicitly covers an expression, the use of the triggering approach for downward implications does not require the grammar to be continuously expanded.

XRup captures the upward and downward implications that result from properties of logical symbols by using new features that are introduced in the following sections. The explicit rules of the grammar will be supplemented by mechanisms that do not use explicitly represented rules. This feature will be used in Chapter 5 to capture upward implications of logical statements. A restricted triggering mechanism will also be added. The triggering feature will be used to capture downward implications.

4.4. Logically Forced Generators

The previously described maximal-generator algorithm computes the generator of an expression by first computing the subexpression generator and then following a grammar rule backwards from the subexpression generator to a nonterminal. For example, with the grammar

```
[PHI] ==> PHI
[PSI] ==> PSI
[OR] ==> OR
[TRUE] ==> ([OR] [PHI] [PSI]) ; (= (OR PHI PSI) TRUE)
        | TRUE,
```

the generator algorithm computes the generator of the expression (OR PHI PSI) by first finding the subexpression generator ([OR] [PHI] [PSI]) and then following a rule backwards to get [TRUE]. (If no rule has the desired subexpression generator on its right-hand side, the algorithm returns the subexpression generator and a special marker.)

In order to capture certain upward implications, it is useful to modify the maximal-generator algorithm so that it gives special treatment to subexpression generators that correspond to applications of special operators. For example, with the slightly different grammar

```
[PSI] ==> PSI
[OR] ==> OR
[FALSE] ==> PHI           ; (= PHI FALSE)
         | FALSE,
```

it is desirable for the algorithm to return [PSI] as the generator of the expression (OR PHI PSI). This result captures the correct upward implication that (OR PHI PSI) equals PSI when PHI is known to be false.

4.4.1. Assigning generators to logical statements

Special treatment for statements that involve logical symbols can be arranged through the introduction of *logically forced generators.* Associated with the generator of OR, for instance, is a procedure for inspecting the subexpression generators of disjunctions. (Special treatment will also later be given to subexpression generators that are atomic constants like 1.) When presented with a subexpression generator, the procedure may take no special action, or it may dictate the value that should be returned as the corresponding generator. Such a procedure should be called during the computation of a generator whenever a subexpression generator that might be subject to the actions of the procedure is encountered; see section 4.11.1 for details.

In the above example, the subexpression generator of `(OR PHI PSI)` is `([OR] [FALSE] [PSI])`. The procedure associated with `[OR]` can eliminate the `[FALSE]` branch from `([OR] [FALSE] [PSI])` and declare that the remaining argument generator `[PSI]` should be the generator of the whole expression. It thus captures the desired implication.

4.4.2. Restricting logically forced generators

The mapping from a subexpression generator to a logically forced generator must express a logical truth. Unlike the mapping implied by an ordinary grammar rule, it does not have an associated rule justification. (The mapping from `([OR] [FALSE] [PSI])` to `[PSI]` is acceptable because `(= (OR FALSE PHI) PSI)` does express a logical truth.)

For technical reasons, the selection of a logically forced generator may involve only two possible actions. When presented with the subexpression generator $(C_f\, C_1\, \ldots\, C_n)$, the first possibility is for the procedure associated with C_f to promote an argument generator C_i to be the logically forced generator. The second possibility is for the procedure to return a nonterminal whose intrinsic term is a constant. (These restrictions, when combined with a special interpretation for nonterminals with constant intrinsic terms, will prevent logically forced generator rules from introducing meaningless or obsolete nonterminal symbols into a grammar.) In either case, the action taken by the procedure must be independent of the grammar involved.

4.4.3. Forced generators and implicit rules

Adding logically forced generators to the generator mechanism is much like adding implicit, logically forced rules to every grammar. For example, the mapping from the subexpression generator `([OR] [FALSE] [PSI])` to the logically forced generator `[PSI]` could be viewed as the result of augmenting the grammar with the implicit, logically forced rule `[PSI] ==> ([OR] [FALSE] [PSI]) ; Ø`.

The two views are not completely equivalent, however. With a grammar that contains only the two rules mapping OR to [OR] and FALSE to [FALSE], the subexpression generator algorithm that was previously described will compute ([OR] [FALSE] (:EXPRESSION PSI)) as the subexpression generator of (OR FALSE PSI). The logically forced generator corresponding to this subexpression generator is then (:EXPRESSION PSI). But the grammar could not contain the implicit rule

```
(:EXPRESSION PSI) ==> ([OR] [FALSE] (:EXPRESSION PSI))
```

because (:EXPRESSION PSI) is not a nonterminal symbol. (Expanding the grammar to cover such cases would defeat the goal of keeping the grammar small.)

4.5. Standardizing Subexpression Generators

The logically forced generator mechanism lets a special operator force a subexpression generator to map to a particular expression generator regardless of the grammar. When that mechanism does not apply, the slightly weaker *subexpression generator standardization* mechanism lets the operator say that certain subexpression generators must all have the *same* generator, without forcing the generator to a particular value. The AND operator, for instance, can say that whatever generator corresponds to the subexpression generator ([AND] [P] [Q]) in a particular grammar, it must be the same as the generator corresponding to ([AND] [Q] [P]).

A special operator achieves such a result by substituting a standardized version of a subexpression generator before the generator algorithm searches for the subexpression generator in the grammar. In the above example, AND might standardize both ([AND] [P] [Q]) and ([AND] [Q] [P]) by mapping them both to ([AND] [P] [Q]). In the grammar

```
[P] ==> P
[Q] ==> Q
[AND] ==> AND
[TRUE] ==> ([AND] [P] [Q])        ; (= (AND P Q) TRUE) ,
```

this would cause both (AND P Q) and (AND Q P) to have [TRUE] as maximal generator.

For technical reasons, the implementation does not allow operators that standardize subexpression generators to be equated to other expressions. Premises like `(= F =)` are forbidden.

Like the logically forced generator mechanism, the subexpression generator standardization mechanism requires transformations to express logical truths and to be independent of particular grammars. Also, although standardization of a subexpression generator may rearrange or eliminate argument generators, it may not introduce new argument generators (unless the new generators are nonterminals corresponding to constants). These restrictions help preserve the integrity of the grammar.

4.6. Analysis Procedures for Special Expression Forms

When the mechanisms of the previous two sections are in place, a special operator has associated with it a procedure for specifying logically forced generators and a procedure for standardizing subexpression generators. Because these two procedures are usually closely related, it is convenient to combine them into a single *special generator analysis procedure* associated with the operator.

A special generator analysis procedure takes a subexpression generator as its argument. It returns two values, an indicator and a datum. The indicator determines whether the datum is a logically forced generator or a standardized subexpression generator. (When no special action should be taken, the procedure should simply "standardize" the original subexpression generator without changing it.) Section 4.11.1 explains how special analysis procedures are integrated into the grammar-manipulation algorithms.

4.7. Noticing Constant Generator Assignment

Special generator analysis procedures can make the generator algorithm derive many upward implications. This section introduces a mechanism that lets the assimilation of new premises trigger many downward implications. For example, when premise assimilation adds the rule `[FALSE] ==> ([OR]` A B`)` ; j to a grammar, the OR operator can use the new triggering mechanism to recursively replace A and B throughout the grammar with `[FALSE]`, justifying with j.

The new mechanism allows a special operator to have an associated *constant generator noticer*. The noticer associated with special operator f is a procedure that will be run on any rule of the form C ==> r ; j that is added to the grammar, where C is a nonterminal such that $\tau(C)$ is a constant, and r corresponds to an application of f. For example, in the above example, the constant generator noticer associated with [OR] is run on the rule [FALSE] ==> ([OR] A B) ; j when that rule is added to the grammar. (Most constant generator noticers look for the case in which the generator being assigned to the subexpression generator is either [TRUE] or [FALSE].)

A constant generator noticer is not allowed to directly modify the grammar because direct modification would disturb the premise assimilation procedures that were in progress when the noticer was called. The noticer must instead enqueue an action that will later achieve the desired effect. The actions that are enqueued during the assimilation of a premise are performed as soon as the assimilation of the premise is finished. The most common enqueued action is *delayed merging*, which is described in section 5.3.2.

4.8. Carrying Attributes on Class Symbols

The next two sections introduce conventions for choosing nonterminal symbols to represent equivalence classes. When such conventions are followed, the choice of a particular nonterminal symbol to represent an equivalence class carries important information about the equivalence class itself. Some attributes of the equivalence class can then be read directly from the class symbol without consulting the grammar that defines class membership. This technique makes those attributes available to mechanisms whose operation must be independent of particular grammars. (Special generator analysis procedures are an important example of such mechanisms.)

Although the technique of carrying class attributes directly on class symbols is used heavily in XRup, the technique is quite limited in scope. Attributes cannot be encoded in this way unless it is possible for the class merging algorithms to maintain the correspondence between attributes of the representing symbol and the represented class.

The attributes that XRup carries on class symbols concern whether equivalence classes contain any expressions of a certain type. When two classes are merged, the final class has a special attribute just in case one of the original classes had the attribute. XRup maintains the correspondence between class attributes and special symbols by reversing the direction of symbol replacement if a special symbol is about to be replaced with an ordinary one. A problem arises if both of the original class symbols have special attributes and the two attributes conflict. With the attributes that XRup uses, such a clash can only be a sign of an inconsistent grammar. Consequently, XRup notes an inconsistency and aborts premise assimilation whenever there is conflict between the attributes of merged classes.

4.9. Special Treatment for Constants

When XRup is given `(= (F X) 1)`, `(= (F Z) 2)`, and `(= Y Z)` as premises, it should realize that `(F X)` and `(F Y)` cannot be equal. Such reasoning is easy to support if XRup can easily determine whether an expression is equated to any constant. This section introduces mechanisms that make the determination trivial by giving special treatment to constants.

The problem is simplified by the fact that inconsistent grammars need not be considered. An equivalence class can contain at most one constant because no consistent grammar can equate two different constants. As a result, it is possible to arrange for the generator of a constant to be the same nonterminal symbol in every grammar. This makes it easy to find out what constant (if any) is equated to an expression by a grammar. In the above example, XRup knows that `(F X)` is equated to `1` and no other constant because the generator of `(F X)` is the special nonterminal `[1]`.

The generators of constants are systematically fixed by assigning to each constant k the logically forced generator C, where C is the nonterminal such that $\tau(C) = k$. XRup automatically considers numbers, strings, and expressions of the form `(QUOTE` x`)` to be constants. Other symbols may also be declared to be constants; in particular, `TRUE`, `FALSE`, and other logical symbols will be declared as constants in Chapter 5.

With special logically forced generators for constants, the equivalence-class attribute of *containing a constant* is carried directly on class symbols. As section 4.8 explained, attributes that are encoded in this way are maintained by possibly switching the direction of symbol replacement when two classes are merged. When the premise (= X 1) is being incorporated into the empty grammar, for instance, the premise assimilation algorithms always replace [X] with [1]; they never replace [1] with [X].

A nonterminal symbol is said to be constant if its intrinsic term is a constant. Because they have special properties, constant nonterminals will henceforth be followed by asterisks when they are written. ([TRUE]* and [1]* are examples.)

4.10. Special Treatment for Assertions

Some expressions can have only TRUE and FALSE as values. If the assumption that (AND PHI PSI) equals TRUE leads to contradiction, (AND PHI PSI) must equal FALSE instead. Expressions that must equal either true or false are said to be *assertional.* Special deductions apply to assertional expressions. Suppose, for example, that φ is assertional. Then φ equals (= φ TRUE) and (NOT φ) equals (= φ FALSE).

An early implementation of XRup attempted to give assertional expressions special properties by automatically assimilating logical truths like

```
(OR (= (AND PHI PSI) TRUE)
    (= (AND PHI PSI) FALSE)) .
```

This approach made the implementation phenomenally slow, and still did not enable the system to easily realize that (AND PHI PSI) could not possibly equal 4. The approach also had the disadvantage of enlarging internal structures every time a new assertional statement was encountered.

4.10.1. Special symbols for assertional classes

XRup now makes classes that contain assertional expressions special in much the same way that it makes classes that contain constants special. Specially marked nonterminal symbols are used as the generators of assertional classes.

The identification of assertional terms is founded on a syntactic characterization of assertions. Some expressions are known to be assertional under every set of assumptions because of the symbols that make them up. (If φ is syntactically assertional, it is not considered necessary to mention that fact explicitly in justifications for reasoning about φ.) Other expressions that are equated to those expressions because of equality premises are also treated as assertional while the equality premises are in effect. (In the current implementation, an explicit statement like `(OR (= X TRUE) (= X FALSE))` does not cause `X` to be treated as assertional. It is still possible to make an arbitrary expression assertional in some contexts but not others, by introducing assertional dummy variables. Usually that is not necessary.)

Carrying an assertionality attribute on class symbols works for equivalence classes that have explicit nonterminal symbols to generate them. However, the assertionality mechanism must also be integrated into the generator mechanism because assertionality can be conferred through upward implication. The expression `(AND PHI PSI)` is assertional in an empty grammar, but this attribute cannot be carried on the nonterminal that generates `(AND PHI PSI)` because the grammar contains no nonterminals.

4.10.2. Predicate levels

The assertionality mechanism is integrated into the generator mechanism by generalizing assertionality into the notion of a *predicate level.* Expressions that must be either true or false are said to have predicate level zero. `AND` is not assertional, but all of its applications are; consequently, `AND` is "one level removed" from being assertional, and its predicate level is one. Similarly, if `F` stands for a second-level function that produces a first-level predicate as its output, then neither `F` nor `(F X)` is assertional, but `((F X) Y)` is assertional; consequently, `F` is "two levels removed" from being assertional, and its predicate level is two.

Since XRup disallows consideration of inconsistent grammars, the expressions within an equivalence class that have a predicate level must all have the same predicate level. When the generator of an equivalence class is an explicit nonterminal, the predicate level of the class is indicated by the predicate level of the intrinsic term of the nonterminal symbol. When the generator is an expression form, the predicate level associated with the generator could be computed by decomposing the expression form into its atomic constituents. However, it is more convenient to compute the predicate level while the generator is being computed. This makes the predicate levels of all generators easily accessible to special generator analysis procedures.

4.10.3. Predicate levels in the generator algorithm

The predicate level and the generator can always be computed at the same time if the generator algorithm builds the predicate level into the `:EXPRESSION` generator that it uses when the generator is not a nonterminal. Instead of returning the form `(:EXPRESSION` $S(x)$`)`, the algorithm should return `(:EXPRESSION` $S(x)$ p`)`, where p is the predicate level of the generated class. For example, operating on the expression $x =$ `(AND PHI PSI)` and an empty grammar, the generator algorithm first computes

```
S(x)  =  ([AND]* (:EXPRESSION PHI NIL)
                 (:EXPRESSION PSI NIL)),
```

then subtracts one from the predicate level of `[AND]*` to get the predicate level associated with the expression:

```
G(x)  =  (:EXPRESSION
           ([AND]* (:EXPRESSION PHI NIL)
                   (:EXPRESSION PSI NIL))
           0)
```

A program can then easily retrieve the predicate level associated with an equivalence class regardless of whether the class is generated by a nonterminal or an expression form.

4.10.4. Maintaining the predicate-level attribute

The equivalence-class attribute of *containing an expression with a certain predicate level* is carried directly on class symbols like the attribute of containing a constant. The encoding of each special attribute is maintained by carefully choosing which nonterminal to replace when two classes are being merged. Because XRup carries two different attributes on class symbols, the resulting two constraints on symbol replacement may conflict.

Each of the two distinct symbols involved in a replacement may have either or both of the constant (K) and predicate-level (P) attributes. If one symbol is ordinary, having neither P nor K, the ordinary symbol can be replaced without complication. If both symbols have K, a contradictory equality between two constants has been derived and grammar update should be aborted. If they both have P and the predicate levels differ, again a contradiction has been derived. If they both have P and the predicate levels are the same, the direction of replacement can be determined as if neither had P. The difficult case arises when one symbol has P but not K, and the other symbol has K but not P. Replacement in either direction would lose information, but no contradiction seems to have been derived.

The solution to this dilemma is to assume that complete knowledge about predicate levels is recorded on all constants. If a constant has no recorded predicate level, it must be because that constant *has* no predicate level. This assumption is valid for all of the expressions that XRup initially regards as constants. Consider for example the constant 2. It is neither true nor false, so its predicate level cannot be zero; it does not denote a function, so its predicate level cannot be greater than zero. 2 does not have a predicate level at all. (TRUE and FALSE, on the other hand, are constants with predicate level zero.) With this assumption, if the system attempts to merge a class that has K but not P with a class that has P but not K, then it has derived a contradiction and should abort grammar update.

4.11. Extended Grammar Algorithms

Previous sections of this chapter have extended the grammar-based framework of XRup in several ways. This section revises the basic grammar algorithms to take the extensions into account. The generator and generator justification algorithms must take into account the new mechanisms that operate outside the grammar. The procedures that incorporate new equalities into the grammar must not bring grammar rules into conflict with those mechanisms.

4.11.1. Revised generator algorithms

The top-level method of reading equalities from the grammar remains the same:

- To determine whether two expressions u_1 and u_2 are *equal according to the grammar*, compute the generators of u_1 and u_2 and see whether they are the same.

The computation of generators changes in four ways. Logically forced generators must be respected, subexpression generator standardization must be performed, predicate levels must be computed when generators are expression forms, and nonterminals with the correct predicate levels must be introduced during grammar expansion. The first new procedures deal with the special analysis procedures that specify logically forced generators and standardize subexpression generators:

- To compute the *logically forced generator* of a subexpression generator r, consider the following cases. If r is an atomic constant, then its logically forced generator is the nonterminal with intrinsic term r. If r is atomic but is not a constant, then it has no logically forced generator. Otherwise, r must be of the form $(r_1 \ldots r_n)$. If r_1 has a special generator analysis procedure f and the computation $f(r)$ specifies a logically forced generator, then that is the logically forced generator of r. Otherwise, r has no logically forced generator.
- To *standardize a subexpression generator* r, consider the following cases. If r is atomic, then its standard form is r. Otherwise, r must be of the form $(r_1 \ldots r_n)$. If r_1 has a special generator analysis procedure f and the computation $f(r)$ specifies a standard form for r, then that is the standard form of r. Otherwise, the standard form of r is r.

Next, a few operations dealing with predicate levels are introduced; recall that symbols like TRUE and FALSE have declared predicate levels:

- To *compute the predicate level of an expression* u, consider the following cases. If u is atomic and has a declared predicate level, use the declared predicate level. If u is atomic, has no declared predicate level, and is a constant, then use —1 as the predicate level. If u is atomic, has no declared predicate level, and is not a constant, then u has no predicate level. If u is not atomic, then u must be of the form $(u_1 \ldots u_n)$. Recursively compute the predicate level of the expression u_1. If u_1 has a positive predicate level p, then the predicate level of u is $p - 1$; otherwise, u has no predicate level. (Whenever a new nonterminal symbol is created, the predicate level of its intrinsic term should be computed and stored.)
- To *retrieve the predicate level of a generator* g, consider the following cases. If g is a nonterminal, then take the (pre-computed) predicate level of $\tau(g)$. Otherwise, g must be of the form (:EXPRESSION s p); take p.
- To *compute the predicate level of a subexpression generator* r, consider the following cases. If r is an atomic nonterminal, then take the predicate level of r considered as an expression. Otherwise, r must be of the form $(r_1 \ldots r_n)$. Retrieve the predicate level of the generator r_1. If r_1 has a positive predicate level p, then the predicate level of r is $p - 1$; otherwise, r has no predicate level.

Finally, these new procedures are tied into the generator algorithm:

- If u is an expression then its *generator* $G(u)$ is defined in terms of its subexpression generator $S(u)$. If $S(u)$ has a logically forced generator g, then $G(u) = g$. Otherwise, standardize the subexpression generator $S(u)$ to produce r. If a rule X `==>` r ; j appears in the grammar, then $G(u) = X$. Otherwise, if a forced maximal generator for grammar expansion is being computed, then let C be the class symbol with intrinsic term u, add the rule C `==>` r ; $J'(u)$ to the grammar, and let $G(u) = C$. (If C was newly created, compute and store the predicate level of its intrinsic term.) Otherwise, let p be the predicate level of the subexpression generator r, and let $G(u) =$ `(:EXPRESSION` r p`)`.
- If u is an expression then its *subexpression generator* $S(u)$ is defined as follows. If u is atomic then $S(u) = u$. Otherwise u must be of the form $(u_1 \ldots u_n)$, and $S(u) = (G(u_1) \ldots G(u_n))$.

(The generator algorithm must actually be slightly more complex than this. When a forced maximal generator is being computed, the above version of the algorithm assumes that the predicate levels of u and r are the same. That need not be true.)

A corresponding small change in the algorithm for computing generator justification is also needed:

- If u is an expression then its *generator justification* $J(u)$ is defined in terms of its subexpression generator $S(u)$ and its subexpression justification $J'(u)$. If $S(u)$ has a logically forced generator, then $J(u) = J'(u)$. If $S(u)$ does not have a logically forced generator, standardize the subexpression generator $S(u)$ yielding r. If a rule X `==>` r ; j appears in the grammar, then $J(u) = j \cup J'(u)$. Otherwise, $J(u) = J'(u)$.
- If u is an expression then its *subexpression justification* $J'(u)$ is defined as follows. If u is atomic then $J'(u)$ is the empty set. Otherwise, u must be of the form $(u_1 \ldots u_n)$, and $J'(u) = J(u_1) \cup \ldots \cup J(u_n)$.

4.11.2. Revised premise assimilation algorithms

The premise assimilation algorithms must also be adjusted. The top-level procedure for premise assimilation remains unchanged:

- To *assimilate a new equality* $(= X\ Y)$ into the grammar, first determine whether the grammar already equates X and Y. If X and Y are already equated, no further action is necessary. If X and Y are not already equated, let C_X and C_Y be the forced maximal generators of X and Y. Merge and replace C_X with C_Y, justifying with $J(X) \cup J(Y) \cup \{(= X\ Y)\}$, except that if $(= X\ Y)$ is known to express a logical truth, justify with $J(X) \cup J(Y)$ instead.

Internal procedures, however, must change. The merge-and-replace procedure must abort grammar update when an inconsistency is discovered. It must also sometimes reverse the direction of symbol replacement in order to preserve attributes that are carried directly on class symbols. Finally, when reversal of replacement direction is not otherwise necessary, it may still be desirable in order to promote efficiency. These possible actions can be expressed as a preprocessing step in the procedure to merge and replace one nonterminal with another:

- To *preprocess the merge arguments* C_X, C_Y, and J_{XY}, first signal an error if C_X and C_Y are the same symbol, since this condition should not occur. Second, consider the special constant (K) and predicate-level (P) attributes of the classes C_X and C_Y. If both C_X and C_Y have K, abort the merging of C_X and C_Y, with justification J_{XY}. If both C_X and C_Y have predicate levels but the levels are not the same, abort the merging of C_X and C_Y, with justification J_{XY}. Otherwise, if C_X has a special attribute that C_Y lacks, or if C_X and C_Y have the same special attributes but C_Y appears on the right-hand sides of fewer rules than C_X, then switch the arguments C_X and C_Y to the merge procedure.
- To *abort the merging* of C_X and C_Y, with justification J_{XY}, abort grammar update, discard the partially modified grammar, and propagate to all contexts the information that either some statement in J_{XY} must be false, or $(= \tau(C_X)\ \tau(C_Y))$ must be true.

Other changes in the premise assimilation methods take into account logically forced generators and subexpression generator standardization. The following algorithms result:

- To *merge and replace* nonterminal C_X with nonterminal C_Y, justifying with some justification J_{XY}, first preprocess the merge arguments C_X, C_Y, and J_{XY}. Then replace every explicitly stored grammar rule of the form C_X ==> r ; j with the new rule C_Y ==> r ; $j \cup J_{XY}$. Next, record the fact that class symbol C_X has been "forwarded" to class symbol C_Y with justification J_{XY}. Finally, as long as the grammar contains a rule C_X ==> r ; j, where r contains C_X, do a right-hand forwarding adjustment on r.
- To do a *right-hand forwarding adjustment* on r, first recover the explicitly stored grammar rule C ==> r ; j containing r. Second, compute r' and J_{aux} by following each class symbol in r to the end of its forwarding chain and accumulating all forwarding justifications into J_{aux}. Then, if r' has a logically forced generator g, let $C' = g$ and let $j' = \emptyset$, or if r' does not have a logically forced generator, standardize the subexpression generator r' yielding r'', and recover any rule C' ==> r'' ; j' that occurs in the grammar. Finally, consider three cases. If r' had no logically forced generator and there was no rule

involving r'', then replace the rule C ==> r ; j with the rule C ==> r'' ; $j \cup J_{aux}$. If $C' \doteq C$, simply remove the rule C ==> r ; j from the grammar. Otherwise, recursively merge and replace C' with C, justifying with $j \cup j' \cup J_{aux}$, and then remove from the grammar any rule involving r.

This completes the list of revised basic algorithms. The example in section 5.5 shows them in action.

4.12. Switching Between Grammars

A final refinement in the framework of the reasoning system lets XRup direct its attention to different assumption sets at different times. The mechanism will not be explained in detail until Chapter 6, but a preview is appropriate.

XRup uses a separate grammar for each set of assumptions. In order to investigate a previously represented set of assumptions plus one new assumption, XRup copies the old grammar and adds the new premise to the copy. It then saves the new grammar and indexes it by the set of assumptions that it represents. As long as the grammar is retained, repeated reconsideration of a set of assumptions does not result in repeated reconstruction of the associated grammar. Efforts are made to keep the representation of a grammar small and easy to copy.

In addition to the grammars that encode sets of temporary assumptions, XRup maintains a *global* grammar that encodes only permanent facts. The grammar that is associated with a set of temporary assumptions is intended to represent the consequences of all global facts plus the given temporary assumptions. The correspondence is maintained as new information enters the system; each new permanent fact is added to every saved grammar, but no non-permanent premises are added to a grammar once it has been created.

5. Reasoning With Special Operators

Chapter 4 extended the grammar-based framework by allowing for special treatment of logical symbols. This chapter shows how XRup can use the new features to reason with truth values, logical connectives, and constants.

5.1. Disequalities Involving Constants

A statement that explicitly equates two expressions is called an equality. The negation of such a statement can be called a *disequality*, and two expressions can be said to be disequated if equality between them is explicitly ruled out. (The term "disequality" is more specific than "inequality" because it rules out such statements as `(< X Y)`; "disequated" is more specific than "unequated" because it rules out the possibility that equality is neither affirmed nor denied.)

When XRup is given `(= (F X) 1)`, `(= (F Z) 2)`, and `(= Y Z)` as premises, it can derive the disequality `(NOT (= (F X) (F Y)))`. It reaches the conclusion through three steps. First, it realizes that `(F X)` and `(F Y)` cannot be equal. Second, it realizes that if two expressions cannot be equal, then a statement equating them must be false. Third, it concludes that the negation of a false statement must be true. The second and third steps, which involve the properties of the symbols `=` and `NOT`, will be covered in later sections. This section explains the first step.

The situations of interest here are those in which the grammar equates two expressions to two different constants. `(F X)` and `(F Y)` cannot be equal because `(F X)` equals `1` but `(F Y)` can be shown to equal `2`. The logically forced generators that were assigned to constants in section 4.9 make such facts easily visible to XRup. Because the generator of the constant `1` is always the nonterminal `[1]*`, an expression is equated to `1` only if its generator is also `[1]*`. Two expressions are equated to different constants if and only if the intrinsic terms of their generators are different constants.

Similar methods make it easy to see disequalities between expressions with conflicting predicate levels. If `V` is known to equal `3` and `W` is known to equal `(OR PHI PSI)`, XRup can easily determine that `V` cannot equal `W`. In the grammar that encodes only the premise (= W (OR PHI PSI)), the generator of W is [(OR PHI PSI)]. The generators [(OR PHI PSI)] and [3]* are incompatible; [(OR PHI PSI)] has predicate level zero, but [3]* is known to have no predicate level.

5.2. Explicit Representation of Truth Values

XRup deals primarily with equalities. It handles other kinds of statements through a trivial translation. Any statement can be affirmed by equating it to TRUE, or denied by equating it to FALSE. This section explains the special treatment that XRup gives to TRUE and FALSE.

The first step is to declare TRUE and FALSE as constants. This step fixes the generators of TRUE and FALSE as [TRUE]* and [FALSE]*. Consequently, a grammar marks a statement as true just in case it assigns to the statement the generator [TRUE]*. It marks a statement as false just in case it assigns the generator [FALSE]*.

The second step is to declare the predicate levels of TRUE and FALSE as zero. Since TRUE and FALSE will be the only constants with predicate level zero, this will prohibit an assertional expression from being equated to any constant except TRUE or FALSE.

XRup also gives the constants TRUE and FALSE special treatment in equalities and logical constructions. This special treatment will be explained in later sections.

5.3. Understanding the Equality Symbol

The basic algorithms that XRup uses already deal with equalities and their consequences. So far, however, there is nothing to connect equality with the explicit symbol "=" that represents it. XRup can tell that 1 and 1 are equated, but it does not know that (= 1 1) equals TRUE. It can tell that 1 and 2 are disequated, but it does not know that (= 1 2) equals FALSE. This section explains how XRup can capture upward and downward implications that are connected with the equality symbol. The upward implications are captured with logically forced generators and subexpression generator standardization. The downward implications are captured with constant generator assignment noticers. There are also mechanisms to reduce unnecessary nesting of equality expressions.

5.3.1. Upward implications

The first step is to make the equality symbol a constant with predicate level one. When this is done, the generator of the symbol "=" is always the nonterminal [=]*. Consequently, it is possible to tell whether a subexpression generator represents an equality, without knowing the rest of the grammar. It can be determined definitely that ([=]* [X] [Y]) represents an equality, but ([P] [X] [Y]) does not.

The next step is to recognize that equality is commutative. Before considering the subexpression generator ([=]* X Y), the special generator analysis procedure for [=]* should put the argument generators X and Y into standard order. (The order used has the property that nonterminals representing constants come first.) If the rearranged subexpression generator falls under none of the special cases listed below, the analysis procedure standardizes the subexpression generator by returning the rearranged version.

It is also necessary to connect the equality symbol to the grammar-based characterizations of equated and disequated pairs of expressions. An equality between equated expressions must be true; an equality between disequated expressions must be false. These facts can be captured through the assignment of logically forced generators. A subexpression generator of the form ([=]* X X) is assigned the logically forced generator [TRUE]*. A subexpression generator of the form ([=]* X Y), where X and Y are distinct constant symbols or have conflicting predicate levels, is assigned the logically forced generator [FALSE]*.

Finally, it is necessary to give special treatment to assertional classes. In the following sentences, φ stands for a generator with predicate level zero. A subexpression generator of the form ([=]* [TRUE]* φ) is assigned the logically forced generator φ. A subexpression generator of the form ([=]* [FALSE]* φ), on the other hand, is standardized by converting it to the subexpression generator ([NOT]* φ).

5.3.2. Downward implications

The above steps capture many of the upward implications that are connected with the equality symbol. A simple constant generator assignment noticer captures downward implications from true equalities. When the rule [TRUE]* ==> ([=]* X Y) is added to the grammar, the classes represented by X and Y must be merged to incorporate the new equality.

Because the actions of a noticer are not carried out immediately, however, the correct action is actually a slightly more complicated *delayed merging*. Delayed merging takes into account the possibility that the symbols to be merged may have already been replaced with other symbols when the merging takes place:

- To carry out the *delayed merging* of nonterminals X and Y, with justification j, first follow the class-forwarding chains of X and Y to yield a new X and Y. While following class forwarding, add the accumulated forwarding justifications into j. Then, if $X = Y$, do nothing; otherwise, merge and replace X with Y, justifying with j.

When it is run on the rule K ==> ([=]* X Y) ; j, the noticer for [=]* should do nothing unless $K =$ [TRUE]*. If $K =$ [TRUE]*, the noticer should enqueue the delayed merging of X and Y, with justification j.

The corresponding downward implication from a false equality says that the arguments of a false equality cannot be equal. Since XRup cannot fundamentally assimilate disequalities as it can assimilate equalities, it captures this implication through its definition of *disequated according to the grammar:*

- To determine whether expressions X and Y are *disequated according to the grammar*, determine whether the generator of (= X Y) is the nonterminal [FALSE]*.

With this definition, XRup will correctly determine that the arguments of a false equality are disequated. It will also notice the disequalities involving constants that were treated in section 5.1, since the above rules assign [FALSE]* as the logically forced generator of subexpression generators like ([=]* [1]* [2]*).

One other kind of downward implication is not directly captured. If (G (F X)) and (G (F Y)) are disequated, X and Y should be disequated also. XRup does not draw this conclusion immediately because there seems to be no easy way to do so. However, if the assumption (= X Y) is investigated, XRup will correctly discover that the assumption leads to contradiction. It will then realize that X and Y cannot be equal.

5.3.3. *Deep nesting of equalities*

XRup also has a feature that eliminates "useless" nesting of equalities. In the early development of the system, expressions like the following were often created:

```
(= (= (= (= (= X Y) TRUE) FALSE) TRUE) FALSE)
```

This expression is equivalent to `(= X Y)`, and the previously described mechanisms can correctly derive the equivalence. Such deeply nested expressions are difficult to understand, however, and there seems to be little advantage in dealing with the deeply nested forms instead of shallower equivalents. XRup now collapses deeply nested equalities whenever they are encountered. It goes beyond considering a deeply nested equality to be equated to its shallow equivalent; for its purposes, it considers the two to be different spellings of the very same expression. It uses the same internal structure to represent them both. In the same way, it also considers `(= X Y)` and `(= Y X)` to be the same expression.

Let φ be used for assertional expressions; let v be used for truth values. The rules that XRup uses to remove useless nesting leave equalities as equalities; they do not strip (= φ TRUE) down to φ. However, they do collapse more deeply nested expressions. (= (= φ v) TRUE) collapses to (= φ v) and (= (= φ v) FALSE) collapses to (= φ v'), where v' is the opposite of the truth-value v.

The expression preprocessing mechanism also allows the symbol ":↑=" to be used as a disequality symbol. It expands uses of ":↑=" into negated equalities.

5.4. Understanding Propositional Connectives

This section explains how XRup makes deductions that involve the propositional connectives AND, OR, NOT, and `:->` (the logical implication symbol). Some such deductions seem necessary for any system that deals with logical connectives. If (AND A B) is true, then both A and B are true. If either A or B is false, then (AND A B) is false. If (OR A B) is true and A is false, then B must be true. In addition, there are other deductions that seem particularly natural in an equality-based system. If A is true, then B, (AND A B), and (OR (NOT A) B) are all equivalent. If (OR A B C) is true, A is false, and B and C are equivalent, then B and C must be true.

As usual, XRup captures upward implications by using logically forced generators and subexpression generator standardization. It captures downward implications by using constant generator assignment noticers. Some connectives are treated as abbreviations for other connectives. Such abbreviations are expanded during the preprocessing of expressions. This expansion simplifies deductive machinery because it reduces the number of logical symbols that the internal mechanisms of XRup must consider.

XRup treats the symbols AND, OR, and NOT as constants with predicate level one. In addition, it requires the expressions that are joined with logical connectives to be syntactically assertional. If φ is syntactically assertional but x is not, the expression (OR φ x) is ill-formed. Because x might be equated to an assertional expression even though x itself is not assertional, XRup repairs the ill-formed term by treating it as (OR φ (= x TRUE)). The expression (= x TRUE) is syntactically assertional, and if x is equated to an assertional expression, the outer equality will have no effect. (The arguments of logical connectives must be assertional if certain deductions are to make sense.)

5.4.1. Upward implications for AND and OR

The special generator analysis procedure for [OR]* receives a subexpression generator of the form ([OR]* $G_1 \ldots G_n$) and may rearrange the G_i or specify a logically forced generator. It begins its analysis by removing duplicates and occurrences of [FALSE]* from the G_i. It then counts the remaining G_i. If there are none left, it returns the logically forced generator [FALSE]*. If there is only one argument generator left, it returns that generator as a logically forced generator. Finally, if there is more than one argument generator left, the analysis procedure standardizes the subexpression generator by sorting the remaining argument generators into a standard order $G'_1, \ldots, G'_m$ and returning the disjunction ([OR]* $G'_1 \ldots G'_m$).

This procedure eliminates occurrences of [FALSE]* in order to capture the fact that (OR $A_1 \ldots A_j \ldots A_n$) equals A_j when every other A_i is false. It eliminates duplicate argument generators because duplicate arguments to OR are redundant. The procedure captures the fact that a disjunction is false when all of its disjuncts are false, since it returns [FALSE]* when no non-false argument generators remain. It captures the fact that (OR A B) equals B when A is false, since it returns the remaining argument generator if only one remains. This action also captures the fact that (OR A B) and (NOT B) together imply A. Finally, the analysis procedure accommodates the fact that OR is commutative, since it sorts argument generators into standard order when there is no logically forced generator.

The special generator analysis procedure for [AND]* is similar to the one for [OR]*. The only difference is that [TRUE]* plays the role that [FALSE]* plays in the procedure for [OR]*.

5.4.2. *Downward implications for AND and OR*

XRup has only one rule for capturing downward implications connected with AND. A constant generator noticer associated with AND triggers on the insertion of each rule of the form [TRUE]* ==> ([AND]* $C_1 \ldots C_n$) ; j. The noticer enqueues the delayed merging of each C_i with [TRUE]*, justifying with j. The noticer for [OR]* is similar but has [FALSE]* in the role of [TRUE]*.

5.4.3. *Implications connected with NOT*

The special actions associated with NOT are simple. The special generator analysis procedure declares [FALSE]* as the logically forced generator corresponding to ([NOT]* [TRUE]*) and declares [TRUE]* as the generator for ([NOT]* [FALSE]*), but otherwise takes no special action. A constant generator noticer merges C with [FALSE]* when the rule [TRUE]* ==> ([NOT]* C) ; j is added to the grammar; a similar action applies when the rule involves [FALSE]* instead.

The symbol NOT also receives special treatment during the preprocessing of expressions. The expression (NOT φ) is treated internally as the expression (= φ FALSE). When no more specialized action applies, the subexpression generator analysis procedure for [=]* transforms this back into ([NOT]* $G(\varphi)$).

5.4.4. *Translating the implication symbol*

XRup treats the logical implication symbol by regarding it as an abbreviation. The implication (:-> φ ψ) is regarded as the same expression as (OR ψ (NOT φ)).

5.5. An Example of Propositional Reasoning

This section shows the special treatment of logical symbols in action. The following example has been constructed to demonstrate most of the special mechanisms that were added to the system. Consider adding the following premises to an initially empty grammar:

```
1. (OR (= (F X) 1)
       (= (F Y) 1)
       (= (F Z) 2))
2. (= X Y)
3. (:-> PHI (= (F Z) 3))
4. PHI
```

(Assume that `PHI` is declared syntactically assertional.) Figures 4–9 illustrate the major stages in the assimilation of these premises. The following discussion tells where the effects of the new mechanisms show up in those figures.

5.5.1. *Incorporating the first premise*

Figure 4 shows the grammar after the first statement has been assimilated by equating it to `TRUE`. The changes in the grammar-based algorithms have little effect on the addition of this premise. Note, however, that the special generator analysis procedure associated with `[=]*` has standardized the right-hand sides of rules in which that symbol occurs. The grammar equates the expressions `(= (F X) 1)` and `(= 1 (F X))`, since the subexpression generators `([=]* [(F X)] [1]*)` and `([=]* [1]* [(F X)])` both standardize to `([=]* [1]* [(F X)])`. The expressions `(= A 1)` and `(= 1 A)` are also equated, though they are not explicitly covered by the grammar. The special analysis procedure still applies during the generator computation, and both of those expressions have the generator `(:EXPRESSION ([=]* [1]* (:EXPRESSION A NIL)) 1)`.

5.5.2. *Adjusting to the second equality*

The addition of `(= X Y)` causes several readjustments in the grammar. As Figure 5 shows, the nonterminal `[X]` is replaced with `[Y]`. This replacement causes the classes represented by `[(F X)]` and `[(F Y)]` to merge. For purposes of illustration, the merging is carried out by replacing `[(F Y)]` with `[(F X)]` instead of doing the replacement in the other direction. (In an unrestricted case like this, XRup chooses

```
[X] ==> X
[F] ==> F
[(F X)] ==> ([F] [X])
[(= (F X) 1)] ==> ([=]* [1]* [(F X)])

[Y] ==> Y
[(F Y)] ==> ([F] [Y])
[(= (F Y) 1)] ==> ([=]* [1]* [(F Y)])

[Z] ==> Z
[(F Z)] ==> ([F] [Z])
[(= (F Z) 2)] ==> ([=]* [2]* [(F Z)])

[TRUE]* ==> ([OR]* [(= (F X) 1)]
                   [(= (F Y) 1)]
                   [(= (F Z) 2)])
                ; (OR (= (F X) 1)
                      (= (F Y) 1)
                      (= (F Z) 2))
```

Figure 4. This grammar incorporates only the first of the four premises discussed in section 5.5.

the direction of replacement by replacing the symbol that occurs on the right-hand side of fewer rules.) The occurrence of X in [(F X)] is meaningless to XRup, and that X does not become obsolete when [X] is replaced by [Y].

The replacement of [(F Y)] with [(F X)] triggers the recursive merging of two more classes; [(= (F X) 1)] is replaced with [(= (F Y) 1)]. The new mechanisms connected with OR come into play during this replacement. The right-hand side of the rule

```
[TRUE]* ==> ([OR]* [(= (F X) 1)]
                   [(= (F Y) 1)]
                   [(= (F Z) 2)])
                ; (OR (= (F X) 1)
                      (= (F Y) 1)
                      (= (F Z) 2))
```

should apparently be adjusted to read

```
[F] ==> F
[(F X)] ==> ([F] [Y])    ; (= X Y)

[Y] ==> Y
     | X                 ; (= X Y)
[(= (F Y) 1)] ==> ([=]* [1]* [(F X)])
                         ; (= X Y)

[Z] ==> Z
[(F Z)] ==> ([F] [Z])
[(= (F Z) 2)] ==> ([=]* [2]* [(F Z)])

[TRUE]* ==> ([OR]* [(= (F Y) 1)]
                   [(= (F Z) 2)])
                ; (OR (= (F X) 1)
                      (= (F Y) 1)
                      (= (F Z) 2)),
                  (= X Y)
```

Figure 5. This grammar results from the grammar of Figure 4 when the premise `(= X Y)` *is added. The three rules starting with* `[Z]` *are unchanged.*

```
([OR]* [(= (F Y) 1)]
       [(= (F Y) 1)]
       [(= (F Z) 2)]).
```

At this point, however, `[OR]*` standardizes the new subexpression generator by removing the redundant class symbol. Consequently, the adjusted rule reads as follows:

```
[TRUE]* ==> ([OR]* [(= (F Y) 1)] [(= (F Z) 2)])
                ; (OR (= (F X) 1)
                      (= (F Y) 1)
                      (= (F Z) 2)),
                  (= X Y)
```

The standardization process does not need to augment the rule justification because it expresses a logical truth.

```
[F] ==> F
[(F X)] ==> ([F] [Y])    ; (= X Y)

[Y] ==> Y
    | X                  ; (= X Y)
[(= (F Y) 1)] ==> ([=]* [1]* [(F X)])
                         ; (= X Y)

[Z] ==> Z
[(F Z)] ==> ([F] [Z])
[(= (F Z) 2)] ==> ([=]* [2]* [(F Z)])

[TRUE]* ==> ([OR]* [(= (F Y) 1)]
                   [(= (F Z) 2)])
                ; (OR (= (F X) 1)
                      (= (F Y) 1)
                      (= (F Z) 2)),
                  (= X Y)
          | ([OR]* [(NOT PHI)] [(= (F Z) 3)])
                ; (:-> PHI (= (F Z) 3))

[PHI] ==> PHI
[(NOT PHI)] ==> ([NOT]* [PHI])
[(= (F Z) 3)] ==> ([=]* [3]* [(F Z)])
```

Figure 6. The grammar discussed in section 5.5 reaches this form after every premise except PHI *has been assimilated. The upper portion of the grammar is unchanged from Figure 5.*

5.5.3. Adding an implication

Figure 6 shows the grammar after the addition of `(:-> PHI (= (F Z) 3))`, which is the next premise. The premise is first internally translated into the form `(OR (NOT PHI) (= (F Z) 3))`. A further translation eventually produces the expression `(OR (= PHI FALSE) (= (F Z) 3))`. The special generator analysis procedure for `[=]*` then standardizes `([=]* [PHI] [FALSE]*)` back into `([NOT]* [PHI])`, which then becomes the right-hand side of a grammar rule.

5.5.4. Triggering conditional conclusions with the final premise

The final premise is assimilated by equating `PHI` to `TRUE`. Figure 7 shows

```
[F] ==> F
[(F X)] ==> ([F] [Y])    ; (= X Y)

[Y] ==> Y
      | X                ; (= X Y)
[(= (F Y) 1)] ==> ([=]* [1]* [(F X)])
                         ; (= X Y)

[Z] ==> Z
[(F Z)] ==> ([F] [Z])
[(= (F Z) 2)] ==> ([=]* [2]* [(F Z)])

[TRUE]* ==> ([OR]* [(= (F Y) 1)]
                   [(= (F Z) 2)])
                ; (OR (= (F X) 1)
                      (= (F Y) 1)
                      (= (F Z) 2)),
                  (= X Y)
            | PHI ; PHI
            | ([=]* [3]* [(F Z)])
                ; PHI, (:-> PHI (= (F Z) 3))


Queue of delayed merging actions:
   Merge [(F Z)], [3]* with justifications PHI,
      (:-> PHI (= (F Z) 3)).
```

Figure 7. The grammar of Figure 6 reaches this form when the symbol `[PHI]` *has been replaced with* `[TRUE]*`*, but the delayed merging operations listed in the queue have not been performed.*

the grammar as it appears when the main part of `[PHI]*` has been replaced with `[TRUE]*`, but enqueued actions have not been performed.

The replacement of `[PHI]` with `[TRUE]*` makes it necessary to adjust the rule `[(NOT PHI)] ==> ([NOT]* [PHI])`. Since the subexpression generator `([NOT]* [TRUE]*)` has `[FALSE]*` as logically forced generator, `[(NOT PHI)]` is replaced with `[FALSE]*`. This replacement then makes it necessary to adjust the rule

```
[TRUE]* ==> ([OR]* [(NOT PHI)]
                   [(= (F Z) 3)])
                ; (:-> PHI (= (F Z) 3))
```

to use [FALSE]* instead of [(NOT PHI)]. The new version of the disjunction contains represents only one remaining active possibility, and [OR]* picks that possibility out as a logically forced generator. Consequently, [(= (F Z) 3)] is recursively replaced with [TRUE]*. The premises PHI and (:-> PHI (= (F Z) 3)) are listed to justify this replacement.

The replacement of [(= (F Z) 3)] with [TRUE]* activates a constant generator noticer when the rule

```
[TRUE]* ==> ([=]* [3]* [(F Z)])
              ; PHI, (:-> PHI (= (F Z) 3))
```

is added to the grammar. As a result, the delayed merging of [(F Z)] and [3]* is entered in the queue of actions to be done when premise assimilation is otherwise finished. The premises PHI and (:-> PHI (= (F Z) 3)) again justify the merging.

5.5.5. Finishing up by emptying queues

When the main part of the assimilation of premise PHI is complete, it is time to empty the queue of pending actions. The only pending action is the delayed merging of [(F Z)] and [3]*. The replacement of [(F X)] with [3]* makes it necessary to adjust the right-hand side of the rule

```
[(= (F Z) 2)] ==> ([=]* [2]* [(F Z)])
                    ; PHI, (:-> PHI (= (F Z) 3)).
```

Because the new version ([=]* [2]* [3]*) has [FALSE]* as logically forced generator, [(= (F Z) 2)] is then replaced with [FALSE]*. This replacement in turn entails the adjustment of the rule

```
[TRUE]* ==> ([OR]* [(= (F Y) 1)] [(= (F Z) 2)])
              ; (OR (= (F X) 1)
                    (= (F Y) 1)
                    (= (F Z) 2)),
                (= X Y).
```

```
[F] ==> F
[(F X)] ==> ([F] [Y])    ; (= X Y)

[Y] ==> Y
    | X                  ; (= X Y)

[Z] ==> Z
[3]* ==> ([F] [Z])       ; PHI, (:-> PHI (= (F Z) 3))

[TRUE]* ==> PHI ; PHI
         | ([=]* [1]* [(F X)])
               ; (= X Y), PHI, (:-> PHI (= (F Z) 3)),
                 (OR (= (F X) 1) (= (F Y) 1) (= (F Z) 2))

Queue of delayed merging actions:
   Merge [(F X)], [1]* with justifications PHI,
       (:-> PHI (= (F Z) 3)), (= X Y),
       (OR (= (F X) 1) (= (F Y) 1) (= (F Z) 2))
```

Figure 8. This grammar results from the grammar shown in Figure 7 when the enqueued merging of [(F Z)] *and* [3]* *is carried out. The merging operation causes another delayed merging operation to be placed in the queue, and at this point the other action has not been performed.*

The adjusted right-hand side of this rule, the subexpression generator ([OR]* [(= (F Y) 1)] [FALSE]*), again contains only one active disjunct. As a result, [(= (F Y) 1)] is recursively replaced with [TRUE]*. Figure 8 shows the grammar as it appears when the replacement of [(F Z)] with [3]* is complete. More deductions remain, however, since the replacement of [(= (F Y) 1)] with [TRUE]* has added another entry to the queue. Figure 9 shows the final grammar, derived by emptying the queues completely.

Note that the grammar in Figure 9 is smaller than the grammar in Figure 4. The addition of new information has caused the grammar to shrink rather than expand. In contrast, many internal data structures are now larger. The forwarding table has grown, and the system implementation in Chapters 7 and 8 retains the structures that represent intermediate states of the grammar. Despite these other enlargements, the shrinkage of the actual grammar is beneficial. Only the actual grammar needs to be adjusted when future premises are incorporated, and consequently the shrinkage speeds up future grammar readjustments. (See section 2.4 for an example that is related to this effect.)

```
[F] ==> F
[1]* ==> ([F] [Y])        ; (= X Y), PHI,
                            (:-> PHI (= (F Z) 3)),
                            (OR (= (F X) 1)
                                (= (F Y) 1)
                                (= (F Z) 2))

[Y] ==> Y
     |  X                 ; (= X Y)

[Z] ==> Z
[3]* ==> ([F] [Z])        ; PHI, (:-> PHI (= (F Z) 3))

[TRUE]* ==> PHI           ; PHI
```

Figure 9. The grammar discussed in section 5.5 takes this form after all premises have been fully assimilated. Note that the final form of the grammar is smaller than the intermediate forms.

6. Maintaining Multiple Contexts

Chapters 4 and 5 have explained how XRup can derive and represent the consequences of a single set of premises. This chapter describes how the system can switch quickly between different sets of assumptions. The next chapters will tell how to implement the mechanisms that have been described.

XRup remembers the consequences of different assumption sets by using a separate data structure called a *context* for each active assumption set. The context system is founded on a distinction between permanent facts and temporary assumptions, and the first section of this chapter discusses that distinction. The second section sketches the structure of the context system. Another section describes the use of contexts to do reasoning by contradiction. The remaining sections explain how contexts are created, maintained, and associated with sets of assumptions.

6.1. Permanent Facts, Temporary Assumptions, and Retraction

The XRup system distinguishes *permanent facts* from *temporary assumptions.* Permanent facts are considered to be part of all sets of assumptions. They cannot be retracted or assumed to be false. In contrast, temporary assumptions characterize assumption contexts and differentiate among them. Given a set S of temporary assumptions, XRup can create a context C_S that incorporates all permanent facts plus the assumptions in S. C_S contains no other premises, and once C_S has been created, no assumptions can be added to it or removed from it. If the set of assumptions S is considered again before C_S is discarded, and if C_S has been kept current by adding new permanent facts to it, then C_S can be reused without reconstruction.

6.1.1. Replacing retraction

As section 4.1 has explained, XRup uses the context mechanism to replace retraction. With a deductive system that allows retraction, an expert system might add retractable assumptions A and B to its knowledge base, then perhaps later remove A and its derived consequences. With XRup, the expert system could instead direct attention to a context that incorporates the current set of assumptions plus A and B. Instead of later retracting A, the system would redirect its attention to a context incorporating all of those assumptions except A.

6.1.2. The value of the context approach

XRup will not work well if the "working set" of active assumption contexts is too large. Although efforts are made to keep contexts small, they require substantial memory space. In addition, the implemented method of mapping assumption sets to stored contexts works poorly when there are many active contexts or large numbers of possible assumptions. Consequently, XRup is best suited to cases in which an expert system switches repeatedly among a small number of different assumption sets.

Because of these restrictions, the context mechanism is not a complete replacement for retraction. It provides no good way to treat a case in which most of a program's beliefs are adopted provisionally. Presumably such a program would have sophisticated procedures for maintaining its "web of belief" (Quine and Ullian, 1978; Doyle and London, 1980) in the face of derived contradictions; it could potentially select any provisional belief for retraction.

However, when the distinction between a large number of permanent facts and a smaller number of temporary assumptions can be maintained, the context system in XRup supports fast hypothetical reasoning. (It was originally developed for use in such a situation; see the appendix.) XRup can switch quickly between different sets of assumptions because it remembers the consequences of several sets at once. In contrast, the process of repeatedly adding and removing assumptions and their consequences can take much computation in a single-context system.

6.2. The Context System

The contexts that represent different sets of assumptions in XRup are linked into a system that coordinates access to the contexts. The main component of a context is a grammar. While operations that involve individual grammars are performed by particular contexts, operations of wider scope are performed by the context system instead.

XRup is implemented with the object-oriented programming techniques provided by the Lisp Machine flavor system (Weinreb, Moon, and Stallman, 1983). In order to ask XRup to test equality between two expressions, a program sends a message to the relevant context. In order to obtain a context that corresponds to a set of assumptions, a program sends a message to the XRup system itself, which is implemented as an object and contains contexts as parts. (This kind of message passing does not involve multiple processors or parallellism, but is merely a form of keyword-oriented procedure call.)

The XRup system always contains one distinguished context called the *global context*. The global context contains all permanent facts but no assumptions. (For this reason, permanent facts will sometimes be called *global premises*.) An assumption context is characterized by the set of local assumptions that it contains in addition to global premises.

The XRup system maintains an index that maps from sets of assumptions to the corresponding assumption contexts. Because the set of local assumptions associated with a context does not change, this index can be used to avoid reconstructing contexts when sets of assumptions are considered repeatedly. (The actual indexing procedure is more complex than this paragraph suggests because it attempts to use the same context for all globally equivalent variants of an assumption set. See section 6.5.)

6.3. Creating New Assumption Contexts

An assumption context contains every global premise and several additional assumptions. Building the grammars of assumption contexts from scratch would be unnecessarily slow because the consequences of permanent facts would be recomputed each time an assumption context was constructed. It is much faster to copy the grammar of the global context and then add the temporary assumptions that characterize the new context. It is even better to locate an existing context that incorporates the correct set of assumptions, when such a context exists. (It is not sufficient simply for all of the desired assumptions to be *true* in an existing context; the context for {`(= X Y)`} equates `(F X)` and `(F Y)`, but encodes many consequences that should not be true in the assumption context for the weaker assumption `(= (F X) (F Y))`.)

6.3.1. Creating new contexts from copies of old ones

The XRup system keeps a list that records all existing contexts and their associated assumptions. When it receives a request for a context that embodies a given set of assumptions, it first looks for an existing context that matches exactly. If it finds an exact match, it returns the existing context.

If no context matches exactly, the system chooses a source context whose assumptions form a subset of the desired assumption set. It copies the source context into a new context and then adds to the copy each assumption that is not already present. It is always possible to locate an appropriate source context because the global context can be used when nothing better is available. The assumption set of the global context is empty; hence it is always a subset of the desired assumption set.

Every permanent fact is true in every context. As a result, the source context that is copied during the creation of a new context already includes permanent facts and their derived consequences. It is never necessary to recompute the consequences of permanent facts; permanent facts are obtained "free" during copying.

6.3.2. Choosing a source context

When no existing context matches the desired assumption set and a new context must be created, several existing contexts may correspond to assumption sets that are subsets of the desired assumption set. It is difficult to tell which one should be copied. One strategy is to choose the one that contains the largest subset of the desired assumption set. This strategy may not work well because it ignores the fact that some assumptions entail more consequences than others. It is desirable to choose a source context that already includes the assumptions that have many consequences. The problem is complicated by the fact that the cost of adding premises is not a linear function of the premises involved. Two inexpensive premises in combination may entail large numbers of consequences.

The current implementation of the reasoning system does not have a completely satisfactory method of choosing among possible source contexts. It attempts to take advantage of as much previous work as possible by picking the source context that required the most computation to create. As currently implemented, the method has several disadvantages.

For example, if A has many consequences and B has few, it makes sense to prefer the context of $\{A\}$ over the context of $\{B\}$ when considering $\{A, B\}$. However, the success of this maneuver causes $\{A, B\}$ to appear inexpensive so that $\{A\}$ will be improperly preferred over $\{A, B\}$ as a source for $\{A, B, C\}$. Although it would not be difficult to fix that particular defect in the cost metric, the existing mechanism would remain crude. The nonlinear nature of premise combination and the complicating effect of global facts make it hard for simple numeric methods to estimate the cost of adding assumptions to a prospective source context. In addition, a good overall context management strategy might take into account other factors than immediate cost. For instance, if a reasoning system is going to consider the assumption sets $\{A, B\}$, $\{A, C\}$, and $\{A, D\}$, an omniscient context management system might save time by saving an intermediate context with assumption set $\{A\}$.

6.3.3. *Physical copying of the source context*

The copying operation that XRup uses to create new contexts is a physical copy. The system does not attempt to save space by using a "virtual copy" scheme (Fahlman, 1979). This decision was made for several reasons. The use of virtual copies would complicate and slow down retrieval algorithms. It would also complicate the assimilation of new permanent facts into the grammar of the source context, since modification of the source context would need to preserve the relationship between that context and related copies. Because new assumptions often yield new consequences, the differences between the source and destination contexts are not confined to the structures that represent the new assumptions; as a result, virtual copies might not save much.

Finally, virtual copies are not necessary because physical copies seem to work. The data structures that contexts use to represent grammars have been designed with the aim of making the copying operation fast. The copying of a context consists largely of copying a set of one-dimensional arrays, and the microcode primitives of the current target machine make such operations fast. In large examples, copying time has been observed to be only a small fraction of the total time spent in the creation of new contexts. Deducing the consequences of new assumptions is a much more significant task.

6.4. Reasoning by Contradiction

One important use of the context system increases the deductive power of XRup through an operation that is called `:try-to-show` in the implementation.* The `:try-to-show` context operation tries to prove that a given statement must be true (or false) in the context even though the grammar does not currently indicate that fact. (The operation is computationally expensive.)

To try to show φ, a context creates a copy of itself and adds the assumption (NOT φ). It then discards the copy and determines whether the addition of the assumption to the copy has caused φ to be marked true in the original context. Sections 4.2 and 4.11.2 have described the information that XRup propagates to all contexts when a contradiction is discovered. If the assumption (NOT φ) leads to a contradiction when combined with the premises in the original context, this propagated information will result in the deduction that φ is true in the original context.

Consider the example from section 4.2. In that example, the following facts are known, but the the grammar does not yet indicate the consequence that `(= X Y)` must be false:

```
(= (home-city X) Boston)
(= (home-city Y) San-Francisco)
(not (= Boston San-Francisco))
```

Let C be a context that contains those premises about X and Y. The `:try-to-show` operation can be used to derive the desired conclusion in C.

When `:try-to-show` assimilates the assumption `(= X Y)` into a copy of C, XRup attempts to derive the contradicted conclusion `(= Boston San-Francisco)`.

*Rup has a similar TRY-TO-SHOW operation.

The discovery of the contradiction causes XRup to discard the new assumption context and add the following logical truth to every context, including C:

```
(OR (NOT (= X Y))
    (NOT (= (home-city X) Boston))
    (NOT (= (home-city Y) San-Francisco))
    (= Boston San-Francisco))
```

Every disjunct except the first is false in C. Consequently, the assimilation of the disjunction will cause `(NOT (= X Y))` to be marked as true and `(= X Y)` to be marked as false in C. Correct justifications are maintained because the normal assimilation procedures are used; there is no special grammar-modification procedure for reasoning by contradiction.

If a contradiction results when `(NOT` φ`)` is added to a copy of a context C, it should always come to be marked in C that φ is true. Let C' be the copy of C. A logically valid disjunction will be assimilated into all contexts when the contradiction in C' is discovered. The disjunction is built from the negations of premises that are true in C'. Since C' starts out as a copy of C, every disjunct must either be the statement φ, or be false in C. Hence the only disjunct that could possibly be true in C is φ. Also, `(NOT` φ`)` should be an underlying premise of the derived contradiction, since the addition of `(NOT` φ`)` was necessary to bring the contradiction to light. Then the disjunction should contain φ as a disjunct, and the assimilation of the disjunction should cause φ to be marked as true in C.

6.5. Identifying Equivalent Sets of Assumptions

As the previous section relates, XRup saves time and space by making use of previously constructed contexts when their associated assumptions reappear. If the system has previously constructed and retained a context that embodies the assumption set {`(= (F X) 1)`}, it need not reconstruct that context when that assumption set is considered again. In fact, it is possible to do better than this. If `(= X Y)` is a global premise, the assumption set {`(= (F X) 1)`} has the same consequences as the assumption set {`(= (F Y) 1)`}. The two assumptions are equivalent because they are made equal by permanent facts.

6.5.1. Indexing by assumption equivalence class

XRup does not index contexts by the actual assumptions that they contain. Instead, it indexes by the global equivalence classes of assumptions. This allows the system to take advantage of the kind of sharing that the above example illustrates. The assumptions `(= (F X) 1)` and `(= (F Y) 1)` fall into the same global equivalence class when `(= X Y)` is a global premise.

With this indexing procedure, the system will sometimes substitute globally equivalent assumptions for the desired assumptions when locating an assumption context. As a result, the underlying premises of derived conclusions may include assumptions that were not mentioned when the context was requested. The actual assumptions will be provably equivalent to the desired assumptions, however, and the equivalence proof will involve only permanently asserted facts.

This indexing procedure can be conveniently implemented by using grammar symbols to represent the assumptions that characterize a context. In the above example, the global grammar might assign the generator `[(= (F X) 1)]` to both of the expressions `(= (F X) 1)` and `(= (F Y) 1)`. The context created for the assumption set {`(= (F X) 1)`} would then be indexed by the assumption generator set {`[(= (F X) 1)]`}. The assumption set {`(= (F Y) 1)`} would also correspond to that assumption generator set, and the same context would be retrieved for the representation of both sets of assumptions.

6.5.2. Adjusting the assumption index

Naturally, the index must be adjusted when new global premises cause the generators of assumptions to change. It is difficult to tell when the generator of an expression changes, if the generator is an `:EXPRESSION` form. Fortunately, however, generator changes are easy to detect when the generator concerned is an explicit nonterminal. In that case, the generator cannot change unless the nonterminal is replaced by another. (This is not true of contexts that can be deactivated as in section 6.7, but the global context is permanent.)

The new generator can then be easily recovered by consulting the forwarding table in the global context. The context assumption generator index can be kept accurate by monitoring changes in the global grammar and replacing symbols in the index when they are replaced in the grammar. In order for this procedure to work, the global grammar must be expanded to give explicit representation to the equivalence class of each assumption.

These methods effectively allow XRup to standardize assumption sets on the basis of global equalities. Still more standardization of assumption sets is possible. Even if `(= X Y)` is not a global premise, the assumption sets { `(= X Y)`, `(= (F X) 1)` } and { `(= X Y)`, `(= (F Y) 1)` } are equivalent. The current implementation of XRup does not attempt to exploit this more complicated equivalence.

6.6. Adding New Permanent Facts

When new permanent facts are added to the system, they must be assimilated into the grammars of all active contexts. This action is necessary in order to preserve the integrity of assumption contexts. (An assumption context is defined to encode the consequences of all permanent facts, combined with additional assumptions.)

In order to assimilate a new permanent fact, XRup first determines whether the fact is already true in the global context. If it is already equated to `TRUE` by the global grammar, there is no need to do anything more. Since all contexts include all permanent facts and the new fact already follows from permanent facts, the new fact must already be true in every context.

XRup also checks to make sure that the new permanent fact is not already equated to `FALSE`. It signals an error if the proposed new fact is already false. Since XRup does not implement retraction of permanent facts, no recovery is possible if the global context is allowed to become inconsistent.

(Section 6.5.2 described adjustment to the assumption-context indexing mechanism that must also be performed when new permanent facts are added.)

6.7. Deactivating and Reconstituting Contexts

Since a new permanent fact must be individually assimilated into each existing context, the addition of new permanent facts becomes difficult when there are many active contexts. Consequently, it is desirable to have some way of pruning the set of active contexts. The current implementation of XRup will discard a context if it discovers that it has spent more time adding new permanent assertions to the context than it spent creating it. (In addition, it always discards inconsistent contexts.) The selective deactivation of contexts speeds up the addition of new permanent facts, but it obviously slows down references to discarded contexts. The deactivation of contexts also reduces system storage requirements; XRup reuses the storage occupied by discarded contexts and is hence able to use less.

The system is free to discard most contexts at will.* An assumption context is completely characterized by the assumptions that it contains; as a result, a context can always be reconstituted by using the normal means of creating an assumption context. (An exception is the global context, which encodes unique information and must never be discarded.)

6.7.1. *Transparent context deactivation*

In order to make the deactivation of contexts transparent to most operations, XRup does not actually return a context when a context is requested. Instead, it returns a small object that always contains a defining set of assumptions, and may contain a context that is the "current incarnation" of the set of assumptions. (In the implementation, this small object is called a regenerator.)

When it contains a context, the regenerator simply passes messages along to the context. When a context is deactivated, however, it is removed from the corresponding regenerator. The next attempt (if any) to refer to the context through the regenerator will cause the regenerator to reconstruct an incarnation of its set of assumptions. (If the context was deactivated because of inconsistency, however, most attempts to refer to the context through the regenerator will signal errors.)

Some properties of contexts and grammars are not preserved over deactivation and reincarnation. For example, when a context is deactivated and reconstituted,

*P. Szolovits has suggested an intermediate state for contexts, in which new permanent facts are noted but not assimilated. Assimilation could wait until the next use of the context, and thus effort could be saved if the context was never needed again. This approach has not been investigated. Note, however, that the storage for a context in the intermediate state could not be recycled. Note also that when a context is deactivated and the set of assumptions is later reincarnated, the permanent facts that have been added in the intervening time come into the new context "free" when it is copied from a source context.

the generators of some expressions can be different from what they would have been if the context had not been deactivated. The same equivalence classes will be generated, but different symbols may generate them.

Differences can arise because a different source context may be copied on the two occasions of creating a context to embody the relevant assumptions. Even if the same source context is copied on both occasions, differences can exist because the addition of new permanent facts has changed the grammar in the source context. Many different grammars can encode the same set of assumptions; the particular grammar that XRup constructs depends on the order of premise addition and the directions chosen for unconstrained nonterminal replacements.

6.7.2. Choosing contexts to deactivate

The method that XRup currently uses to select contexts for deactivation is not particularly sophisticated. It depends only upon the ratio between the amount of time that XRup has spent adding new permanent facts to the context and the time that it took to create the context. The justifying notion is that a context that looks cheaper to recreate than to maintain is more of a liability than an asset, and ought to be thrown away. This notion leaves room for considerable improvement, since it ignores the fact that an active context is available for copying and access, while a deactivated context is not.

7. Basic Implementation Techniques

This chapter describes the implementation of the context system. It distinguishes information that is common to the entire system from information that differs from context to context. It explains the basic methods that are used to make per-context storage compact and easy to copy while still permitting simple retrieval methods. Chapter 8 will modify and refine the techniques described here, but the fundamental ingredients will remain. The implementation will be sketched in simplified form rather than pictured in complete detail.

7.1. Separating System-Wide and Per-Context Information

In principle, a multiple-context reasoning system could just use simple algorithms to coordinate several copies of a single-context reasoning system. In practice, however, that approach leads to needless duplication of information that is common to all contexts. The first step in implementing the XRup system is to save storage by sharing system-wide structures among different contexts. This section distinguishes system-wide information from information that must be stored in each context.

7.1.1. System-wide and per-context data structures

The design of the XRup system makes most complex structures common to the entire system. It stores only the relationships between those structures in individual contexts. For example, the structures that represent the left-hand and right-hand sides of grammar rules are shared by all contexts, but each context stores its own version of the associations that link them into rules.

In many cases, these relationships must be stored in encoded form, if contexts are not to occupy prohibitively large amounts of storage. When encoding is necessary, structures that describe the encodings form another class of structures that are shared among all contexts. The structures that support system-wide functions, such as the mapping from assumptions to contexts, are also stored by the system itself rather than particular contexts.

7.1.2. System-wide structures

The major structures that are shared among contexts in XRup represent grammar nonterminals, the right-hand sides of grammar rules, set encodings, and expressions. Because these structures are shared, different contexts do not store duplicate copies of information that is associated with the structures themselves rather than their roles in different grammars. In addition to eliminating duplicates of the structures themselves, the system-wide sharing eliminates the need for duplicate copies of the hash tables and other structures that facilitate access to the structures.

For example, consider the shared structure that represents a grammar nonterminal. The system-wide information stored in the structure includes the intrinsic term of the nonterminal, its precomputed predicate level, and the special attribute bits that record whether it represents a constant or has a predicate level. It describes the set encoding that is used in storing the sets of rule right-hand sides that are associated with the nonterminal in various grammars. Finally, it includes the unique array index that is assigned to the nonterminal for references to per-context storage. As section 7.4.1 explains, the array index is used for access to the data array in each context that holds the per-context information associated with the nonterminal.

7.1.3. Per-context structures

It is desirable for per-context storage to be compact, easy to copy, and easy to access. Most of the information that is stored in a context encodes the grammar and forwarding table of the context. Arrays are used to store this information. An array position is globally assigned to each system-wide object that requires per-context storage. The information associated with a system-wide object is stored at the same position in the data arrays of different contexts. The use of simple arrays to implement per-context storage makes copying and access simple.

Per-context information must be stored in forms that are convenient for grammar-manipulation algorithms. For example, those algorithms require the ability to follow grammar rules in either direction. The representation of the grammar must support both the single-valued mapping from a rule right-hand side to the corresponding left-hand side, and the inverse mapping that yields a set of right-hand sides.

The sets of rule right-hand sides associated with grammar nonterminals can be large. It would take large amounts of space to store these and other per-context sets as simple lists of set members. XRup uses an encoded set representation that often saves space in a system with many contexts. As section 7.3.2 explains, this set encoding allows a form of sharing among contexts even when information varies somewhat from context to context. Only the system-wide set encoding lists the actual members that a set may have; the small, encoded per-context representation uses a bit vector to say which possible members are actually present.

7.2. Representing Expressions With Unique Structures

One kind of data structure that XRup shares among contexts is used to represent expressions. The system maps all occurrences of the same expression into the same internal data structure, which is called a *term*. The term structure that corresponds to an expression provides a convenient place to store information that is globally associated with the expression. The use of a single structure to represent all occurrences of a single expression also has other implementation advantages; for example, it simplifies the operation of internal hash tables.

In an object-oriented system like the Lisp Machine flavor system with which XRup is implemented, the terms that represent different expressions are all instances of the same class of objects. Each instance contains local storage and a pointer to information about the class definition. An operation on an instance is performed by sending a message to the instance, which consults the class definition to find out how to respond. (Message dispatching is a fast, microcoded operation.)

The term object class handles many messages that are specific to terms. For example, terms respond to messages about such term properties as subterms, existing superterms, and special attributes. The special attributes of terms distinguish those that are constants, are commutative operators, or are special in other ways. Each term has a serial number, and the arbitrary, fixed ordering provided by the serial numbers is sometimes used to sort term sequences into a standard order.

The term object class also gives a term-specific interpretation to many messages that can be sent to objects other than terms. For example, the term object class responds to messages about printing and pretty-printing (Waters, 1981), thus causing terms to be printed differently from other objects. Since a single message can be interpreted differently by different kinds of objects, it is usually unnecessary for programs to explicitly dispatch on object type. For example, section 8.5 describes a mechanism for abbreviating justification sets. Although different kinds of objects serve as abbreviations in different ways, a program can expand any abbreviation without knowing its type. The program sends the object a message asking it to expand itself, and each kind of object that can serve as an abbreviation will respond to the message in its own manner.

Unlike many objects that are shared among contexts, terms do not have per-context storage directly associated with them. However, section 8.7 will describe a system refinement that introduces a per-context cache that remembers the generators of a limited number of terms. (Direct per-context storage of information associated with terms has been intentionally avoided in the design of XRup, because many applications involve the consideration of very large numbers of terms.)

7.3. Representing Sets Compactly

This section describes the encoding method that XRup uses to reduce the size of per-context representations of sets. The encoding technique is used when per-context information maps system-wide structures to sets of other system-wide structures. For example, the kind of set encoding described here is used to store the set of rule right-hand sides that a grammar associates with a given left-hand side. The encoding technique saves space and makes it easy to copy set representations.

Each set encoding is a bit-vector representation scheme based on a fixed enumeration of the members of a universal set. Because a set encoding provides a structural framework for the specification of sets, the objects that represent set encodings are known in the implementation as *set skeletons*. A given set skeleton can only encode sets that are subsets of its associated universal set. (However, the universal set can be expanded at any time without affecting previously encoded sets.)

7.3.1. Examples of set representation

Consider the following fragment of a grammar G_1, with justifications omitted:

```
[A] ==> A | ([F] [X]) | B | C
```

Now suppose the grammar in which these rules appear is copied and then the copy undergoes the replacement of `[X]` with `[Y]`. The resulting grammar G_2 will contain the following adjusted fragment:

```
[A] ==> A | ([F] [Y]) | B | C
```

Let S_1 and S_2 be the sets of rule right-hand sides that correspond to `[A]` in G_1 and G_2, respectively. S_1 and S_2 have much in common; they both contain `A`, `B`, and `C`. It is possible to take advantage of this fact by writing each of the sets as a subset of a larger universal set that is associated with the nonterminal `[A]`. The above fragments of G_1 and G_2 can then be represented by marking the rule right-hand sides that are included in each one, without listing the right-hand sides separately in each set:

```
[a] ==> a | ([f] [x]) | b | c | ([f] [y]) | ...
  G1:   1       1       1   1       0       0 ...
  G2:   1       0       1   1       1       0 ...
```

(Zero indicates that a right-hand side is absent; one indicates that the right-hand side is present.)

If this picture is now turned around so that new possible right-hand sides are added to the universal set on the left instead of the right, the bit strings that represent S_1 and S_2 can easily be interpreted as binary numbers. (New elements must be added to the universal set on the left instead of the right so that the encodings of previously encoded sets will not be changed by additions.) S_1 can be represented by the (reversed) bit string ...001111 or the number 15, while S_2 can be represented by the bit string ...011101 or the number 29.

7.3.2. Saving space with set skeletons

Set skeletons encode sets in exactly the manner illustrated above. Sets are encoded as numbers; each bit position in the numeric representation of a set corresponds to one element of the underlying universal set. If the elements of the universal set are numbered from zero, the ith element of the universal set corresponds to the bit position with weight 2^i.

The use of this representation can save space in a system with multiple contexts, if sets in different contexts overlap greatly in membership. Consider the representation of the sets $S_i, i = 1, \ldots, 10$, where

$$S_i = \{ j{:}1 \leq j \leq 10 \bigwedge j \neq i \}$$

If the elements of each S_i are listed separately, roughly 100 units of storage are used. If the above set encoding method is used, only about 20 units are required; it takes 10 units to list the elements of the universal set and 10 units to store the small integers that represent the S_i.

When there is less overlap in membership, less space will be saved. Also, the representation method will use space inefficiently if the base set grows large while individual sets remain small. For instance, suppose the base set contains 100 elements and S is a singleton set consisting only of the last element of the base set. It takes only one or two words of storage to list S explicitly, but takes several words to represent the encoded version of S, which is 2^{99}.

This set encoding has advantages beyond its space savings. The representation of sets as binary numbers makes it easy to compare sets for equality and to perform the operations of intersection, union, and set subtraction. It also makes it easy to copy the set representations in existing contexts when new contexts are being created. (Naturally, bitwise logical operations cannot be used to operate on pairs of sets that are encoded under different set encodings.)

7.3.3. Wasting space with set skeletons

As section 7.3.2 indicated, the set-skeleton representation uses space inefficiently when the sets to be represented overlap little in membership. Such a situation will come about if too *few* set skeletons are used for the encoding of a family of disparate sets.

The set-skeleton representation also uses space inefficiently when too *many* set skeletons are used. Consider a family of sets that are related and tend to have large overlap in membership. If the sets of the family are encoded on different set skeletons, space will be wasted because each member will tend to be listed in the universal sets of many set encodings.

XRup encodes uses set skeletons for encoding whenever per-context information maps system-wide structures to sets of other system-wide structures. For example, section 7.4.1 describes one way to store the per-context mapping from a rule right-hand side r to its associated rule justification set. A system-wide set skeleton K_r, associated with r, is used to encode the sets of premises that justify the mapping from r to its associated rule left-hand side in various contexts. The rule justification j that is associated with r in a particular context is stored by using K_r to encode j and then storing the resulting number in the context.

This implementation is relatively straightforward because the encoding that is used to encode data associated with r is also associated with r. As section 8.1 will explain in greater detail, however, this implementation uses space inefficiently because it uses too many set skeletons for encoding families of related sets. Consider how a new assumption context is created. An appropriate source context is copied and additional assumptions are assimilated into the copy. This often causes recoding of justification sets. If (= X Y) is one of the new premises, rules like

```
[1]* ==> ([F] [Y])        ; j
```

will be replaced, in the new context, by rules like

```
[1]* ==> ([F] [X])        ; j, (= X Y).
```

The elements of j will then be appear in the universal sets of the set skeletons associated with both ([F] [Y]) and ([F] [X]). The elements must appear in the skeleton for ([F] [Y]) because j is encoded by that skeleton for storage in the source context. The elements must appear in the skeleton for [F] [X] because j and (= X Y) are encoded by that skeleton for storage in the new context. It would save space if both of the above rule justifications were encoded by the same set skeleton. Chapter 8 describes implementation refinements that alleviate the problem.

7.3.4. The implementation of set skeletons

As implemented in XRup, a set skeleton uses an array to store its underlying universal set. It translates a set element into a bit position by searching the array to discover the array index of the element. If the element is not present in the array, it is added at the end. An array of fixed size is allocated when the set skeleton is created. (Some space efficiency is thus lost, but it still takes only a word or two to represent each new copy of a large set.) The array is copied to a new, larger version if it must expand beyond its initial size.

Searching the member array is usually fast because it is carried out by Lisp Machine microcode. However, if the member array of a set skeleton grows beyond a certain threshold, the set skeleton creates a hash table to speed up the search. The effects of "tuning" the values for initial member array size and hash-conversion threshold have not been investigated.

A set skeleton encodes a member set by translating the elements of the set into bit positions and setting those bit positions in its output representation. Its operation depends on the Lisp Machine's ability to store and manipulate arbitrarily large integers, since the size of the underlying universal set is not limited. In corresponding fashion, a set skeleton decodes a number into an explicit set-member list by repeatedly taking the highest unprocessed bit position of the number and retrieving the corresponding element from the member array.

One other useful operation is the sequential enumeration of a member set that may grow or shrink before the enumeration is finished. Given the location where a changing encoded member set is stored, it is possible to produce an enumerator that keeps a representation of the elements it has already enumerated. When asked to enumerate another element, the enumerator re-examines the location of the encoded set and produces an element (if any) that is a member of the currently represented set, but has not previously been processed. Such an enumerator is often useful in connection with grammar algorithms that repeat some action for as long as the grammar contains a rule of a certain sort.

7.4. Storing Grammar Rules

This section describes a way of storing a grammar and forwarding table in each context. Although this implementation was used in an early version of XRup because it is straightforward, Chapter 8 will describe another implementation that modifies the details to save space.

The rules of the grammar have the form C `==>` r ; j. The rules must be accessible in several forms. The generator algorithm requires the ability to recover C and j, given r. The premise assimilation algorithms require the additional ability to recover the set of all right-hand sides r that are associated with a given left-hand side C. They also require the ability to recover all rules whose right-hand sides contain a given nonterminal X. (The rule right-hand sides that contain X are called the *parents* of X.)

7.4.1. Per-context data arrays

Grammar storage in each context can be implemented with several arrays. Two arrays map from a rule right-hand side r to the corresponding left-hand side C and rule justification j. Two more arrays map from a nonterminal C to the corresponding sets of rule right-hand sides and parent right-hand sides. (The set of parent right-hand sides varies from context to context because right-hand sides drop out of the grammar when they are replaced by new versions.) Two final arrays map from forwarded nonterminals to the corresponding forwarding destinations and justifications.

The system-wide structure that represents a grammar nonterminal or the right-hand side of a rule contains a globally assigned index into per-context data structures. The class symbol `[(F X)]`, for instance, might be assigned the class array index 33. In that case, the encoded set of right-hand sides corresponding to `[(F X)]` in a particular context would be stored in element 33 of the rule right-hand-side array of the context. The same array index would correspond to `[(F X)]` in every context.

Class symbols and rule right-hand sides index into different per-context arrays, so the assignment of class array indices is independent of the assignment of right-hand-side array indices. Also, since at any time there are many class symbols that have never been forwarded in any context, the assignment of forwarding array indices to class symbols is made separate from the assignment of class array indices. Every class symbol has a class array index, but many classes do not have forwarding array indices.

A new class symbol receives a class array index when it is created. Similarly, a new rule right-hand side receives a new right-hand-side array index. A class does not receive a forwarding array index until it is forwarded in some context. When a new index into context data arrays is assigned, it is necessary to expand any context data arrays that are too small to include the new index.

Per-context data that are not sets are stored directly, but sets are encoded and stored as numbers. For example, consider the sets of rule right-hand sides and parent right-hand sides that correspond to a class symbol C in a particular context. The sets are encoded by a set skeleton associated with C and then stored as numbers in the rule right-hand-side array and parent right-hand-side array of the context. Similarly, the sets associated with a rule right-hand side r are encoded by a set skeleton associated with r before they are stored.

7.4.2. Related system-level operations

The XRup system itself implements several operations that are used in conjunction with per-context grammar storage. For example, the system implements the mapping from a term x to the possibly new class symbol whose intrinsic term is x. The system also implements the special generator analysis procedures and other mechanisms that go beyond the explicitly stored rules of the grammar.

The most useful versions of the grammar access procedures combine system-level operations with references to context data arrays. For example, the handler for the `:lhs-of-rhs` message to a context accepts a subexpression generator r as argument and recovers either a logically forced generator or the left-hand side of an explicitly stored rule whose right-hand side is r. It first invokes context-independent analysis to give any special operator involved a chance to specify a logically forced generator or a standardized version for r. If that analysis does not determine the generator, the handler looks for the system-wide right-hand-side structure that holds the right-hand-side array index corresponding to r. If there is no such structure, then the grammar cannot contain a stored left-hand side for r. If there is such a structure, the message handler indexes into context data arrays after retrieving the right-hand-side array index that the structure contains.

7.5. Copying Contexts

With the above representation for per-context data structures, copying the contents of one context into another is simple. After the data arrays of the destination context have been adjusted to an adequately large size, the data arrays of the source context are copied into them. The data arrays contain only numbers and pointers to system-wide objects. They do not contain pointers to mutable per-context structures like lists, which would need individual, space-consuming copying to keep the two contexts independent. (A context cannot be copied while it is in the process of being modified, since structures like the queue of pending class-merging operations are not copied.)

The Lisp Machine flavor system allows the different per-context data arrays to be managed by different modular components that implement the encoding and storage of different kinds of information. Each component arranges for its arrays to be copied, by contributing its own piece of code to be executed during context copying. (Lisp Machine system software automatically assembles the individual copying methods into an overall copying method.) This modularity has been quite valuable during system development, since the implementation and organization of the XRup system have changed many times during its history.

7.6. Mapping Assumption Sets to Contexts

The use of a set skeleton also makes it easier to map from a set of temporary assumptions to the assumption context that incorporates those assumptions. The XRup system maintains a set skeleton whose universal set contains the global generators of all terms that are temporary assumptions in any context. It also maintains a list that associates encoded sets of assumption generators with existing contexts. The encoded set representation aids in searching the list because it transforms the necessary set equality, subset, set difference, and intersection operations into simple operations on integers.

Given a set of assumptions, the system first separates conjunctive assumptions into components so that { (AND A B) } and { A, B } are treated as the same assumption set. It then computes the global generators of the assumptions, removing occurrences of the generator [TRUE]* and reporting that an assumption is contradicted if the generator [FALSE]* turns up. It checks to make sure that the conjunction of the assumption generators is not mapped to [FALSE]*, again reporting that the assumptions are contradicted if their conjunction is false. It then encodes the set of assumption generators and uses the encoded representation to carry out the procedures described in sections 6.5 and 6.3.

7.7. Deactivating and Reconstituting Contexts

This section goes slightly further into detail about the context deactivation mechanism that was described in section 6.7. Context deactivation always occurs when a context becomes inconsistent, and consistent contexts are sometimes selected for deactivation. The system uses the previously described regenerator mechanism to insulate outside programs from the effects of context deactivation. Internally, it recycles the storage of deactivated contexts.

The system implements the recycling mechanism by using a set skeleton whose universal set includes all existing context structures. It maintains encoded sets of active and deactivated contexts. (The encoded representation makes it easy to transfer contexts between these lists. In addition, the enumeration operation of the set skeleton provides a convenient way to map some operation over all active contexts, when processing one context may cause others to be activated or deactivated.) When the system needs a new context structure, it looks on the deactivated context list before allocating new storage. It adjusts the size of the arrays in the old context before copying into it. It also resets the context in other ways so that it will serve as well as a freshly allocated one.

A context can become inconsistent only during grammar update; it is not necessary to provide for this possibility during read operations. Consequently, top-level grammar alteration procedures such as premise assimilation must set up the environment for a possible inconsistency abort before they proceed. Top-level alteration procedures must also set up the queue mechanism that allows low-level procedures to specify actions that should be performed when the main part of grammar alteration is over. The necessary framework is set up by the use of a special message to the context to be altered. In essence, the message specifies an action and tells the context to perform the action with abort and queue mechanisms set up. If the alteration method is called recursively, it does not establish a nested alteration environment, but uses the one that is already set up.

If an inconsistency is detected during grammar update, the resulting abort operation abandons the nested procedure invocations that are in progress and returns immediately to the outer alteration method. The abort operation also gives the XRup system logical information about the inconsistency. The context is deactivated, and the inconsistency information is propagaged to all active contexts.

8. Refining the Implementation

This chapter refines the system implementation that was sketched in Chapter 7. Although they are more complex, the revised implementation methods make the system run faster, use less storage, and produce smaller justification sets for some conclusions. The first two sections point out deficiencies in the previous implementation. The next three sections remedy these deficiencies by changing grammar storage conventions. The last two sections describe additional refinements.

8.1. Hidden Copying of Rules

The grammar storage method sketched in Chapter 7 uses set skeletons to reduce the total amount of storage needed to represent families of related sets. That implementation is relatively straightforward because a piece of per-context information associated with an object X is either directly stored in a context data array, or encoded using a set skeleton inside X.

As experience with early XRup systems pointed out, however, that implementation is wasteful of storage and can be improved. The implementation wastes storage because it tends to cause a single set element to be duplicated in the universal sets of many different set skeletons. Although the set-skeleton representation makes it possible to copy a set without copying a list of its members, this duplication is a kind of "hidden copying" of set membership lists. Section 7.3.3 gave one example of the problem. This section further illustrates the problem, and later sections describe a cure.

8.1.1. Hidden copying of rule right-hand sides

Consider a context that incorporates the following equalities:

```
(= A X)          (= A Y)          (= A Z)
```

Assume that the context contains the following grammar:

```
[A] ==> A
      | X                 ; (= A X)
      | Y                 ; (= A Y)
      | Z                 ; (= A Z) .
```

The right-hand sides A, X, Y, and Z then occur in the set skeleton associated with [A], since that skeleton is used to encode the set of rule right-hand sides associated with [A].

Now consider the creation of an assumption context that uses the above context as its source context and, in addition to the assumptions of the source context, incorporates the following equalities:

```
(= A B)          (= B C)          (= C D)
```

One possible way of assimilating these equalities into the new context involves (among other actions) the series of symbol replacements

```
[A] ---> [B] ---> [C] ---> [D].
```

When these replacements have been carried out, the following grammar results:

```
[D] ==> D
      | C           ; (= C D)
      | B           ; (= B C), (= C D)
      | A           ; (= A B), (= B C), (= C D)
      | X           ; (= A B), (= B C), (= C D), (= A X)
      | Y           ; (= A B), (= B C), (= C D), (= A Y)
      | Z           ; (= A B), (= B C), (= C D), (= A Z).
```

The universal set of the set skeleton associated with [D] now contains the rule right-hand sides A, X, Y, and Z. But the universal set of the set skeleton associated with [A] must still contain them also, since they must remain associated with [A] in the grammar of the source context.

In fact, the universal sets of the set skeletons associated with [B] and [C] will also contain A, X, and Y as a result of intermediate states that the assumption-context grammar has passed through. Once a set skeleton has been used to encode a set element, the universal set of the set skeleton will always contain a position assigned to that element. As section 7.3.4 notes, it is possible to *add* a new element to a set skeleton without disturbing the encodings of sets that have previously been encoded by the set skeleton. However, it is not possible to *remove* an element and preserve old encodings in the same way.

As this example illustrates, the grammar-storage implementation described in Chapter 7 often causes the same rule right-hand side to be encoded in many set skeletons. This "hidden copying" of rule right-hand sides wastes space, and it is desirable to avoid it.

8.1.2. Hidden copying of rule justifications

The implementation described in Chapter 7 also causes hidden copying of justification sets. Although there is some copying of forwarding justifications, a more serious problem is the hidden copying of rule justifications. Consider the following equalities:

```
(= (F A) X)
(= A B)
(= B C)
(= C D)
```

After the first equality has been assimilated, one possible grammar contains the following rule:

```
[X] ==> ([F] [A])          ; (= (F A) X)
```

When the premise `(= A B)` is added, the classes `[A]` and `[B]` must be merged. Suppose the merging is carried out by replacing `[A]` with `[B]`. (XRup would actually choose to replace `[B]` with `[A]` in order to reduce grammar readjustment, but ignore that subtlety for this example.) As a result of that replacement, the above rule is adjusted to read as follows:

```
[X] ==> ([F] [B])          ; (= (F A) X), (= A B)
```

In the implementation that section 7.4.1 describes, the set skeletons that encode rule justifications are contained in the structures that represent the right-hand sides of rules. Consequently, the premise `(= (F A) X)` is now encoded in the set skeletons of both `([F] [A])` and `([F] [B])`. Such duplication of a single element is not serious, but consider the next step. When the premise `(= B C)` is assimilated, the above rule is again adjusted:

```
[X] ==> ([F] [C])          ; (= (F A) X), (= A B), (= B C)
```

In this step, both (= (F A) X) and (= A B) have been copied from the skeleton of ([F] [B]) to the skeleton of ([F] [C]). When the final premise (= C D) is assimilated, they will be copied again, and (= B C) will be copied also. The situation would be even worse if the original grammar contained other right-hand sides involving [A].

Rule justifications can grow large, and the structures that represent the right-hand sides of rules are numerous. It is disastrous for a large set of premises to be recoded on every iteration of the right-hand forwarding adjustment algorithm. The time and space costs of this "hidden copying" are likely to be quite large.

One key to reducing the hidden copying of rule justifications lies in the fact that the simple kind of forwarding adjustment illustrated above leaves the right-hand sides of rules unchanged. In such simple cases, there is less copying and recoding if rule justifications are encoded by the class symbol rather than the rule right-hand side. Later sections of this chapter present a revised grammar storage implementation that incorporates this change, in addition to others.

8.2. The Growth of Rule Justifications

A related problem surfaces when large systems based on XRup operate for long periods of time. The deduction mechanisms of XRup operate by combining premises to produce new conclusions. Consequently, the sets of premises that justify conclusions usually get larger as time goes on. As these large justification sets come to participate in more and more conclusions, they tend to occur in more and more places throughout the grammar.

XRup saves spaces by storing premise sets in encoded form, but it still takes some space to list the premises in the universal sets of set skeletons. It also takes time for premise assimilation and justification retrieval algorithms to decode, manipulate, and encode their elements. The problem is compounded by the hidden copying described in the previous section.

It is a design goal of XRup that the system should perform well in applications where there are large numbers of equalities. As a result, it would be especially bad if the growth of rule justifications caused the system to perform especially poorly in such applications. Fortunately, later sections explain how XRup can use abbreviated forms of justification sets in its internal operations. It saves time and space to operate with abbreviated sets instead of delving into their membership.

8.3. Using Forwarding History to Reduce Rule Copying

The first revision in the implementation of grammar storage is designed to reduce the copying of rule right-hand sides. It also plays a part in the abbreviation of justifications. The revision eliminates physical movement of rule right-hand sides during class forwarding.

8.3.1. Making access procedures follow class forwarding

Consider the left-hand replacement of nonterminal X with nonterminal Y, justified with some premise set J_{XY}. With the grammar-storage implementation that Chapter 7 described, left-hand replacement was implemented by removing each rule of the form X `==>` r ; j and replacing it with the adjusted rule Y `==>` r ; $j \cup J_{XY}$. An entry was then made in the forwarding table to record the fact that X had been replaced by Y with justification J_{XY}. Each rule switch had three effects on per-context storage. The recorded left-hand side associated with r was changed from X to Y. The sets of right-hand sides recorded for X and Y were changed in a corresponding way. The rule justification associated with r was enlarged by adding in the elements of J_{XY}.

By changing the procedures that access per-context storage, it is possible to dispense with adjustments to individual rules in left-hand replacement. The procedure that retrieves the left-hand side associated with r should follow class forwarding after retrieving the recorded left-hand side. The procedure that retrieves the rule justification associated with r should follow the same class-forwarding path, adding in forwarding justifications as it goes. When X is forwarded to Y with justification J_{XY}, these revised access procedures will effectively change each rule X `==>` r ; j into Y `==>` r ; $j \cup J_{XY}$, as desired.

This revision shifts computational effort from update to access, and the revised access procedure takes longer to execute than the old one. The old access procedure did not need to follow class forwarding after retrieving a class symbol from the left-hand-side array. However, an additional refinement described in section 8.6.4 will in most cases speed up the new procedure.

8.3.2. Adjusting other operations

A few other changes must be made. It must still be possible to retrieve the set of right-hand sides associated with Y, and that set must include right-hand sides that are physically attached to X rather than Y. The set of right-hand sides can be enumerated with a simple tree-walking procedure if an inverse of the forwarding table is provided. In addition to recording the fact that Y is the symbol to which symbol X has been forwarded, the system must record the fact that X is in the set of symbols that have been forwarded to Y.

It is convenient to use the same per-context storage for the set of classes forwarded to Y and the set of rule right-hand sides associated with Y. The set that is actually encoded and stored will then be the union of those two sets. Because class symbols and rule right-hand sides are distinguishable, the two sets can still be separated. (In fact, the system can separate them quickly, if it stores with each class symbol an encoded set that tells which skeleton elements are classes and which are rule right-hand sides.)

Consider finally the deletion of the rule C ==> r ; j, where r is actually associated with a symbol C' that has been forwarded to C. With the revised storage conventions, the deletion amounts to removing C' as the left-hand side recorded for r, and removing r from the set of right-hand sides recorded for C'. (If this action leaves C' with no recorded right-hand sides or inverse forwarded classes, C' should then be recursively removed from the inverse forwarded classes of its immediate forwarding destination. This relieves the tree-walking procedure that recovers right-hand sides from fruitlessly walking the empty tree headed by C'.)

8.4. Reorganizing Rule Justification Encodings

The next step is to remove set skeletons from the structures that represent the right-hand sides of rules. The premise assimilation algorithms require the ability to map from a rule right-hand side r to the premise set j that justifies the associated rule C ==> r ; j. In the old implementation, the set associated with r is encoded by a set skeleton associated with r and then stored in the per-context array position assigned to r.

In the new implementation, there is no set skeleton associated with r. The set skeleton K_C associated with C is used instead. The set j associated with r is encoded by K_C and then stored as before in the array position assigned to r. In order to recover the decoded form of j from r, the system must retrieve the associated left-hand side C in addition to the encoded form of j. It then uses K_C to decode the representation of j.

Since a grammar contains more right-hand sides than it contains nonterminals, this change should reduce the number of set skeletons whose universal sets contain a given justifying premise. Consequently, it should reduce the hidden copying of rule justifications. Furthermore, if combined with the other changes in this chapter, it will not cause the set skeletons of nonterminals to grow too large by encoding large justification sets.

Consider now the left-hand replacement of X with Y. If this operation still involved moving right-hand sides from X to Y, the new method of storing rule justifications would involve much recoding and hidden copying of justification sets. When the left-hand side associated with a right-hand side r changed from X to Y, it would be necessary to first use X to decode the rule justification and then use Y to re-encode it.

With the revised retrieval algorithms described in the previous section, it is not necessary to re-encode the rule justification. When X is replaced with Y, X remains internally recorded as the left-hand side of r. Although the procedures that extract the left-hand side for external use will follow forwarding from X to Y, X can still be used internally to encode and decode the rule justification associated with r.

By itself, the change from encoding j on r to encoding j on C also has another disadvantage. For a given C, there may be many rules of the form C `==>` r ; j. The different justification sets j may not have many elements in common, and their union may be large. Under such conditions, the set skeleton associated with C will provide poor set representations. The integer representing each j will contain only a few "1" bits and many "0" bits; set representations will no longer be compact. The use of abbreviated forms for justification sets can alleviate this problem; see the next section.

8.5. Abbreviating Justification Sets

The final step in revising the implementation of per-context storage is to introduce abbreviations for sets of justifications. With this revision, the internal forms of sets can contain both ordinary elements and subset abbreviations as members. When necessary, the internal forms can be converted to external forms by expanding abbreviations and removing duplicates. The use of abbreviations for justification sets enhances system performance when justification sets are large.

8.5.1. Using abbreviated sets internally

The use of abbreviations is made possible in part by the fact that grammar manipulation algorithms do not deal with justification sets in detail. Virtually the only set operation in those algorithms is set union. If a abbreviates set A and b abbreviates set B, the set $\{a, b\}$ is an easily constructed abbreviated form for $A \cup B$. Because the abbreviated form is often shorter and easier to construct than the actual set $A \cup B$, the use of the abbreviated form can save space and speed up internal operations. The actual work of computing the set union must be performed when the internal form $\{a, b\}$ is converted to external form, but internal operations are often more frequent than conversions to external form.

Ordinary, an abbreviation scheme would take extra space for storing the mapping from abbreviations to the sets they abbreviate. However, a careful choice of abbreviations lets the abbreviation scheme in XRup avoid using such extra space. The basis of the scheme is the observation that any structure that is associated with a set can be used as an abbreviation for that set, *provided* that the structure remains forever associated with the *same* set once the association has been established.

8.5.2. Abbreviations for justification sets

Under the revised grammar-storage conventions that previous sections of this chapter have described, two kinds of structures can usefully serve as abbreviations for justification sets. The structures that can serve as abbreviations represent rule right-hand sides and grammar nonterminals.

Consider the new method of storing the grammar rule C ==> r ; j. The justification set j that is internally associated with a given right-hand side r never changes after it is first stored. After C is replaced with some other class symbol, the access procedure that recovers j from r will return accumulated forwarding justifications in addition to the j that is actually stored. Because of this feature of the access procedure, j appears to change in the manner specified by grammar modification algorithms. However, the version of j that is internally recorded never changes. The rule right-hand side r can be used as an abbreviated for the original, internally stored form of j.

Forwarded grammar nonterminals can also serve as abbreviations. A nonterminal symbol can be forwarded at most once. Although the forwarding chain that originates at a forwarded symbol may get longer, the forwarding justification that is recorded for each link in the chain never changes. Once it has been replaced by another nonterminal, a grammar nonterminal can serve as an abbreviation for the justification set that is associated with the first link of its forwarding chain.

8.5.3. A space-saving coincidence

The choice of rule right-hand sides and forwarded grammar symbols to abbreviate justification sets is particularly fortunate because these structures tend to be already present for other reasons in the set skeletons of classes. If a grammar modification algorithm replaces class symbol C with C', in most cases it must first reach C by traversing a rule that has C on its left-hand side. In traversing that rule, it will accumulate the associated rule justification into the set that will be used to justify the replacement of C with C'.

If the rule right-hand side r is actually attached to C, then r must already be encoded in the set skeleton of C. That position in the set skeleton can do double duty when r later occurs (as an abbreviation) in the forwarding justification of C. If r is actually attached to some other symbol X that has been replaced with C, then X must already be encoded (for the storage of the inverse forwarding set) in the set skeleton of C. Again the position in the set skeleton will later be able to do double duty, when $\dot{X}$ occurs (as an abbreviation) in the forwarding justification of C. Thus in both of these cases, the choice of abbreviations helps keep the set skeleton of C small.

8.5.4. An example of an abbreviated justification set

The advantages of abbreviated justification sets are best shown in large examples where justification sets contain many premises. However, the same effects can be seen on a smaller scale in an augmented version of the propositional reasoning example that was presented in section 5.5.

The original example involves the following premises:

```
(OR (= (F X) 1)
    (= (F Y) 1)
    (= (F Z) 2))
(= X Y)
(:-> PHI (= (F Z) 3))
PHI
```

In the new example, the additional premise `(= (H (F X)) W)` should be assimilated before those premises. With one possible assimilation procedure, the grammar will initially contain the rule

```
[(H (F X))] ==> ([H] [(F X)]).
```

After the other premises have resulted in the eventual replacement of `[(F X)]` with `[1]*`, the grammar will contain the following rule:

```
[(H (F X))] ==> ([H] [1]*)
     ; PHI, (:-> PHI (= (F Z) 3)), (= X Y),
       (OR (= (F X) 1) (= (F Y) 1) (= (F Z) 2))
```

Abbreviation procedures have not been fully described; indeed, the details remain experimental. However, the current XRup implementation uses the abbreviated representation `[(F X)]` for the above rule justification. The abbreviation `[(F X)]` refers to the premise set that was used to justify the replacement of symbol `[(F X)]` with symbol `[1]*`, and eventually expands into the premise set shown above.

8.6. Using Global Proofs to Shorten Justifications

In the XRup system as described so far, the global context is like other contexts except for the special role that it plays in the mapping from assumptions to assumption contexts (see section 6.5). Consequently, the system considers the derivation details of globally derivable conclusions when premises are being added in the creation of assumptions contexts, just as it would if the premises were being added to the global context.

8.6.1. *Omitting the details of global derivations*

For example, suppose an assumption context is being created by copying the global context and assimilating the additional premise `(= (F X) 1)`. Suppose the global context assigns the generator `(:EXPRESSION ([F] [A]) NIL)` to the expression `(F X)`, listing the following permanent facts in the generator justification:

```
(= A B)     (= B C)     (= C D)     (= D X)
```

With the current implementation, XRup will list these permanent facts and the assumption `(= (F X) 1)` as justifications for the conclusion `(= (F A) 1)` that it draws in the assumption context. In effect, it repeats the derivation of the globally derivable conclusion `(= X A)`.

Justification sets are large when they repeat the underlying premises of globally derivable conclusions. Although the internal effects of this condition are made less severe by the internal use of abbreviations for justification sets, abbreviations must be expanded before justification sets are returned to external programs. When possible, it is desirable to shorten the external forms of justification sets also.

It is often useful to make an externally visible distinction between globally derivable conclusions and conclusions that depend in part on temporary assumptions. The *assumptions* that underlie a conclusion are often of primary interest, since only assumptions can be retracted or questioned. In such situations, proofs and justification sets can be shortened by merely *citing* conclusions that are globally derivable, rather than listing their underlying justifications. In the above example, the system can produce a simplified two-element justification set for the conclusion

(= (F A) 1), listing only the assumption (= (F X) 1) and the globally derivable conclusion (= X A).

8.6.2. Trivial justifications for global rules

The current implementation of XRup uses globally derived conclusions to shrink some justification sets in the above manner. The mechanism is best illustrated by example. In the above example, the global context contains the following rule, among others:

```
[A] ==> X ; (= A B), (= B C), (= C D), (= D X)
```

(Since the rule appears in the global context, all elements of the rule justification are permanent facts.) This rule is the source of the equalities that replicate the global derivation of (= X A) in the generator justification of (F X).

With the new mechanism, the rule still takes this form in the global context. However, the global context exhibits special behavior when it copies itself into an assumption context. The rule takes the following simplified form in the assumption context:

```
[A] ==> X                ; (= X A)
```

The simplified rule must be valid if the original rule is valid.

A trivial justification like this one can be constructed for any valid rule. Normally, such trivial justifications are useless because they do not connect derived conclusions with the externally specified premises that justify them. In this case, however, trivial justifications are useful. When underlying permanent facts in addition to underlying temporary assumptions are of interest, the global derivations of globally derived conclusions can be obtained from the global context. (The global context uses the ordinary rule justifications that consist of externally specified premises.) When only underlying temporary assumptions are of primary interest, it is easier to consider the single justification (= X A) than it is to consider the original four-element justification set.

8.6.3. A poor implementation

The above feature is somewhat tricky to implement. Consider the global grammar rule

```
[Y] ==> ([F] [A])        ; (= (F A) X), (= X Y).
```

With the grammar storage implementation sketched in section 7.4.1, the global context might represent the rule with the following implementation structures. Write "<---" for a class forwarding link:

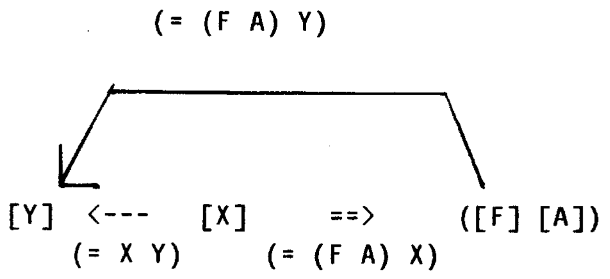
```
[Y]  <---    [X]      ==>      ([F] [A])
    (= X Y)      (= (F A) X)
```

This diagram indicates that the right-hand side ([F] [A]) is actually attached to [X], which has been forwarded to [Y].

The simplified version of this rule should be

```
[Y] ==> ([F] [A])        ; (= (F A) Y) .
```

The revised justification must replace the justification accumulated over the entire path from ([F] [A]) to ([Y]). The adjusted implementation structures should "short-circuit" the old path:



The simplest way of accomplishing the change would be to attach the right-hand side ([F] [A]) directly to [Y] and list the new justification. (Ignore for the moment the fact that this would invalidate the use of ([F] [A]) as a justification abbreviation.) Unfortunately, this implementation is deficient. It re-introduces the hidden copying of rule right-hand sides that was eliminated by the implementation changes in section 8.3. It also expands the set skeleton for [Y] by causing numerous new justifications like (= (F A) Y) to be encoded.

8.6.4. A better implementation

One better solution has two parts. The first step is to actually store the links indicated above. The "short-circuited" path from ([F] [A]) to ([Y]) is stored in a separate array from the longer path that goes through [X]. The inverse of this path, however, is not directly stored. In order to get from `([F] [A])` to `([Y])`, the system follows the short-circuited link, but in order to get from `([Y])` to the corresponding right-hand sides (including `([F] [A])`), the system follows the old inverse links.

The right-hand side `([F] [A])` does not need to be copied from `[X]` to `[Y]` in this implementation, since the inverse of the short-circuited link is not stored or encoded. With this implementation, the transformations that are carried out when the global context is copied yield an added benefit, because they reduce the number of forwarding links that must be followed in mapping from the left-hand sides to the right-hand sides of rules.

The second step is to modify the set-skeleton representation of sets. In the implementation of XRup, it is never necessary to encode sets of numbers. For this reason, there can be no confusion if a singleton set is represented directly by its single element, while any other set is represented by a number. In the transformed copy of the global grammar, the rule justification of almost every rule will be a singleton set. (Note also that transformed justifications do not contain abbreviations. The abbreviation system in the copied context starts out with a "clean slate.")

In the above example, the transformed justification set { `(= (F A) Y)` } contains only one element and is stored directly, as `(= (F A) Y)`. With the modified set representation convention, the transformed justification set does not need to be encoded on the set skeleton of `[Y]`. Consequently, the use of the simplified justifications usually will not cause set skeletons to grow large. (Some transformed justifications will eventually be encoded on set skeletons as they come to support new conclusions throughout the grammar.)

The global context does not actually transform its rules each time it copies itself. Instead, it maintains two versions of its internal data arrays. One version is used in ordinary operation, while the other version is used only when the global context copies itself into an assumption context.

8.7. Using a Cache to Speed Generator Computations

A final refinement in the implementation of contexts saves time at the expense of some storage space. If an expression is complex, it can be somewhat expensive to carry out the process of recursive descent by which its generator is computed. It can save time to maintain a cache that directly associates generators with terms. (See the appendix for information about cache performance in one application.) When the generator of a term (or common subterm) in a context has already been computed once and the generator is requested again, it will be possible to retrieve the generator from the cache without recomputing it. In the current implementation, the hash table is cleared when it reaches a fixed size limit. This policy prevents cache storage from growing to encompass the entire universe of expressions.

The generator cache is implemented as a hash table keyed by terms. In addition to the generator and generator justification of a term, a hash table entry contains two counters that indicate when the generator was last known to be correct. The counters in the table entries have counterparts in the context itself; counter values are copied from the context to the entry whenever a generator is computed and stored in the entry. Changing the counter values in the context provides a way to invalidate cached information without taking the time to actually clear the cache.

The *major cache count* in a context is incremented whenever the context is reused after deactivation. This has the effect of invalidating all cached information when context storage is reused, since the major cache count in a table entry must match the count in the context before the entry is considered valid.

The *minor cache count* in a context is incremented whenever the grammar of the context is changed. If the minor cache count of a table entry does not match the count in the context, it still may not be necessary to recompute the generator from scratch, as long as the major cache count matches. If the stored generator is a constant class symbol such as [TRUE]*, the generator cannot have changed. The cached information is still valid, and the minor cache count can be set to the current context value. If the stored generator is not a constant but is a nonterminal symbol, the generator may have changed, but only if the stored nonterminal has been replaced with another. The stored generator can be updated by following class forwarding, and it is not necessary to go through the full generator computation. If the stored generator is only an :EXPRESSION form, the generator must be recomputed.

The generator cache was relatively bug-prone during its development because the generator algorithm is recursive. If the table entry that corresponds to a term has been located, examined, and found to contain information that is no longer valid, the table may be cleared or moved during the recursive generator recomputation. For this reason, the implementation must be careful when it stores the updated generator information back into the previously located table entry.

9. Conclusions and Possible System Extensions

Previous chapters have described the structure, operation, and implementation of the existing XRup system. This chapter presents simple conclusions and briefly explores possible future directions for extending the system. Some system extensions could be easily implemented, while others would require research and system redesign.

9.1. Discussion of the Implemented System

The construction of the XRup system served several purposes. The project explored the potential of McAllester's (1982b) grammar-based representation for the consequences of equalities. As section 1.1.2 indicated, it is expensive for a reasoning system to give redundant consideration to equivalent variants of expressions, especially in an application that involves many equalities. In one experimental application (see the appendix), the manifestations of this problem were sufficiently severe with the Rup system that the system could not be used. Although McAllester's algorithms had not been implemented, his grammar-based representation promised the ability to operate on schematic representations of equivalence classes instead of considering class members individually.

9.1.1. XRup as an experimental system

The XRup system represents an experimental attempt to exploit this ability. Expression grammars were made the fundamental basis of the new reasoning system. However, the only function of the original grammar-based algorithms was to determine the consequences that follow from equalities by substitution of equals for equals. The new representation could not form the central component of a reasoning system unless it could be elaborated to provide other capabilities. It was necessary to extend the basic mechanisms so that XRup could provide supporting premises for derived conclusions, reason about constants, truth values, and logical connectives, and consider temporary assumptions. (In the extension effort, one goal was to preserve the use of a small, schematic representation that does not expand to include every new expression that the system encounters.) In addition, early implementations made it apparent that considerable attention to the time and space efficiency of implementation techniques was required.

As previous chapters have explained, the final system design succeeds in eliminating the underlying TMS, making the equality system primary, and arranging for the equality and context systems to take over the functions of the old TMS. It is indeed possible to expand McAllester's expression grammars into the basis of a Rup-like reasoning system, and the design does streamline equality reasoning by reducing the redundant consideration of equivalent variants (see Figure 2 in section 1.1). In at least one application, the new system appears to run faster than the old one (see the appendix).* In addition, the new system exhibits a better integration of propositional and equality reasoning. Rup gave no special recognition to equalities between statements and truth values. Propositional reasoning and equality reasoning were largely separate; for example, if a Rup-based program wanted true statements to simplify to TRUE, it had to use special simplication procedures that respected that fact, instead of using Rup's native simplification capabilities.† The better integration between propositional and equality reasoning also makes XRup derive more conclusions than Rup does.‡

The system is thus partially successful and deserves further investigation. However, the current system is experimental and its range of potential application has yet to be practically tested. Consequently, it is probable that the current system has many deficiencies that have not yet come to light. It is already clear that system operation should be improved in many areas. As later sections of this chapter indicate, many useful capabilities are missing. In addition, there are difficulties with

*A "production" version of the reasoning system could run faster than the current version, if internal procedure-calling, argument-passing, and data-structure conventions were altered. As previously mentioned, the current system relies heavily on the Lisp Machine flavor system. The modularity provided by the flavor system was crucial during development, but reliance on that system could be reduced if necessary. Although flavor-system overhead is not tremendously large, it pervades the current system. In addition, various internal data could be bound as "special variables" instead of being repeatedly passed as parameters.

†Although XRup implements logical equivalence as equality and recognizes equality between a statement and its truth value, it does not currently implement simplification.

‡Although the example in section 5.5 has not been run in the current Rup system, McAllester (personal communication) believes that it will fail to derive many of the conclusions that XRup derives.

the context system and with the lengths of proofs. Further research will be required before the source of these problems can be understood and perhaps eliminated.

9.1.2. Current difficulties with the context system

The context system can save much time when an expert system does repeated context-switching among a relatively small number of contexts, especially when many consequences follow from each set of assumptions. However, the system is often clumsy when few consequences follow from an assumption set, since each set of assumptions is embodied as a somewhat bulky context. In addition, as section 6.7 notes, it is difficult to add new clauses and permanent premises when the set of active contexts is large. The context system also has an unfortunate multiplicative character in some situations. When the consequences of possible assumptions factor into subsets that do not interact, it seems unfortunate that the context system will give independent treatment to all of the combinations of independent assumptions that an application program considers. A TMS has an advantage when sets of assumptions factor in this way.

9.1.3. The length of justification sets

When only equality reasoning is involved, the proofs produced by Rup and XRup often seem roughly comparable; for simple equality reasoning, they use somewhat similar algorithms. However, XRup sometimes lists longer proofs for conclusions that are derived by propositional reasoning. When given the premises `(= B C)` and `A`, XRup may produce both `(= B C)` and `A` as justifications for `(OR A B)`. In addition, if XRup is able to derive a conclusion φ, it is unable to shorten the proof of φ if it is then given φ as an additional premise. In general, the existing proof of φ will be distributed through the grammar, and there will be no way to shorten that proof to nothing without disturbing the proofs of other conclusions. (In some cases, it might be possible to remedy matters by exploiting the fact that justifications are often stored in abbreviated form. If new information made it possible to contract the expansion of the abbreviation without disturbing validity, scattered references to the abbreviated justification would be automatically contracted as well.)

9.2. Simplifying Expressions

Expression simplification was an important intended application of McAllester's original grammar-based representation for equivalence classes. The general task of McAllester's simplifier is easy to describe. Given an expression x, a set of equalities and a simplicity ordering that satisfies certain restrictions, the simplifier produces the simplest expression in the equivalence class of x. (If there is no unique simplest expression, the simplifier produces an expression such that no other expression is simpler.)

For example, suppose the symbols `F` and `C` are taken to be independent variables and other symbols are not. Suppose expressions that contain only independent variables are considered simpler than expressions that contain other symbols. Consider the three equalities `(= A (F (F (F A))))`, `(= B (F A))`, and `(= C (F B))`. Then the simplest expression in the equivalence class of `B` is `(F (F C))`. In effect, the simplifier will "solve" the equalities to determine `(F (F C))` as the "value" of `B`.

McAllester (1982b) has described simplification algorithms that are based on his expression grammars. In an earlier paper (McAllester, 1981b), he has also described incremental forms of related algorithms. Minor adaptation of McAllester's algorithms could probably give XRup the ability to simplify expressions.

As Chapters 3 and 4 have explained, however, there are differences between McAllester's expression grammars and the grammar-based framework of XRup. The framework of XRup includes special mechanisms like the logically forced generator mechanism. Because these special mechanisms are present, the grammars that XRup uses do not completely describe equivalence classes in the way that McAllester's grammars do.

This incompleteness would necessitate some adjustments in McAllester's framework and simplification algorithms. For example, the grammars that XRup uses do not include explicit rules to establish the fact that `(= 1 1)` equals `TRUE`. If `(= 1 1)` happened to be the simplest expression in the equivalence class of `TRUE`, an algorithm that was based completely on the rules of the grammar would have difficulty simplifying the expression TRUE. Difficulties of this sort are probably best overcome through additional restrictions on the simplicity ordering.

9.3. Intersecting the Consequences of Assumption Sets

McAllester (1982b) also describes an interesting way of using grammars to intersect the consequences of different sets of assumptions. If McAllester's algorithms were implemented in XRup, and if they proved computationally tractable, the resulting capability would be quite useful.

As one illustration, consider the following two sets of assumptions:

```
{ (= X A), (= (F A) 1) }
{ (= X B), (= (F B) 1) }
```

The consequence `(= (F X) 1)` follows under either set of assumptions, and hence `(= (F X) 1)` is in the intersection of the consequences of the two assumption sets. (Note that that the intersection of the assumption sets themselves is empty.)

As another illustration, consider these two sets of assumptions:

$$\{\ (=\ (F^5\ X)\ X)\ \}$$
$$\{\ (=\ (F^7\ X)\ X)\ \}$$

The intersection of the consequences of these two assumption sets includes $(=\ (F^{35}\ X)\ X)$, $(=\ (F^{70}\ X)\ X)$, and similar equalities. (Note that it would be difficult to discover equalities like these through search, in a more complicated example.)

If only one possible equality is of interest, it is not necessary to use McAllester's intersection algorithms to determine whether the equality follows from both of two assumption sets. In such a case, one can simply try both sets of assumptions and determine whether the equality follows from both of them.

That method, however, does not provide an explicit representation of the (possibly infinite) equivalence class of an expression. Expression simplification and similar applications require an explicit representation. It is desirable for an expression simplifier to be able to consider two assumption sets and produce the simplest expression that is guaranteed to be equal to a given expression, regardless of which assumption set is actually true. (In the first example above, such a simplifier might simplify (F X) to 1.) If a system with this capability is to use a grammar-based simplification method, it requires a grammar intersection method.

9.4. Beyond One-Level Decomposition of Expressions

The grammar system on which XRup is based is a "one-level" system. With the current grammar design, there is no way to notice relationships that inherently span more than one level of expression parentheses. Although the consequences of equalities can propagate to expressions that are arbitrarily deeper and larger than the expressions mentioned in the equalities, they propagate one level at a time.

Consider a statement like (= PHI (NOT (NOT PHI))). (The current XRup system can derive it, but only because it effectively considers PHI and (NOT (NOT PHI)) to be the same expression. Ignore that feature.) In a grammar that incorporates no premises, PHI and (NOT (NOT PHI)) could be represented as follows:

```
[(NOT (NOT PHI))] ==> ([NOT]* [(NOT PHI)])
[(NOT PHI)] ==> ([NOT]* [PHI])
[PHI] ==> PHI
```

The consequence (= PHI (NOT (NOT PHI))) would follow, if the grammar were more like the following instead:

```
[(NOT PHI)] ==> ([NOT]* [PHI])
[PHI] ==> PHI
       | ([NOT]* ([NOT]* [PHI]))
```

However, this grammar does not satisfy the requirements that define expression grammars. The second rule expanding [PHI] spans more than one level of parentheses, thus violating a restriction. (The methods of assimilating premises into the grammar will not work properly without the restriction.)

In a one-level grammar system, the rule that maps the right-hand side ([NOT]* [(NOT PHI)]) into the corresponding generator [(NOT (NOT PHI))] cannot "see inside" the class [(NOT PHI)] to decompose it into [NOT]* and [PHI]. In fact, looking at the class symbol itself would not be enough, as the following grammar illustrates:

```
[(NOT (NOT PHI))] ==> ([NOT]* [PSI])
[PSI] ==> PSI
        | ([NOT]* [PHI])
[PHI] ==> PHI
```

It does not matter whether the spelling of the class symbol includes NOT; rather, it matters whether any rules expand the class symbol into a right-hand side of the form ([NOT]* x).

One possible approach to this problem might involve keeping per-context data about the *expressibility* of class symbols. In the context containing the immediately preceding grammar, it might be recorded that [PSI] is NOT-expressible, as ([NOT]* [PHI]). Perhaps such data could enable other new mechanisms to determine that the classes of [(NOT (NOT PHI))] and ([PHI]) should be merged.

On the other hand, a more specialized mechanism for dealing with NOT might be appropriate. The current XRup implementation also misses other properties of NOT.* For example, it is unable to perceive that (NOT PHI) and PHI cannot possibly be equal; its current disequality mechanisms are somewhat limited.

The current XRup system also has other small deficiencies in propositional reasoning. For example, if (OR A B) is asserted, the current system will not deduce that (OR A B C) is true. (However, if (OR A B) and (NOT (OR A B C)) are both asserted, it will derive a contradiction.)

9.5. Mechanisms for Control of Reasoning

Demon mechanisms are the most notable feature that XRup lacks in comparison to its historical predecessor (Rup). Demons are special programs that can be dynamically associated with statements (see Winston, 1977). The system activates the demons associated with a statement whenever the statement undergoes a previously specified change in truth value. Demons constitute a powerful mechanism that can be used to control the reasoning of an expert system.

Demon mechanisms were left out of XRup mainly because they were not required for the first application of the system. A more serious consideration is

*Rup also misses many such properties, since it does not consider equalities between logical expressions.

that it would require some research to discover how to best integrate them into the system. The functions of some kinds of demons would need to be reinterpreted in a system like XRup; for example, the function of IF-REMOVED demons would need reinterpretation in a system without retraction. Another problem, however, is that demon mechanisms in XRup could not be implemented as they are implemented in Rup and similar systems.

Rup maintains correct truth values for the entire set of known statements. When a new premise is assimilated, Rup adjusts the stored truth value of every statement that is affected. During this process, it is easy for Rup to activate the demons on the statements whose truth values it is changing.

XRup intentionally avoids maintaining stored truth values over the entire set of known statements. It cannot directly tell which statements change truth values as a result of changes in the grammar. Consequently, XRup cannot associate demons directly with statements, or it will not know when to run them.

One possible alternative might internally associate demons with grammar nonterminals, on a global or a per-context basis. The demons that were associated with a nonterminal could be activated when the nonterminal was replaced with [TRUE]* or [FALSE]*. In order for this scheme to be practical, however, the system would need a way of recognizing demons that are redundant variants of each other.

Suppose, for example, that an automatic mechanism is installing an IF-ADDED demon on every statement of the form (P x). Suppose each such demon contains a pattern-match variable that is bound to x. The demons that are associated with (P A), (P B), (P C), and (P D) are all different because they contain different pattern-match bindings. However, if A, B, C, and D are all equated, then the demons are redundant variants of each other. The system will function inefficiently if it runs them all; it needs a way of eliminating redundant variants from demonic channels just as it eliminates redundant variants from its internal database.

This example brings up the general topic of pattern matching in demon mechanisms. The status of pattern matching in a system like XRup is equally unclear at present. It is undesirable for equivalent variants of a pattern to constitute separate matches. On the other hand, it is essential for the chosen set of matches to span enough combinations for the installed network of demons to accomplish their intended purpose.

9.6. Final Summary

A single object or attribute can have many possible names in a system that deals with equalities. As a result, there can be many variant forms for a single fact. When there are many equalities, it is expensive for a system to give individual consideration to all variant forms.

The XRup system can deal with whole classes of variants at once because it represents variation schematically. Consequently, it often assimilates new facts and assumptions faster than systems that deal with variants individually. The central mechanisms of the system are based on recent grammar-based algorithms that McAllester (1982b) has described but not yet implemented. In the design of XRup, those algorithms have been extended beyond their original form in order to provide supporting premises for derived conclusions, reason about constants, truth values, and logical connectives, and consider temporary assumptions.

The XRup system can also remember the consequences of several assumption sets at once. This capability can speed up applications in which an expert system often considers the same set of assumptions repeatedly.

This thesis describes the design and implementation of the XRup system, and that description is now complete. The thesis has also placed the system in its historical and functional context. (The appendix, which follows, will sketch an experimental application of the system.) This final chapter has presented simple conclusions about the implemented system and has suggested possible future system extensions, such as the addition of mechanisms for the simplification of expressions and the invocation of demons.

10. Appendix: A Performance Example

This appendix briefly discusses the performance of the Rup and XRup systems in an application involving large amounts of equality reasoning. The application was highly experimental, and the operation of the application system did not approach practicality with either Rup or XRup. However, as section 10.2 mentions, the use of XRup did make it possible to run the example to completion.

The performance figures in this appendix are extremely rough, may be downright wrong, and cannot be taken as more than suggestive. They are included only in order to sketch the rough outlines of the performance of XRup in the application that was the driving force for the implementation. The details of the application will not be discussed.

10.1. An Experimental Alerting System

Buneman and Morgan (1977) have introduced the notion of interactively and dynamically defined monitors called *alerters* for databases with changing contents. The author's original Master's Thesis proposal (Barton, 1981) proposed the construction of an *abstract alerting system* that would automate the derivation of low-level alerters from abstract "user-level" specification of the events to be detected. The preliminary design of the system* involves a pre-analysis phase that has as its goal the derivation of large numbers of equalities expressing data-flow relationships among the primitive actions of a database transaction.

The database system is assumed to be similar to the commercially available system described by Gerritsen (1980), which uses a variant of the Codasyl database model. With such a model, most data-flow links go through an area of common memory called the *user work area* (UWA). Consequently, it is possible for a low-level alerter to avoid many database retrievals by reading data from the UWA at crucial points in the transaction. One component of the proposed alerting system would use the data-flow equalities that are derived in the pre-analysis phase to find ways of checking alerter conditions without doing database retrievals. Since it is impossible

*This design was described in a presentation at the author's Oral Qualifying Examination at M.I.T. in November, 1981.

to predict ahead of time what kind of information a user will mention in the specification of alerters, it is necessary for the pre-analysis phase to discover as many data-flow equalities as possible.

Figure 10 shows the database transaction that was analyzed to obtain the data in this appendix. Figure 11 shows a portion of the schema that describes the organization of the database on which the transaction runs. (The database organization was taken from an example in Ullman (1980).) Figure 12 shows a sample data-flow equality that the system derives in the course of analysis. In the notation that the system uses, mutable objects like variables and database records are modeled as functions from computational states to immutable *behaviors.* (Since this appendix is not intended for detailed scrutiny, the details of these figures will not be discussed.)

10.2. Rough Performance Figures

The analysis of the transaction shown in Figure 10 was attempted with two versions of the analysis system. One version used the (single-context) Rup system and used retraction to switch between different sets of assumptions. The other version used the XRup system and used the XRup context mechanism to switch between different sets of assumptions. Each version ran on a Lisp Machine (MIT-APIARY-2) with $512K$ of physical memory.* Figure 13 shows the approximate real time that it took for each version to perform various steps of the computation.

Each version was running with a few statistics counters installed. In the completed XRup computation there were 3969 calls to the object method that adds a premise term to a context and 10329 calls to the right-hand forwarding adjustment method. The forwarding adjustment corresponds to what is called a congruence check in Rup, and hence these figures yield an average of 2.6 congruence checks per premise-term addition in the XRup implementation. In the aborted Rup computation there were 808 calls to the Rup function ASSERT, 53 calls to RETRACT-PREMISE, and 30 calls to NODE-TRY-TO-SHOW, for a total of 891 calls to the

*The comparison is not completely fair because the Rup interface had a minor bug; the Rup version was not run to completion because it encountered an error. It was not run again because of time limitations. Previous attempts to run the Rup version to completion had caused the machine to run out of virtual memory and crash after several hours of paging and computation.

```
; Change or add an item price

(define-update-transaction (note-item-price hvfc-schema)
                           ((item :string)
                            (supplier :string)
                            (price :float))
  (prog ()
    (begin-transaction)

    ; find the item record
    ; find the prices record owned by the item in itempr that
    ;   corresponds to the given supplier
    ;   if prices record found, modify it
    ;   if no prices record found, add it and link it into the
    ;   sets

    ; find the item record
    (setf (uwa-ref (iname items)) item)
    (find-record-by-calc-key items)

    ; look through the prices for the item
    (permitting-database-exceptions
     (find-first-record-in-current-set itempr))
    (conditional-transfer (null (error-status)) loop)
    (conditional-transfer (equal (error-status)
                                 '(:find :set-empty))
                          new-supplier)
    (abort-transaction)

  loop
    (find-owner-of-current-record suppr)
    (get-current-program-record suppliers)
    (conditional-transfer (samepnamep
                            (uwa-ref (sname suppliers))
                            supplier)
                          got-prices-record)
    (permitting-database-exceptions
     (find-next-record-in-current-set itempr))
    (conditional-transfer (null (error-status)) loop)
    (conditional-transfer (equal (error-status)
                                 '(:find :no-more-members))
                          new-supplier)
    (abort-transaction)

   got-prices-record
    (setf (uwa-ref (price prices)) price)
    (find-current-of-record prices)
    (modify-current-program-record prices)
    (go commit)

    ; here we didn't find a prices record
  new-supplier
    (setf (error-status) nil)
    (setf (uwa-ref (sname suppliers)) supplier)
    (find-record-by-calc-key suppliers)
    (setf (uwa-ref (price prices)) price)
    (store-new-record prices)
  commit
    (commit-transaction)))
```

Figure 10. This figure shows the definition of the database transaction that was analyzed to obtain the performance figures in this appendix. The transaction records a price for an item from a supplier, searching through a portion of the database to find the record where the item price is stored. It creates a new price record if there is no old one.

```
(define-database-schema hvfc-schema
   (:record suppliers
      (:field sname :string)
      (:field saddr :string)
      (:calc-key sname))
   (:record items
      (:field iname :string)
      (:field order-units :string)
      (:calc-key iname))
   (:record prices
      (:field price :float))
   (:set suppr
      (:owner suppliers)
      (:member prices))
   (:set itempr
      (:owner items)
      (:member prices))
   ...)
```

Figure 11. This figure shows a portion of the schema for the database on which the transaction shown in the previous figure is intended to run. The major represented entities are items and suppliers.

user-level functions that correspond to changes in the premise set. There were 9586 calls to `INITIAL-CONGRUENCE-CHECK`, 46558 calls to `CONGRUENCE-CHECK`, and 129151 calls to `NON-MEM-CONGRUENCE-CHECK`. This makes a total of 185295 congruence checks and yields an average of 208.0 congruence checks per premise-set change in the Rup implementation. (The user-level addition of clauses was inadvertently neglected because clause addition is implemented by the same functions as premise addition in the XRup system. If user-level clause addition had been properly included in the Rup figures, the number of congruence checks per premise-set change would have been smaller.) XRup was expected to do a smaller number of congruence checks per premise-term addition because of its compact representation for equivalence classes.

Before the Rup example was run, an efficiency heuristic was removed. The heuristic relied on the assumption that some kinds of terms would never be equated or compared for equality. That heuristic would make Rup run faster, but would also prevent the proper conclusions from being derived. With `(= X Y)` asserted,

```
(= (UWA-ITEMS-INAME
     (IN
       (FIND-1 REPRESENTATIVE-NOTE-ITEM-PRICE)))
   (ITEMS-INAME-FIELD
     ((OWNER
        (ITEMPR-OCCURRENCE
          ((CURRENT-OF-RUN-UNIT
             (IN
               (MODIFY REPRESENTATIVE-NOTE-ITEM-PRICE)))
           (OUT REPRESENTATIVE-NOTE-ITEM-PRICE))))
      (IN REPRESENTATIVE-NOTE-ITEM-PRICE))))
```

Figure 12. This figure shows one of the data-flow equalities that can be derived for the transaction shown in Figure 10. The first argument of the equality denotes the value of the ITEMS-INAME *string variable in the UWA before the first* FIND *operation of the transaction. The second argument of the equality denotes the result of noticing what record is pointed to by the* CURRENT-OF-RUN-UNIT *variable before the* MODIFY *operation, finding which* ITEMPR *set occurrence that record is in at the end of the transaction, finding the* ITEMS *record that is the* OWNER *of that set occurrence, and retrieving the value that was in the* INAME *field of that record at the beginning of the transaction. These two values must be equal if the* MODIFY *operation is reached and the transaction does not abort with error, but the first value is easier to retrieve than the second. The analysis system must reason about loops in order to derive this equality.*

the heuristic prevents the derivation of equality between `(= (F X) (F Z))` and `(= (F Y) (F Z))`. It also causes LISP errors when two expressions of the restricted kinds are asserted to be equal.

The XRup implementation also contained statistics counters to monitor the performance of the generator cache that was described in section 8.7. Each context had a cache for the generators of 500 terms. In the completed XRup computation, there were 88088 attempted cache retrievals, and 33926 (39%) of those operations found the generator of the desired expression in the cache. An additional 21094 operations used partial information in the cache to avoid complete generator recomputation.

XRup Step Start Time	Rup Step Start Time	XRup Step Time	Rup Step Time	Rup/XRup Step Time Ratio	Reference Event Description
0	0	4	2	0.5	Computation begins
4	2	5	7	1.4	Reachedness splicing fails
9	9	1	14	14.0	Loop splicing fails for modified vars
10	23	7	34	4.9	Join splicing fails for modified vars
17	57	5	> 9	> 1.8	Loop splicing pass begins
--	65	...	...		Patience is exhausted in Rup example
23	--	1	--		Second loop splicing pass begins
24	--	9	--		Loop splicing for objects succeeds
33	--	5	--		Join splicing for objects succeeds
38	--	26	--		Frequent try-to-show calls begin
64	--	7	--		Object join splicing at commit succeeds
71	--	4	--		Expression splicing begins
75	--				Computation finishes

Figure 13. This table shows the approximate real time (in minutes) that it took for the Rup and XRup versions of the database analysis program to perform various steps in the analysis of the transaction shown in Figure 10. The Rup version was not run to completion, and had a minor bug. A step is the time between two listed reference events; the reference events were chosen as significant points in the analysis.

11. Bibliography

Barton, G. Edward, Jr. (1981). "The Construction of an Abstract Alerting System." Unpublished M.S. thesis proposal submitted to the Department of Electrical Engineering and Computer Science, M.I.T., Cambridge, Mass.

Buneman, O. P., and H. L. Morgan (1977). "Implementing Alerting Techniques in Database Systems." Working Paper 77-03-04, Department of Decision Sciences, Wharton School, University of Pennsylvania.

Davis, Martin, and Hilary Putnam (1960). "A Computing Procedure for Quantification Theory." *JACM* 7, 201-215.

Davis, Randall (1982). "Expert Systems: Where Are We and Where Do We Go From Here?" AI Memo 665, M.I.T. Artificial Intelligence Laboratory, Cambridge, Mass.

Downey, P. J., R. Sethi, and R. E. Tarjan (1980). "Variations on the Common Subexpression Problem." *JACM* 27, 4, 758-771.

Doyle, Jon (1977). *Truth Maintenance Systems for Problem Solving.* M.S. Thesis, Department of Electrical Engineering and Computer Science, M.I.T., Cambridge, Mass.

—, and P. London (1980). "A Selected Descriptor-Indexed Bibliography to the Literature on Belief Revision." AI Memo 568, M.I.T. Artificial Intelligence Laboratory, Cambridge, Mass.

Fahlman, Scott (1979). *NETL: A System for Representing and Using Real-World Knowledge.* Cambridge: M.I.T. Press.

Gerritsen, R. (1980). *SEED Reference Manual.* Philadelphia: International Data Base Systems, Inc.

Hawkinson, Lowell (1980). "XLMS: A Linguistic Memory System." Technical Memorandum LCS-TM-173, M.I.T. Laboratory for Computer Science, Cambridge, Mass.

McAllester, D. A. (1980). *The Use of Equality in Deduction and Knowledge Representation.* AI Technical Report TR-550, M.I.T. Artificial Intelligence Laboratory, Cambridge, Mass.

— (1981a). "An Outlook on Truth Maintenance." AI Memo 551, M.I.T. Artificial Intelligence Laboratory, Cambridge, Mass.

— (1981b). "Solving Uninterpreted Equations." Unpublished paper dated June 1981, M.I.T. Artificial Intelligence Laboratory.

— (1982a). "Reasoning Utility Package User's Manual." AI Memo 667, M.I.T. Artificial Intelligence Laboratory, Cambridge, Mass.

— (1982b). "Solving Uninterpreted Equations with Context-Free Expression Grammars." Manuscript dated September, 1982; will appear in 1983 as AI Memo 708, M.I.T. Artificial Intelligence Laboratory, Cambridge, Mass.

McDermott, D., and Jon Doyle (1978). "Non-Monotonic Logic I." AI Memo 486, M.I.T. Artificial Intelligence Laboratory, Cambridge, Mass.

McNaughton, Robert (1967). "Parenthesis Grammars." *JACM* 14, 3, 490-500.

Quine, W. V. O., and J. S. Ullian (1978). *The Web of Belief (second edition).* New York: Random House.

Shrobe, H. E. (1978). *Logic and Reasoning for Complex Program Understanding.* Ph.D. Thesis, Department of Electrical Engineering and Computer Science, M.I.T., Cambridge, Mass.

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

Stefik, Mark, Jan Aikins, Robert Balzer, John Benoit, Lawrence Birnbaum, Frederick Hayes-Roth, and Earl Sacerdoti (1982). "The Organization of Expert Systems, A Tutorial." *Artificial Intelligence* 18, 135-173.

Sussman, G. J., T. Winograd, and E. Charniak (1971). "Micro-Planner Reference Manual." AI Memo 203a, M.I.T. Artificial Intelligence Laboratory, Cambridge, Mass.

Ullman, J. D. (1980). *Principles of Database Systems.* Potomac, Md.: Computer Science Press.

Waters, Richard C. (1981). "GPRINT – A LISP Pretty Printer Providing Extensive User Format-Control Mechanisms." AI Memo 611a, M.I.T. Artificial Intelligence Laboratory, Cambridge, Mass.

Weinreb, Daniel, David Moon, and Richard Stallman (1983). *Lisp Machine Manual,* fifth edition. Available from the M.I.T. Artificial Intelligence Laboratory, Cambridge, Mass.

Winston, P. H. (1977). *Artificial Intelligence.* Reading, Mass.: Addision-Wesley.

CS-TR Scanning Project
Document Control Form

Date : 2 / 15 / 96

Report # AI-TR-715

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

☒ Artificial Intellegence Laboratory (AI)
☐ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:

Document Information

Number of pages: 145 (153-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☒ Single-sided or
☐ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☐ Unknown ☒ Other:

Check each if included with document:

☒ DOD Form (2) ☐ Funding Agent Form ☒ Cover Page
☒ 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:
(A) IMAGE MAP: (1-145)	1-145
(146-153)	SCANCONTROL, COVER, SPINE, DOD(2), TRGT'S (3)
(B) Cut&Paste Fig's on pages 9, 11, 18, 31, 82-85, 87-88, 140-143.	

Scanning Agent Signoff:
Date Received: 2/15/96 Date Scanned: 2/22/96 Date Returned: 2/27/96

Scanning Agent Signature: [illegible]

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 AI-TR-715	2. GOVT ACCESSION NO.	3. RE[illegible] NUMBER AD-A132369
4. TITLE *(and Subtitle)* A Multiple -Context Equality-Based Reasoning System		5. TYPE OF REPORT & PERIOD COVERED technical report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) George Edward Barton, Jr.		8. CONTRACT OR GRANT NUMBER(s) N00014-80-C-0505
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, Massachusetts 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd Arlington, Virginia 22209		12. REPORT DATE April 1983
		13. NUMBER OF PAGES 145
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)* Office of Naval Research Information Systems Arlington, Virginia 22217		15. SECURITY CLASS. *(of this report)* UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Distribution of this document is unlimited.

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

18. SUPPLEMENTARY NOTES

None

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

equality reasoning   truth maintenance systems
automated deduction
parenthesis grammars
propositional reasoning

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

Expert systems are too slow. This work attacks that problem by speeding up a useful system component that remembers facts and tracks down simple consequences. The redesigned component can assimilate new facts more quickly because it uses a compact, grammar-based internal representation to deal with whole classes of equivalent expressions at once. It can support faster hypothetical reasoning because it remembers the consequences of several assumption sets at once. The new design is targeted for situations in which many of the stored facts are equalities. OVER

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

UNCLASSIFIED

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

The deductive machinery considered here supplements stored premises with simple new conclusions. The stored premises include permanently asserted facts and temporarily adopted assumptions. The new conclusions are derived by substituting equals for equals and using the properties of the logical connectives AND, OR and NOT. The deductive system provides supporting premises for its derived conclusions. Reasoning that involves quantifiers is beyond the scope of its limited and automatic operation. The expert system of which the reasoning system is a component is expected to be responsible for overall control of reasoning.